Dimensionality reduction

By Waad ALMASRI

Introduction

The curse of dimensionality

What's this presentation about? Use this slide to introduce yourself and give a high level overview of the topic you're about to explain.

- High dimensional datasets are at risk of being very sparse: most training instances are likely to be far away from each other.
- This also means that a new instance will likely be far away from the training instances → making predictions much less reliable than in lower dimensions, since they will be based on much larger extrapolations.
- In other terms, the more dimensions the training set has, the greater the risk of overfitting it.
- In theory, one solution to the curse of dimensionality could be to increase the size of the training set to reach a sufficient density of training instances. Unfortunately, in practice, the number of training instances required to reach a given density grows exponentially with the number of dimensions. With just 100 features ranging from 0 to 1, you would need more training instances than atoms in the observable universe in order for training instances to be within 0.1 of each other, assuming they were spread out uniformly across all dimensions → $10^{100}\ versus\ 10^{78\ to\ 82}\ atoms$

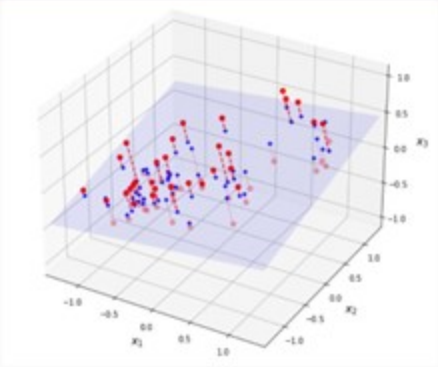
Dimensionality reduction approaches

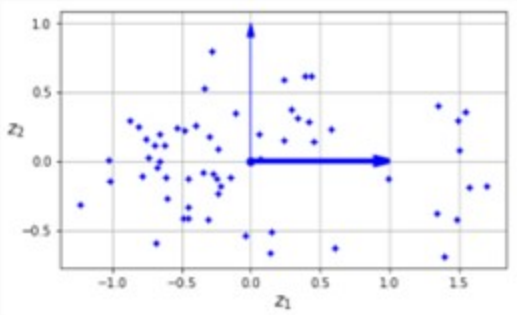
Dimensionality reduction approaches

- Projection:
- Manifold

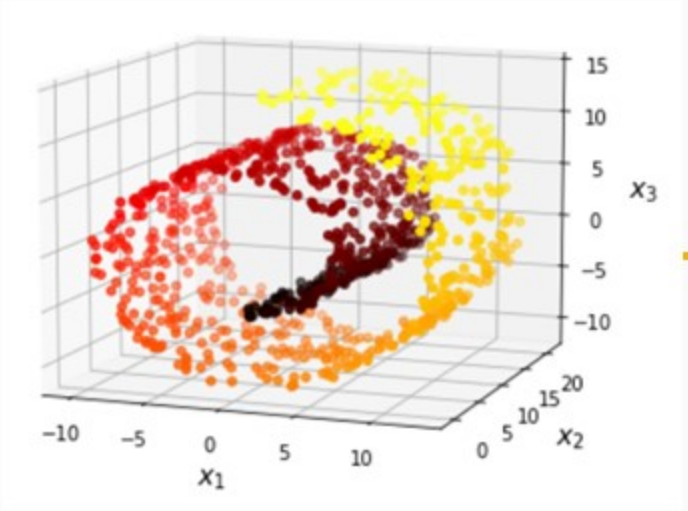
Projection

- In the real world, training instances usually lie in a much lower-dimensional space than they are for several reasons: features are constant, highly correlated, etc.,
- In the figure, on the right top corner, we can see that that all instances lie close to a plane; this is a lower-dimensional (2D) subspace of the high-dimensional (3D) space.
- After projecting the instances instance perpendicularly onto this subspace, we end up with the 2D dataset in the figure below.
- The new axes correspond to the new features z1 & z2.

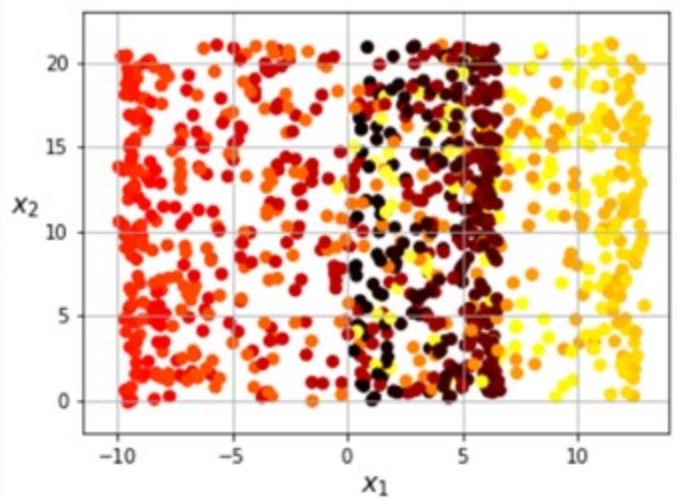




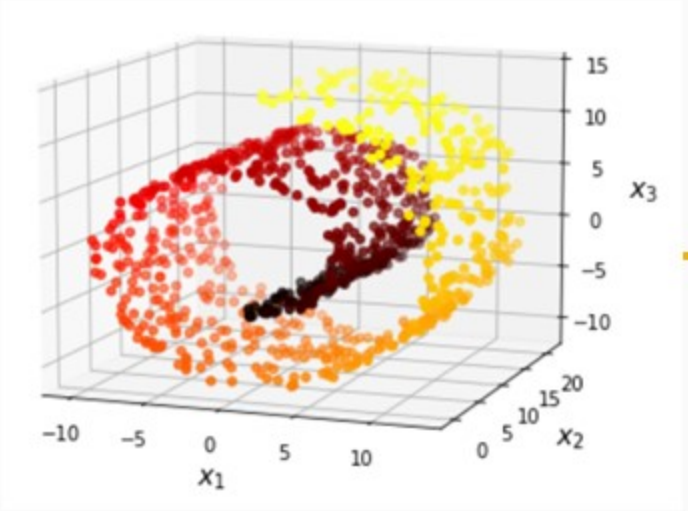
Projection (2)



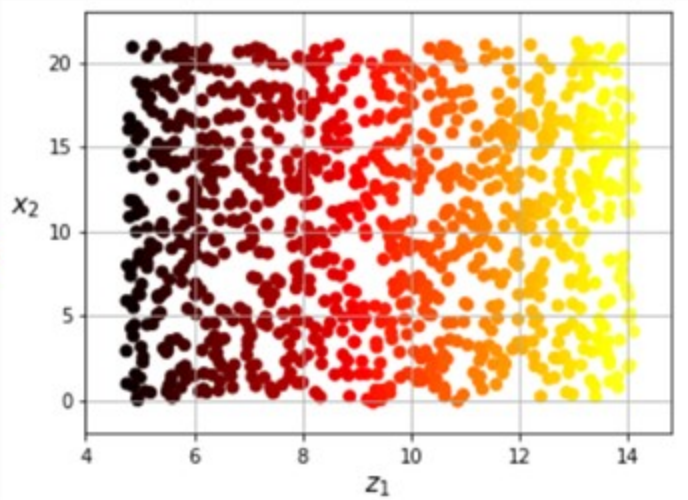
projection



Manifold – Swiss Roll is a 2D manifold



Unroll (Manifold)

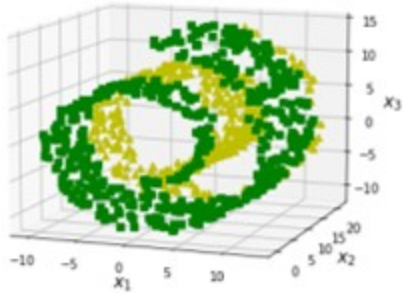


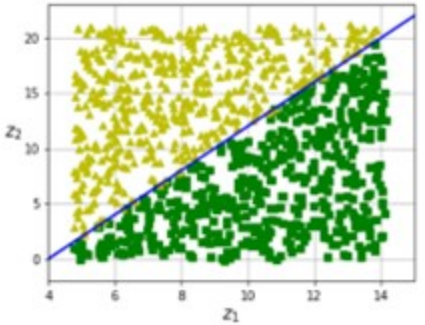
Manifold

- A d-dimensional manifold is a part of n-dimensional space (where d<n) that locally resembles a d-dimensional hyperplane.
- The Swiss roll locally resembles a 2D plane, but it is rolled in the third dimension → d=2 and n=3.
- Manifold Learning: It relies on the manifold assumption/hypothesis, which states that most real-world high-dimensional datasets lie close to a much lower-dimensional manifold.
- MNIST dataset

Manifold (2)

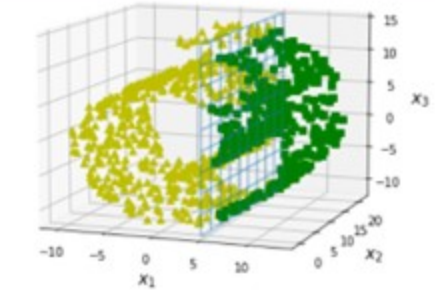
- With the manifold assumption, another implicit assumption is always there: the current problem (classification or regression) is simpler in the lower-lower-dimensional space of the manifold.
- In the figure on the right, we have a dataset with 2 classes.
- The decision boundary in the 3D space is complex.
- When unrolled in the 2D manifold space, the decision boundary is a simple straight line → much simpler





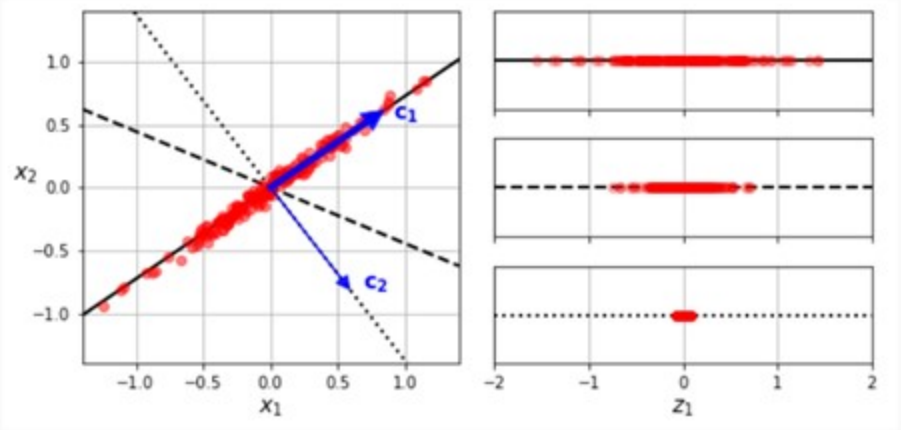
Manifold (3)

- This assumption of simplicity is not always true.
- In the right figure, the 3D decision boundary is a simple hyperplane of x1=5.
- However, when unrolled in a 2D manifold, the decision boundary becomes a collection of 4 independent line segments.
- → **Reducing the dimensionality of your training set before training a model will usually speed up the training, but it may not always lead to a better or simpler solutions, it all depends on the dataset.**



Principal Component Analysis (PCA)

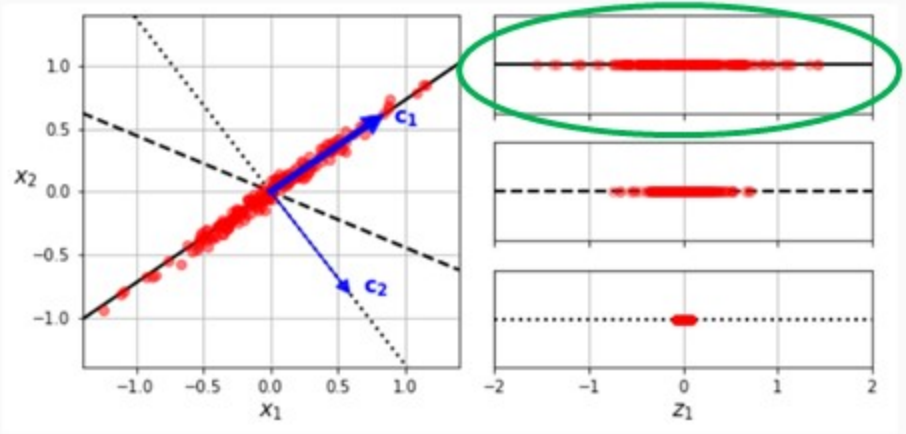
- It identifies the hyperplane that lies closest to the data, and then it projects the data onto it.
- It reduces the dimensionality of the data while preserving the maximum variance.
- The highest the variance, the lower the amount of lost information.
- The idea behind the PCA is: the new axes are the ones that minimize the Mean Squared Distance between the original dataset and its projection onto theses axes.



Which of the 3 axes on the right would be the best projection of the dataset on the left?

Principal Component Analysis (PCA)

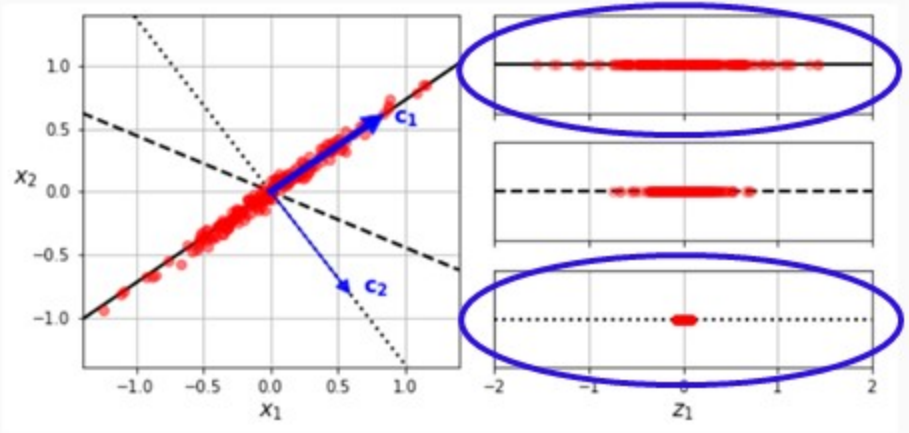
- It identifies the hyperplane that lies closest to the data, and then it projects the data onto it.
- It reduces the dimensionality of the data while preserving the maximum variance.
- The highest the variance, the lower the amount of lost information.
- The idea behind the PCA is: the new axes are the ones that minimize the Mean Squared Distance between the original dataset and its projection onto theses axes.



Which of the 3 axes on the right would be the best projection of the dataset on the left?

PCA (2)

- PCA identifies the axis that accounts for the largest amount of variance in the training set is the solid line.
- It also finds a second axis, orthogonal to the first one, that accounts for the largest amount of the remaining variance, i.e., the dotted line.
- → PCA finds as many axes as the number of dimensions in the dataset, all orthogonal to one another.
- The i^{th} axis is called the i^{th} Principal Component (PC) of the data.
- Here the 1st PC is the axis on which lies the vector c_1, the 2nd PC is the axis on which lies the vector c_2



PCA (3) – Principal Components (PC)

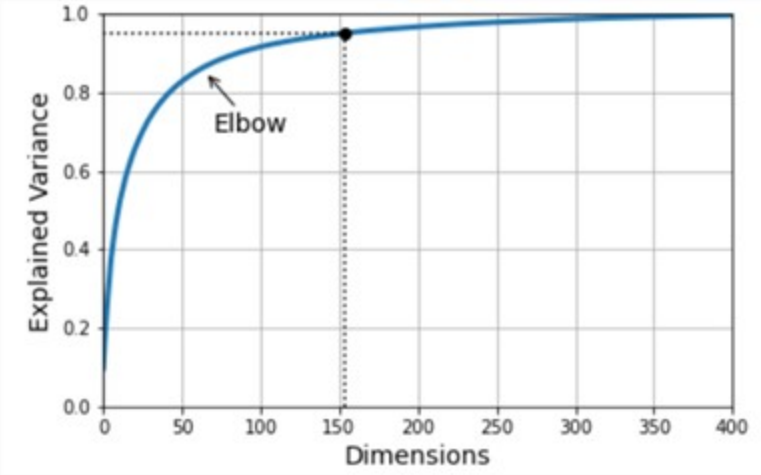
- To identify the PCs, the PCA algorithm uses the standard matrix factorization technique called ***Singular Value Decomposition*** (SVD).
- SVD decomposes a training matrix X into a matrix multiplication of 3 matrices $U \Sigma V^T$, where V contains the unit vectors defining all the principal components.
- $V = \begin{matrix} c11 & c21 & c31 \\ c21 & c22 & c32 \\ \dots & \dots & \dots \end{matrix} = (c1, c2, \dots, cn)$
- Once the PCs are identified, we reduce the dimensionality of the dataset down to d dimensions by projecting it onto the hyperplane defined by the first d principal components.
- Selecting this hyperplane ensures that the projection will preserve as much variance as possible.
- The reduced dataset $X_{d-proj} = XW_d$, W_d is the matrix containing the first d columns of V.

PCA (3) – Principal Components (PC)

- Scikit-Learn: **pca** = sklearn.ecomposition.PCA(n_components=…)
- After fitting the **pca** transformer to the dataset, the attribute *components_* is the transpose of W_d, *i.e.*, each row is a principal component, and it contains the d principal components with *d* = the parameter *n_components*.
- *pca.explained_variance_ratio_* = the proportion of the dataset's variance that lies along each principal component.
- Example: *pca.explained_variance_ratio_ of a 3D dataset= [0.76, 0.15]* means that
  - 76% of the dataset's variance lies along the first PC
  - 15% lies along the second PC
  - The remaining 9% lies along the third PC → the third PC carries little info.
  - NB: the dataset is 3D → 3 principal components maximum.

PCA (4) – Choosing the right number of Principal Components (PC)

- Scikit-Learn: **pca** = sklearn.ecomposition.PCA(n_components=…)
- Instead of arbitrarily choosing the number of reduced dimensions, we choose the number of dimensions that add up to a sufficiently large portion of the variance.
- i.e., **pca** = sklearn.ecomposition.PCA(n_components= *ratio*) with *ratio* being the *ratio* of the variance we wish to keep, *ratio* = 95%, 90%, etc.
- Another option is to plot the explained variance versus the number of dimensions.
- From the figure on the right, using 100 to 150 dimensions will not loose too much explained variance.



PCA (5) – data compression

- Dimensionality reduction plays a role of data compression technique
- Less dimensions → less space for a small percentage of variance loss (information loss).
- With PCA, it is possible to decompress the reduced dataset back to the original dimensions by applying the inverse transformation of the PCA projection: *pca.inverse_transform(X_reduced)*.
- This command is equivalent to: $X_{recovered} = X_{-proj} W_d^T$

PCA (6) – Randomized PCA

- In the Scikit-learn's PCA implementation, there is a hyperparameter called the *svd_solver*.
- When set to « randomized », Scikit-learn uses a stochastic algorithm called randomized PCA that quickly finds an approximation of the first d principal components.
- Its computational complexity is $\Theta(n \times d^2) + \Theta(d^3)$ instead of $\Theta(n \times m^2) + \Theta(m^3)$ for the full SVD approach, where n is the total number of training instances and m is total number of features.
- If the maximum between n & m >500 and $n\_components\_ < 80\% \, of \, the \, \min(n,m)$.

PCA - ! ! !

- For each principal component, PCA finds a zero-centered unit vector pointing in the direction of the PC. Since 2 opposing unit vectors lie on the same axis, the direction of the unit vectors returned by PCA is not stable: if you perturb the training set slightly and run PCA again, the unit vectors may point in the opposite direction as the original vectors. However, they will generally still lie on the same axes. In some cases, a pair of unit vectors may even rotate or swap (if the variances along these 2 axes are very close), but the plane they define will generally remain the same.
- PCA assumes the dataset is centered around the origin; the PCA of Scikit-Learn implements data centering explicitly for us.
- When used as a preprocessing step in a learning task (classification or regression), PCA's number of dimensions (n_components) is a hyperparameter to be tuned like any other Machine Learning hyperparameter to optimize the training.
- When PCA is used for data visualization goals, the n_components is 2 or 3; since we can only see and understand 2 to 3D data.

PCA (7) – Incremental PCA

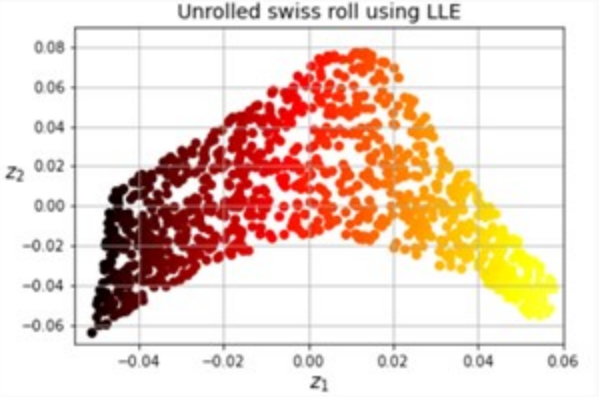
- Previous PCA implementations require the whole training set to fit in memory.
- This could be problematic when the training set is too large and cannot fit in memory.
- Solution = Incremental PCA (IPCA); it splits the training set into mini-batches and fit the PCA with one mini-batch at a time. This is called PCA online (on the fly, as new instances arrive).
- Scikit-Learn's IPCA = IncrementalPCA(), when used with min-batches, the function partial_fit() is called not the traditional fit() method.

Random Projection

- It projects the data to a lower-dimensional space using a random linear projection.
- With high probability, a random projection preserves distances fairly: 2 similar instances will remain similar after the projection and 2 different instances will remain different. This was demonstrated by the Willima B. Johnson & Joram Lindenstrauss' s famous lemma.
- The minimum number of dimensions to preserve to ensure the distances won't change by more than a given tolerance ε is $d \geq \frac{4\log(n)}{\frac{1}{2}\varepsilon^2 - \frac{1}{3}\varepsilon^3}$, n= the total number of training instances
- With random projection, no training is required, the algorithms only needs to create the random matrix (its size is equal to the training data's size); the data itself is not used.
- The algorithms proceeds as follows:
  - Determines the target dimensionality using the equation is $d \geq \frac{4\log(n)}{\frac{1}{2}\varepsilon^2 - \frac{1}{3}\varepsilon^3}$
  - Create a random matrix P of shape $[d, m]$, where each item is sampled randomly from a Gaussian distribution with mean 0 and variance $\frac{1}{d}$
  - Use P to project the dataset from n dimensions down to d: $X\_reduced = X.P^T$
- Scikit-Learn: GaussianRandomProjection & SparseRandomProjection

Locally Linear embedding (LLE)

- A nonlinear dimensionality reduction technique based on the manifold learning.
- The algorithm proceeds as follows:
  - It measures how each training instance linearly relates to its nearest neighbors
  - It looks for a low-dimensional representation of the training set where these local relationships are best preserved
  - It then projects the instances to the lower dimensional space
- With LLE, the distances between instances are locally preserved but distances on a large scale are not.



The unrolled Swiss roll should be a rectangle not a stretched and twisted band. This is an example of LLE preserving distances between instances locally but not on a large scale.

LLE – The algorithm (2)

- For each training instance x_i, the algorithm identifies its k-nearest neighbors, then reconstructs x_i as a linear function of these neighbors.
- It finds the weights $w_{i,j}$, such that the squared distance between x_i and $\sum_{j \in neighbors\ of\ i} w_{i,j} x_j$ is as small as possible; the constrained optimization problem: $\widehat{W} = \underset{W}{\operatorname{argmin}} \sum_{i=1}^{n} (x_i - \sum_{j=1}^{n} w_{i,j} x_j)^2$, $\widehat{W}$ holds the local linear relationships between the training instances.
  - Subject to:
    - $w_{i,j} = 0\ if\ x_j\ is\ not\ one\ of\ the\ k\ nearest\ neighbors\ of\ x_i$
    - and $\sum_{j=1}^{n} w_{i,j} = 1\ \ for\ i = 1, 2, \ldots, n$
- After computing $\widehat{W}$, LLE maps the x_i into a d-dimensional space ($d < n$) while preserving these local relationships → with z_i being the image of x_i in the d-dimensional space, we want the squared distance between z_i and $\sum_{j=1}^{n} \hat{w}_{i,j} z_j$ to be as small as possible; the unconstrained optimization problem:
  - $\hat{Z} = \underset{z}{\operatorname{argmin}} \sum_{i=1}^{n} (z_i - \sum_{j=1}^{n} \hat{w}_{i,j} z_j)^2$
  - Here we are finding the optimal positions of the training instances' images.
- Scikit-Learn: LocallyLinearEmbedding(), its computational complexity = $\Theta(n.log(n).m.\log(k))$ for finding the $\mathrm{k} - \mathrm{nearest\ neighbor} + \Theta(m.n.k^3)\ for\ optimizing\ the\ weights + \Theta(dn^2)\ for\ constructing\ the\ low - dimensional\ representations$
  - → the quadratic complexity $\Theta(dn^2)$ with respect to the number of training instances makes LLE scale poorly to large

References

- https://www.jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf?fbcl

t-distributed Stochastic Neighbor Embedding t-SNE

- It is a non-linear technique used for visualizing high-dimensional data like embeddings by reducing them to two or three-dimensional data, all while maintaining the same data distributions in original high-dimensional input space and low-dimensional output space.
- In simpler terms, t-Distributed stochastic neighbor embedding (t-SNE) minimizes the divergence between two distributions: a distribution that measures pairwise similarities of the input objects and a distribution that measures pairwise similarities of the corresponding low-dimensional points in the embedding.