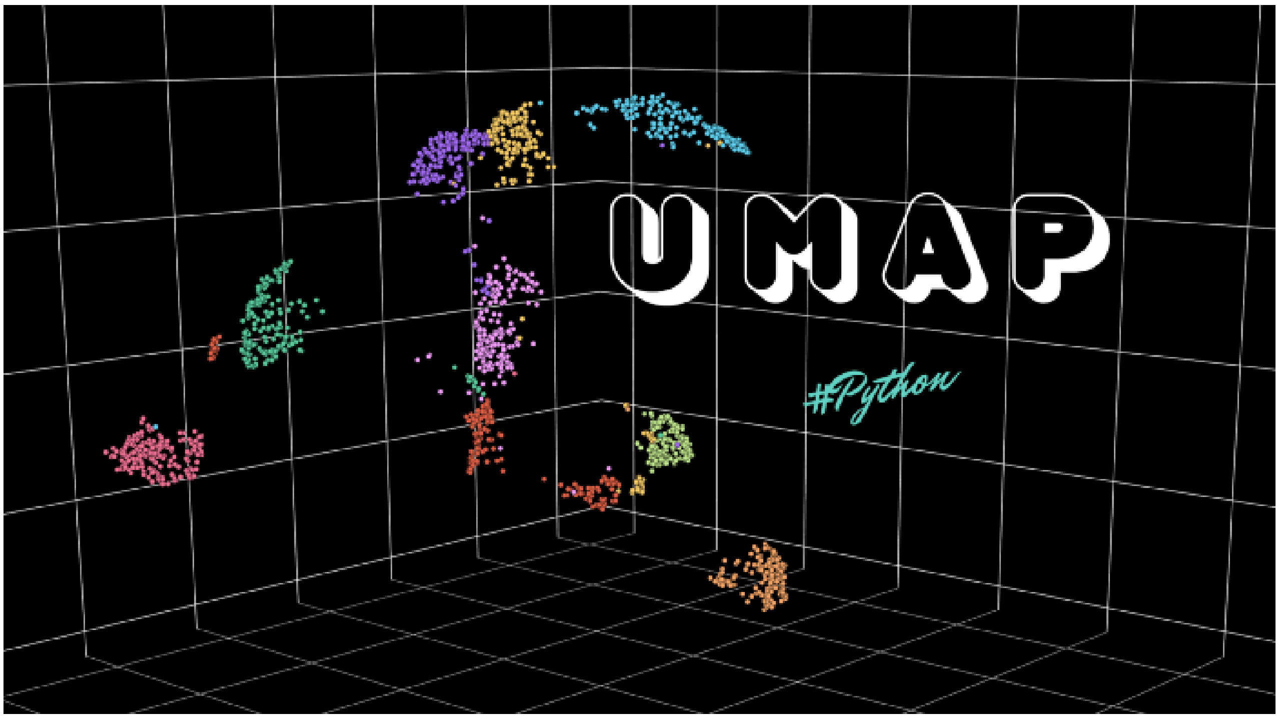
UMAP Dimensionality Reduction — An Incredibly Robust Machine Learning Algorithm

towardsdatascience.com/umap-dimensionality-reduction-an-incredibly-robust-machine-learning-algorithm-b5acb01de568

November 13, 2021

Hands-on Tutorials, Machine Learning

How does Uniform Manifold Approximation and Projection (UMAP) work, and how to use it in Python



Uniform Manifold Approximation and Projection (UMAP). Image by .

Intro

Dimensionality reduction is not just for data visualization. It can also help you overcome the "curse of dimensionality" by identifying critical structures in the high-dimensional space and preserving them in the lower-dimensional embedding.

This article will take you through the inner workings of an increasingly popular dimensionality reduction technique called **Uniform Manifold Approximation and Projection (UMAP)** and provide you with a Python example that can be used as a guide when working on your Data Science projects.

Contents

Uniform Manifold Approximation and Projection (UMAP) in the universe of Machine Learning algorithms

The below sunburst chart is my attempt to categorize the most commonly used Machine Learning algorithms. I have created it to satisfy the need of Data Scientists like you and me to have a more structured and visual way of identifying how various algorithms fit together.

This particular view of the Machine Learning universe enables us to see similarities and differences between algorithms, serving as a quick guide when looking for a suitable solution for a specific project.

The graph is **interactive** so make sure to explore it by clicking on different categories.👇

Machine Learning algorithm classification. Interactive chart created by the .

As you can see, I have placed UMAP at the border of **supervised** and **unsupervised** dimensionality reduction techniques. While UMAP is most commonly used for unsupervised learning, it can also perform supervised dimensionality reduction. You will find an example of that in the Python section at the end of this article.

How does Uniform Manifold Approximation and Projection (UMAP) work?

Analyzing the UMAP name

Let's start by dissecting the UMAP name, which will give us a broad idea of what the algorithm is supposed to do.

> Note, the below statements are not official definitions but rather a set of descriptions that will help us understand key ideas behind UMAP.

- — the process or technique of reproducing a spatial object upon a plane, a curved surface, or a line by projecting its points. You can also think of it as a mapping of an object from high-dimensional to low-dimensional space.
- — the algorithm assumes that we only have a finite set of data samples (points), not the entire set that makes up the manifold. Hence, we need to approximate the manifold based on the data available.

- — a manifold is a topological space that locally resembles Euclidean space near each point. One-dimensional manifolds include lines and circles, but not figure eights. Two-dimensional manifolds (a.k.a. surfaces) include planes, spheres, torus, and more.
- — the uniformity assumption tells us that our data samples are uniformly (evenly) distributed across the manifold. In the real world, however, this is rarely the case. Hence, this assumption leads to the notion that the distance varies across the manifold. i.e., the space itself is warping: stretching or shrinking according to where the data appear sparser or denser.

Putting the above statements together, we can describe UMAP as:

> A dimensionality reduction technique that assumes the available data samples are evenly (**uniformly**) distributed across a topological space (**manifold**), which can be **approximated** from these finite data samples and mapped (**projected**) to a lower-dimensional space.

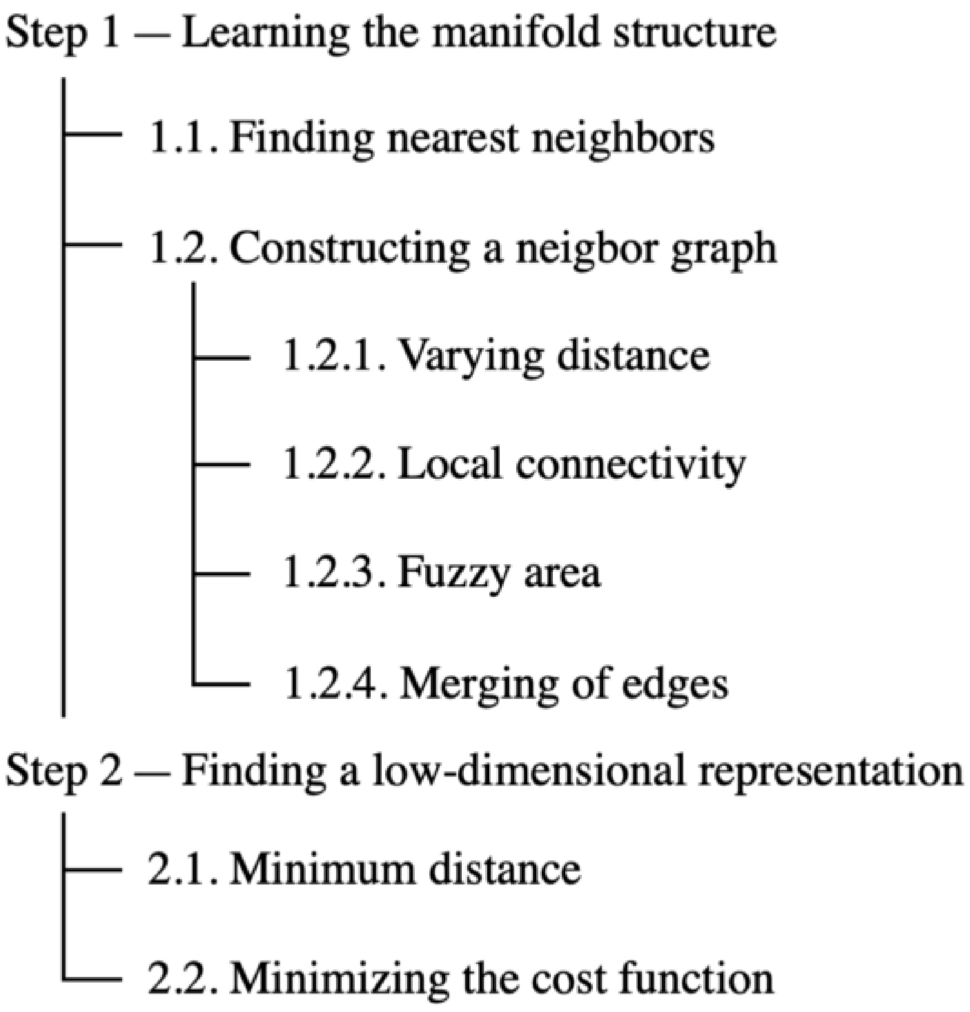
The above description of the algorithm may help a little, but it is still vague on how UMAP does its magic. So, to answer the “how” question, let’s analyze the individual steps that UMAP performs.

High-level steps performed by UMAP

We can split UMAP into two major steps:

1. learning the manifold structure in the high-dimensional space;
2. finding a low-dimensional representation of said manifold.

We will, however, break this down into even smaller components to make our understanding of the algorithm deeper. The below map shows the order that we will go through in analyzing each piece.


Step 1 — Learning the manifold structure
1.1. Finding nearest neighbors
1.2. Constructing a neigbor graph
1.2.1. Varying distance
1.2.2. Local connectivity
1.2.3. Fuzzy area
1.2.4. Merging of edges
Step 2 — Finding a low-dimensional representation
2.1. Minimum distance
2.2. Minimizing the cost function

UMAP steps and components — image by .

Step 1 — Learning the manifold structure

It will come as no surprise, but before we can map our data to lower dimensions, we first need to figure out what it looks like in the higher-dimensional space.

1.1. Finding nearest neighbors
UMAP starts by finding the nearest neighbors using the Nearest-Neighbor-Descent algorithm of Dong et al. You will see in the Python section later on that we can specify how many nearest neighbors we want to use by adjusting UMAP's hyperparameter.

It is important to experiment with the number of because it **controls how UMAP balances local versus global structure in the data**. It does it by constraining the size of the local neighborhood when attempting to learn the manifold structure.

Essentially, a small value for means that we want a very local interpretation that accurately captures the fine detail of the structure. In contrast, a large value means that our estimates will be based on larger regions, thus more broadly accurate across the manifold as a whole.

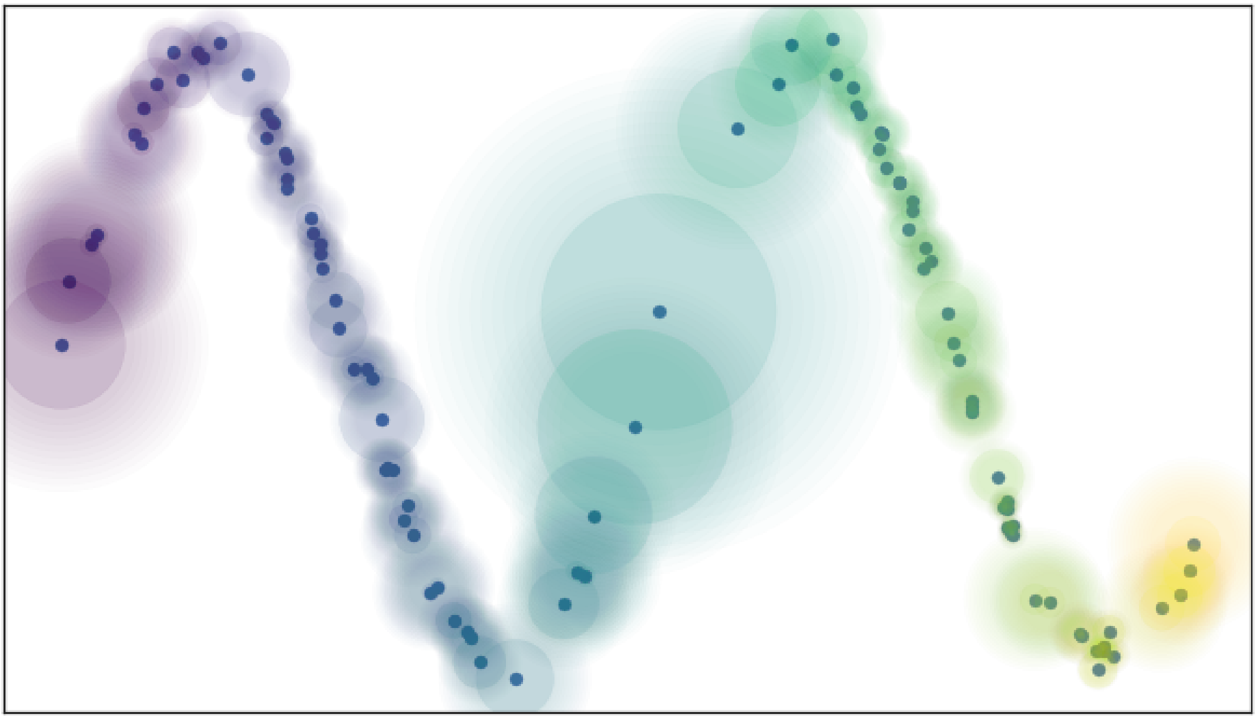
1.2. Constructing a graph

Next, UMAP needs to construct a graph by connecting the previously identified nearest neighbors. To understand this process, we need to look at a few sub-components that explain how the neighborhood graph comes to be.

1.2.1. Varying distance

As outlined in the analysis of the UMAP's name, we assume a uniform distribution of points across the manifold, suggesting that space between them is stretching or shrinking according to where the data appears to be sparser or denser.

It essentially means that the distance metric is not universal across the whole space, and instead, it varies between different regions. We can visualize it by drawing circles/spheres around each data point, which appear to be different in size because of the varying distance metric (see illustration below).



Local connectivity and fuzzy open sets. Image source: .

1.2.2. Local connectivity

Next, we want to ensure that the manifold structure we are trying to learn does not result in many unconnected points. Luckily, we can use another hyperparameter called (default value = 1) to solve this potential problem.

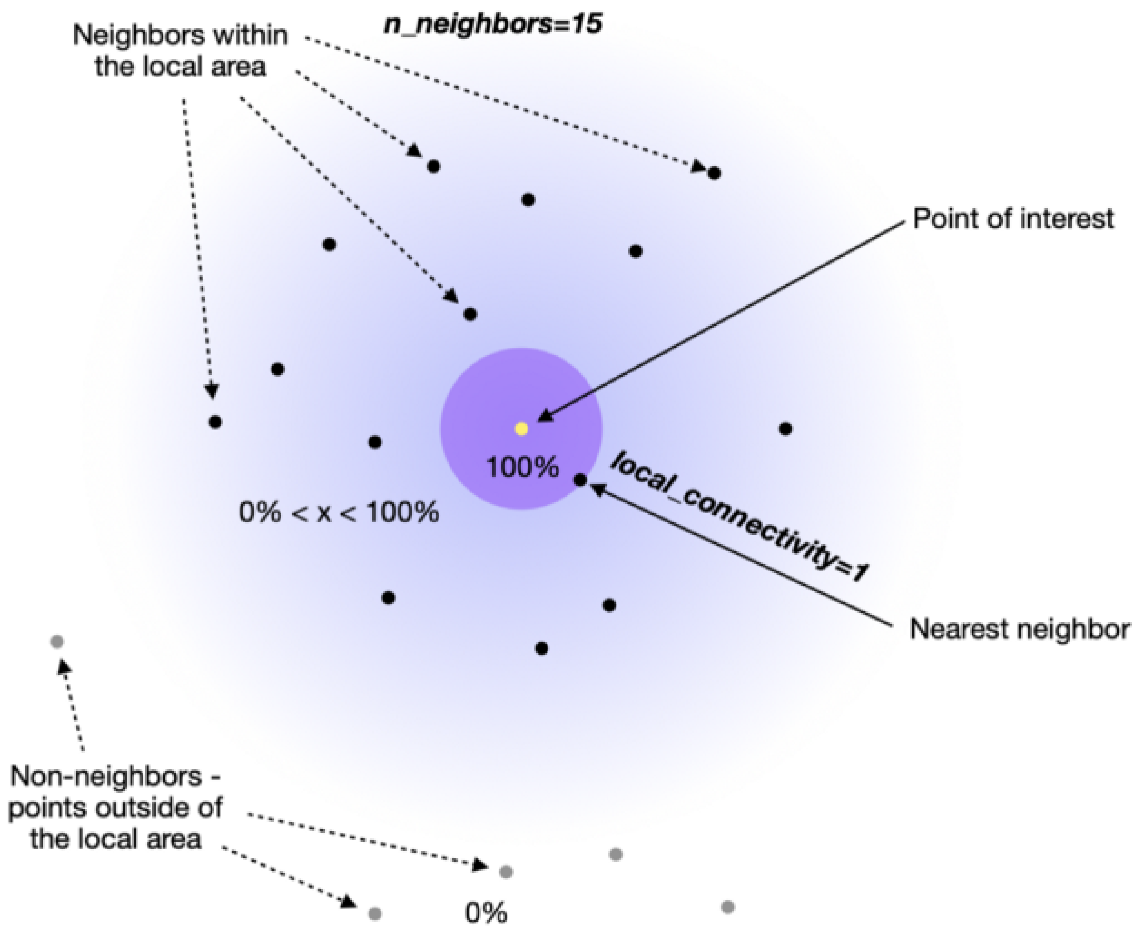
When we set we tell the algorithm that every point in the higher-dimensional space is connected to at least one other point. You can see in the above illustration how each solid circle touches at least one data point.

1.2.3. Fuzzy area

You must have noticed that the illustration above also contains fuzzy circles extending beyond the closest neighbor. This tells us that the certainty of connection with other points decreases as we get farther away from the point of interest.

The easiest way to think about it is by viewing the two hyperparameters (and) as lower and upper bounds:

- — there is 100% certainty that each point is connected to at least one other point (lower limit for a number of connections).
- — there is a 0% chance that a point is directly connected to a 16th+ neighbor since it falls outside the local area used by UMAP when constructing a graph.
- — there is some level of certainty (>0% but <100%) that a point is connected to its 2nd to 15th neighbor.

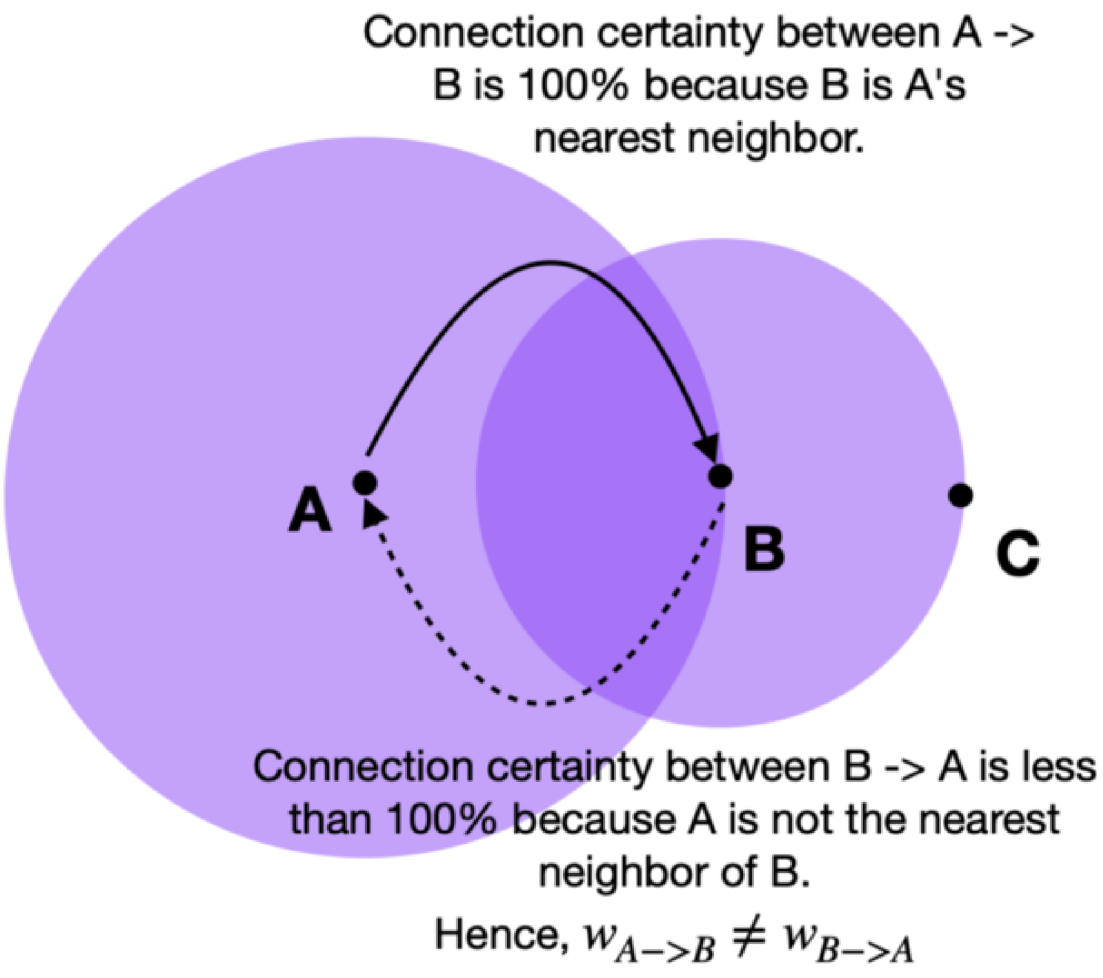


Neighborhood illustration. Image by .

1.2.4. Merging of edges

Finally, we need to understand that the connection certainty discussed above is expressed through edge weights ().

Since we have employed a varying distance approach, we will unavoidably have cases where edge weights do not align when viewed from the perspective of each point. E.g., the edge weight for points A→ B will be different from the edge weight of B→ A.


Connection certainty between A -> B is 100% because B is A's nearest neighbor.
A
B
C
Connection certainty between B -> A is less than 100% because A is not the nearest neighbor of B.
Hence, $w_{A->B} \neq w_{B->A}$

Disagreeing edge weights. Image by .

UMAP overcomes the problem of disagreeing edge weights we just described by taking a union of the two edges. Here is how UMAP documentation explains it:

> If we want to merge together two disagreeing edges with weight a and b then we should have a single edge with combined weight $a+b-a\cdot b$. The way to think of this is that the weights are effectively the probabilities that an edge (1-simplex) exists. The combined weight is then the probability that at least one of the edges exists.

In the end, we get a connected neighborhood graph that looks like this:

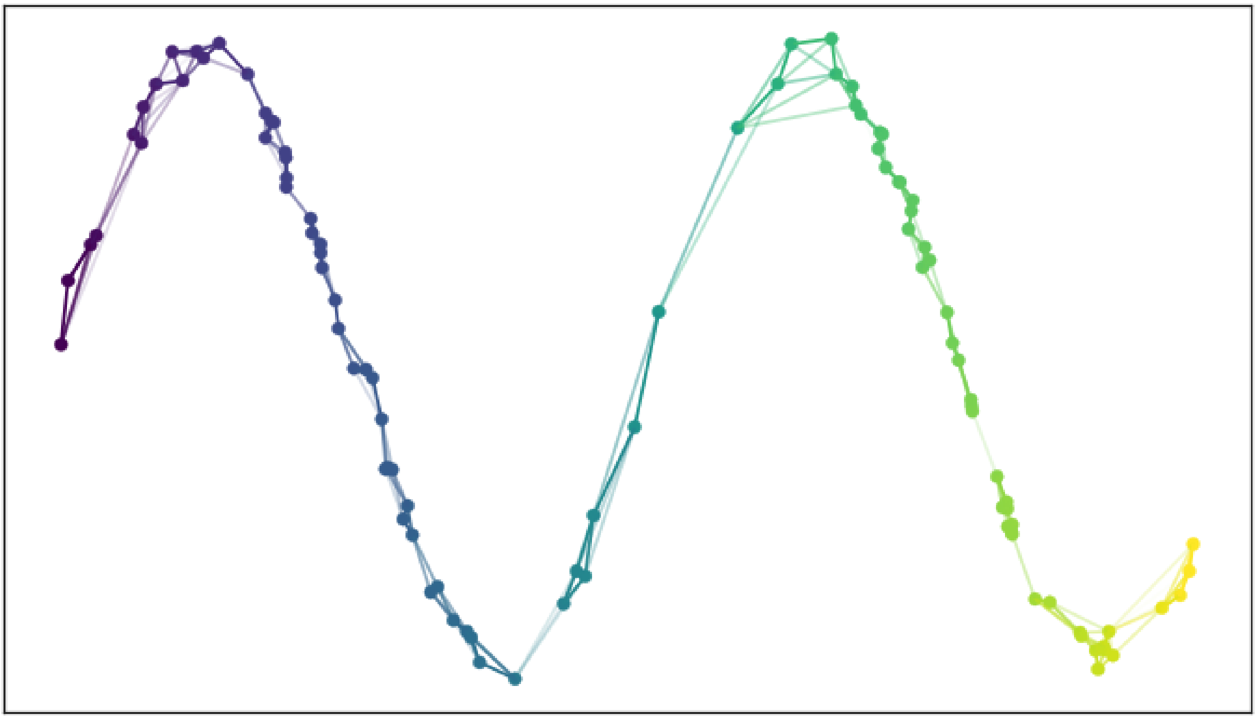


Image source: .

Step 2 — Finding a low-dimensional representation

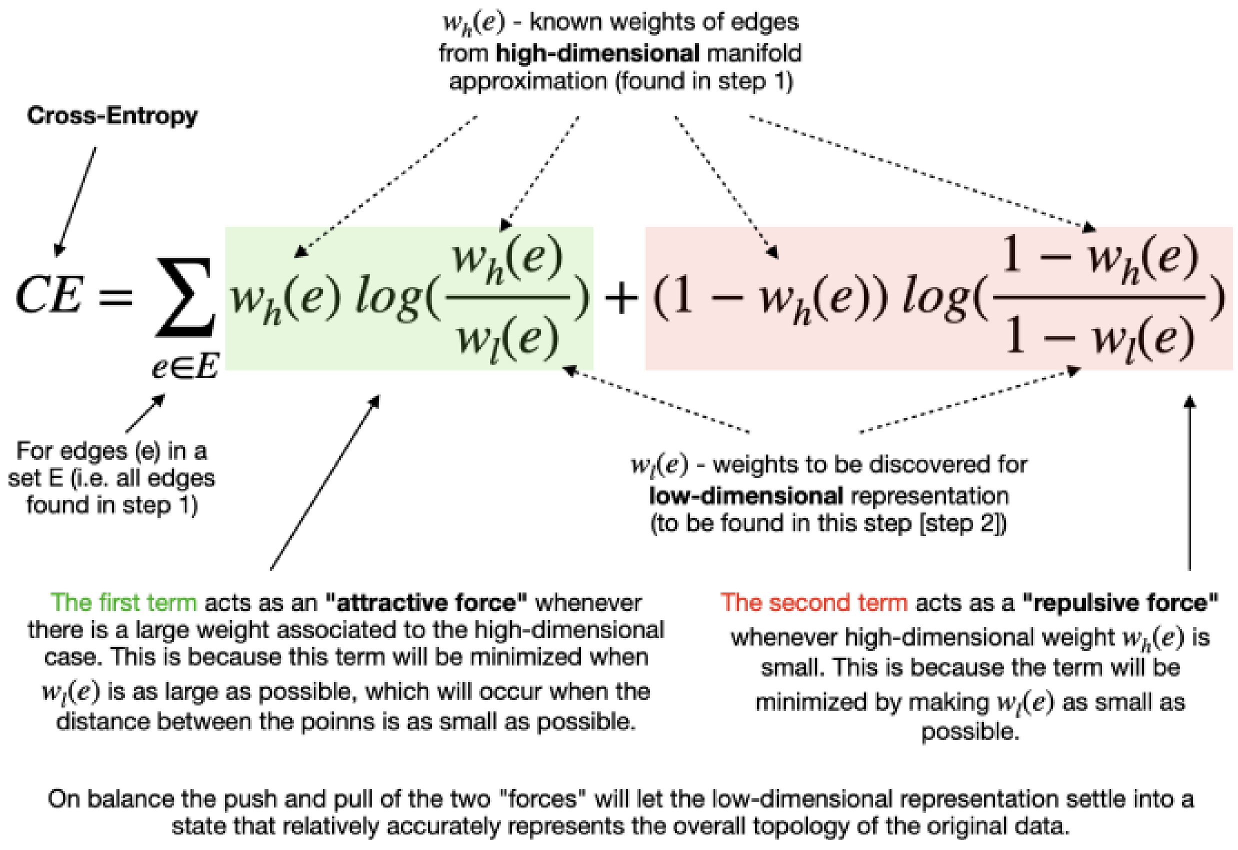
After learning the approximate manifold from the higher-dimensional space, the next step for UMAP is to project it (map it) to a lower-dimensional space.

2.1. Minimum distance Unlike the first step, we do not want varying distances in the lower-dimensional space representation. Instead, we want the distance on the manifold to be standard Euclidean distance with respect to the global coordinate system.

The switch from varying to standard distances also impacts the proximity to nearest neighbors. Hence, we must pass another hyperparameter called *(default=0.1)* to define the minimum distance between embedded points.

Essentially, we can control the minimum spread of points, avoiding scenarios with many points sitting on top of each other in the lower-dimensional embedding.

2.2. Minimizing the cost function (Cross-Entropy) With the minimum distance specified, the algorithm can start looking for a good low-dimensional manifold representation. UMAP does it by minimizing the following cost function, also known as Cross-Entropy (CE):


$w_h(e)$ - known weights of edges from **high-dimensional** manifold approximation (found in step 1)
Cross-Entropy
$$CE = \sum_{e \in E} w_h(e) \log\left(\frac{w_h(e)}{w_l(e)}\right) + (1 - w_h(e)) \log\left(\frac{1 - w_h(e)}{1 - w_l(e)}\right)$$
For edges (e) in a set E (i.e. all edges found in step 1)
$w_l(e)$ - weights to be discovered for **low-dimensional** representation (to be found in this step [step 2])
The first term acts as an **"attractive force"** whenever there is a large weight associated to the high-dimensional case. This is because this term will be minimized when $w_l(e)$ is as large as possible, which will occur when the distance between the poinns is as small as possible.
The second term acts as a **"repulsive force"** whenever high-dimensional weight $w_h(e)$ is small. This is because the term will be minimized by making $w_l(e)$ as small as possible.
On balance the push and pull of the two "forces" will let the low-dimensional representation settle into a state that relatively accurately represents the overall topology of the original data.

Minimizing the Cross-Entropy cost equation. Image by .

As you can see, the ultimate goal is to **find the optimal weights of edges in the low-dimensional representation**. These optimal weights emerge as the above Cross-Entropy cost function is minimized following an iterative stochastic gradient descent process.

And that is it! The UMAP's job is now complete, and we are given an array containing the coordinates of each data point in a specified lower-dimensional space.

Using UMAP in Python

Finally, we can use our newly acquired knowledge of UMAP to perform dimensionality reduction in Python.

We will apply UMAP on the MNIST dataset (a collection of handwritten digits) to illustrate how we can successfully separate digits and display them in the low-dimensional space.

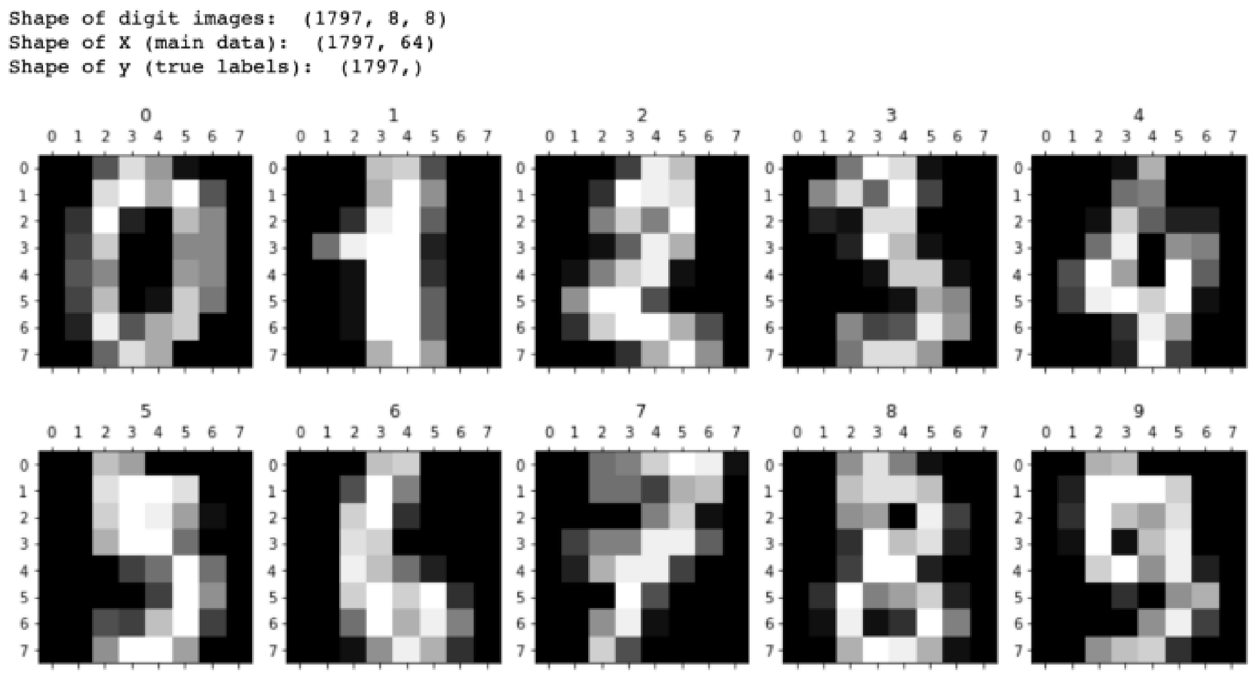
Setup

We will use the following data and libraries:

- for1) MNIST digit data ();2) splitting data into train and test samples ();
- library for performing dimensionality reduction;
- and for data visualizations;
- and for data manipulation.

The first step is to import the libraries that we have listed above.

Next, we load the MNIST data and display the images of the first ten handwritten digits.



```
Shape of digit images:  (1797, 8, 8)
Shape of X (main data):  (1797, 64)
Shape of y (true labels):  (1797,)
```

Images of the first ten handwritten digits (8x8=64 pixels, i.e., 64 dimensions). Image by .

Next, we will create a function for drawing 3D scatter plots that we can reuse multiple times to display results from UMAP dimensionality reduction.

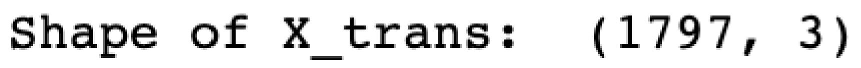
Applying UMAP to our data

Now, we take the MNIST digit data that we have previously loaded into an array X. The shape of X (1797,64) tells us that we have 1,797 digits, each made up of 64 dimensions.

We will use UMAP to reduce the dimensionality from 64 down to 3. Please note that I have listed every hyperparameter available in UMAP with a short explanation of what they do.

While, in this example, I am leaving most of the hyperparameters set to their default values, I encourage you to experiment with them to see how they affect the results.

The above code applies UMAP to our MNIST data and prints the shape of the transformed array to confirm that we have successfully reduced dimensions from 64 to 3.



```
Shape of X_trans:  (1797, 3)
```

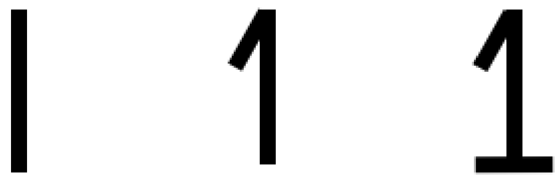
The shape of the new transformed array. Image by .

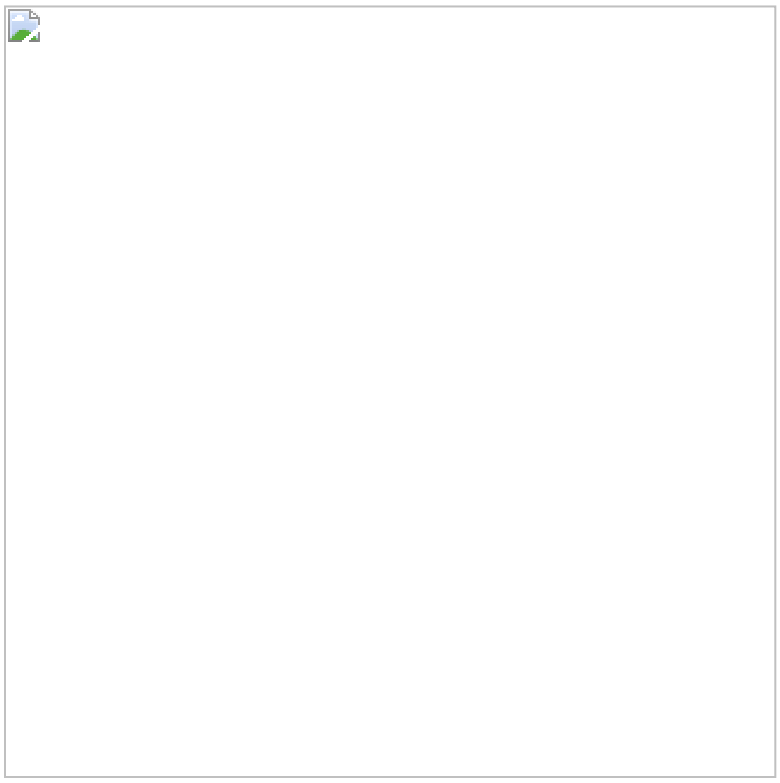
We can now use the chart plotting function we created earlier to visualize our 3-dimensional digits data. We call the function with a simple line of code passing the arrays we want to visualize.

```
chart(X_trans, y)
```

Results of the dimensionality reduction. Chart by .

The results look pretty great, with a clear separation between the digit clusters. Interestingly digit 1 formed three distinct clusters, which can be explained by different variations of how people write digit 1:





Different ways of writing digit 1. Image by .

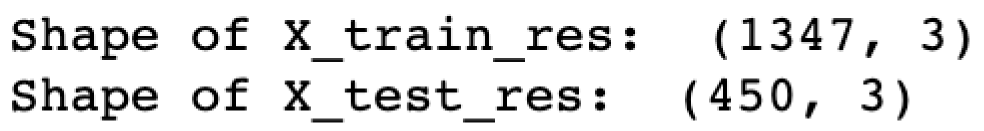
Note how writing 1 with a base at the bottom makes it looks similar to digit 2. We can find these cases in a small red cluster of 1's located very close to the green cluster of 2's.

Supervised UMAP

As mentioned at the beginning of the article, we can also use UMAP in a supervised manner to help reduce the dimensions of our data.

When performing supervised dimensionality reduction, in addition to image data (X_train array), we also need to pass labels data (y_train array) into a method (see code below).

Also, I want to bring your attention to the fact that I have made a couple of other minor changes to hyperparameters, setting and for nicer visualization and better results on a test sample.



```
Shape of X_train_res:  (1347, 3)
Shape of X_test_res:  (450, 3)
```

The shape of the transformed train and test arrays. Image by author.

Now that we have successfully reduced dimensions using the supervised UMAP approach, we can plot 3D scatterplots to show the results.

```
chart(X_train_res, y_train)
```

Results of the dimensionality reduction — . Chart by .

We can see that UMAP has formed very tight clusters of each digit separated by a considerable distance.

We now create the same 3D graph for the test data to see if UMAP could successfully place new data points into these clusters.

```
chart(X_test_res, y_test)
```

Results of the dimensionality reduction — Chart by .

As you can see, the results are pretty good, with only a few digits placed in the wrong clusters. In particular, it looks like the algorithm struggled with digit 3, with a few examples located next to 7's, 8's, and 5's.

Conclusion

I appreciate you reading this long article, and I hope that every part of it has given you more insight into how this great algorithm operates.

In general, UMAP has a solid mathematical foundation, and it can often do a better job than similar dimensionality reduction algorithms such as t-SNE.

The secret lies in UMAP's ability to infer both **local** and **global** structures while preserving relative global distances in the lower-dimensional space. These abilities enable us to unlock specific insights, such as finding similarities between the handwritten forms of digits 1 and 2.

Please feel free to share your thoughts and feedback, which will help me write better articles in the future.

Cheers 👏
Saul Dobilas

How would you like to become a **Medium member** to gain full access to my articles as well as stories from thousands of other writers? If so, please consider using my personalized link below, and Medium will share a portion of your subscription fee directly with me: https://solclover.com/membership

Other articles you may find interesting:

t-SNE Machine Learning Algorithm — A Great Tool for Dimensionality Reduction in Python

How to use t-Distributed Stochastic Neighbor Embedding (t-SNE) to visualize high-dimensionality data?

towardsdatascience.com

LLE: Locally Linear Embedding — A Nifty Way to Reduce Dimensionality in Python

A detailed look into how LLE works and how it compares to similar algorithms such as Isomap

towardsdatascience.com