**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?

*Note, the below statements are not official definitions but rather a set of descriptions that help us understand key ideas behind t-SNE.*

* **Embedding**— typically high-dimensional data represented in a lower-dimensional space;
* **Neighbor**— a data point that resides close to the data point of interest;
* **Stochastic** —the use of randomness in the iterative process when searching for a representative embedding;
* **t-Distributed**— probability distribution used by the algorithm to calculate similarity scores in the lower-dimensional embedding.

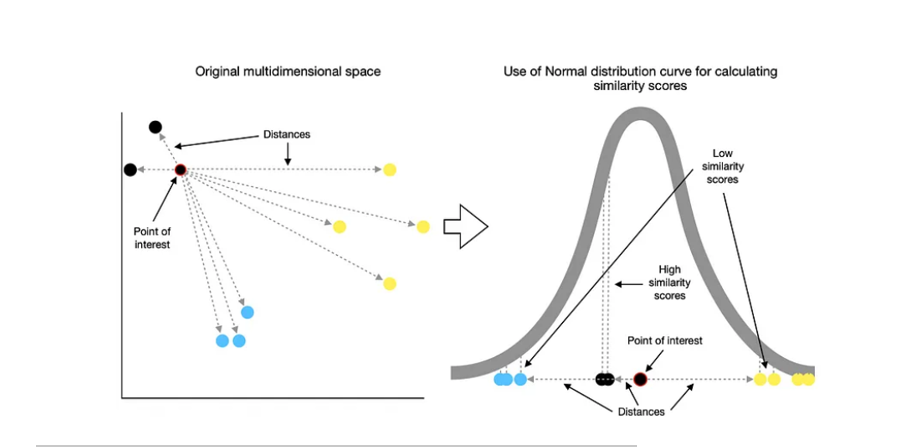
# Putting the above statements together, we can describe t-SNE as a technique that utilizes a gradual iterative approach to find a lower-dimensional representation of the original data while preserving information about local neighborhoods.

# t-SNE is something called nonlinear dimensionality reduction. What that means is this algorithm allows us to separate data that cannot be separated by any straight line.

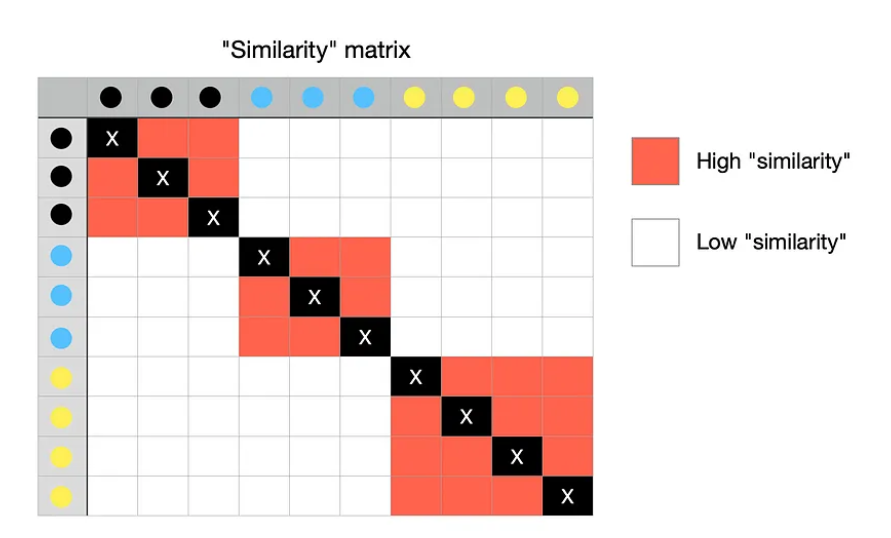
# First step: t-SNE starts by determining the “similarity” of points based on distances between them. Nearby points are considered “similar,” while distant ones are considered “dissimilar.”

It achieves this by measuring distances between the point of interest and other points and then placing them on a Normal curve. It does this for every point, applying some scaling to account for variations in the density of different regions.

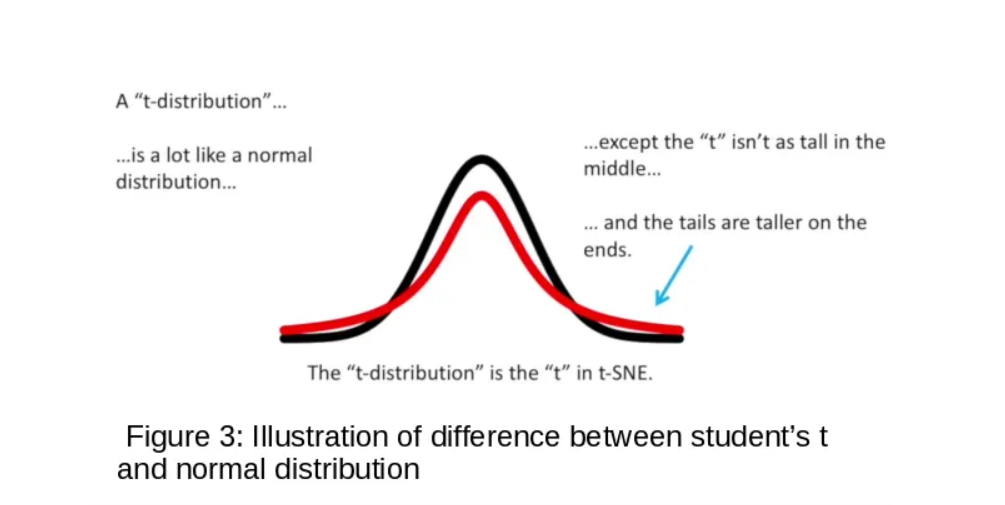
For example, the below illustration has a higher density in the region occupied by black points and a lower density in the region occupied by yellow and blue points:



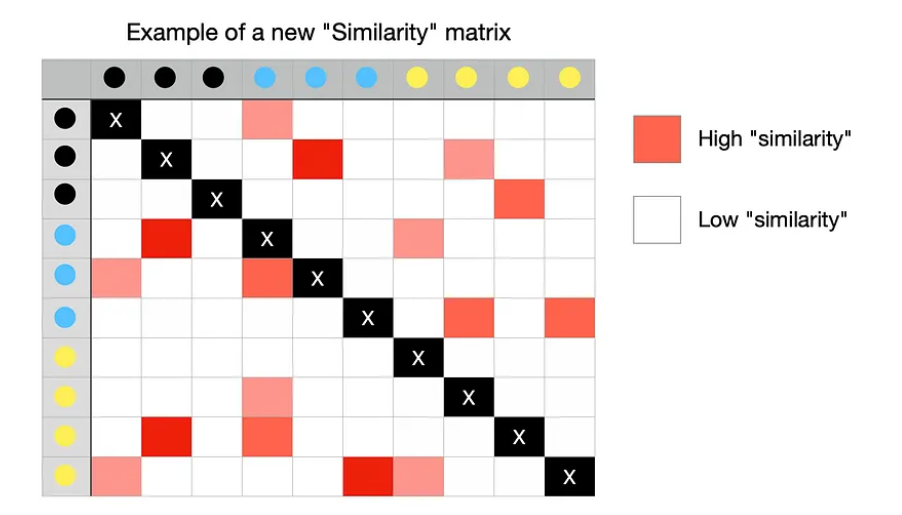
The result of these calculations is a matrix that contains similarity scores between each pair of points from the original multidimensional space.



**Step 2**  
Next, t-SNE randomly maps all the points onto a lower-dimensional space and calculates “similarities” between points as described in the process above. One difference, though, this time, the algorithm uses **t-distribution** instead of Normal distribution.



Unsurprisingly, though, the new “similarity” matrix differs significantly from the original one because of the random mapping. Here is an example of what it might look like.



**Step 3**Now the goal of an algorithm is to make the new “similarity” matrix look like the original one by using an iterative approach. With each iteration, points move towards their “closest neighbors” from the original higher-dimensional space and away from the distant ones.

The new “similarity” matrix gradually begins to look more like the original one. The process continues until the maximum number of iterations is reached or no further improvement can be made.

In more scientific terms, the above explanation describes the process of an algorithm trying to minimize the Kullback–Leibler divergence (KL divergence) through gradient descent.

## Perplexity

One important aspect that I haven’t mentioned yet is a hyper-parameter known as perplexity. It describes the expected density around each point or, in other words, relates to the target number of nearest neighbors from the point of interest.

Perplexity parameter plays a vital role in determining the final result of your embedding. Generally, you may want to choose perplexity to be somewhere between 5 and 50, but you should definitely experiment with different values.

A low value makes the algorithm “focus” on fewer neighbors, which results in many small groups. By contrast, high perplexity value “expands the neighborhood horizon,” resulting in fewer, more tightly packed groups.