Implemeting t-SNE with numpy

April 27, 2021

Authors: Abdullah AlOthman, Ethan Swartzentruber

Date: 4/27/2021

Original Paper: Visualizing Data using t-SNE

Github Link: https://github.com/epswartz/t-sne

Python Package Installation: `pip install simple_tsne` (see github readme for usage example)

1 Team Member Contributions

Abdullah AlOthman: PyPI packaging, plots for paper, equal contribution to implementation

Ethan Swartzentruber: Most of paper writing, equal contribution to implementation

2 Abstract

t-SNE, meaning t-distribution symmetric neighbor embedding, is a dimensionality reduction technique which is mainly focused on making high dimensional data possible to visualize by reducing it to a lower number of dimensions, typically 2 or 3. We present an explanation of the t-SNE algorithm, a comparison of the algorithm with other methods of dimensionality reduction, and an implementation of the algorithm in python which heavily leverages the numpy library for numerical computation. In general, we found the algorithm to perform at least as well as other dimensionality reduction techniques for the purposes of visualization. However, it has a longer runtime than simpler techniques, and should not be used for extremely large data as the runtime would be prohibitive.

3 Background

A recurring problem in machine learning and statistics is the visualization of high dimensional data - it is extremely important to researcher's intuitive understanding that the data be visualized, but straightforward ways of plotting high dimensional data (plotting only a subset of the dimensions, etc) often perform very poorly. Dimensionality reduction is a common way to remedy this problem. Ideally, these techniques are able to represent the data with a number of dimensions small enough to plot without losing too much of the structure of the data.

Basic dimensionality reduction techniques, such as PCA, are often enough to see some patterns in the data. However, these techniques can be insufficient, or even outright misleading, when applied to datasets where the dimensions have complex, non-linear relationships. The t-SNE algorithm is a more general dimensionality reduction technique, and does not make as many assumptions about the structure of the data. In essence, t-SNE is based on the assumption that data points that are close to each other in high dimensional space should be similarly close in low dimensional space, compared to other points.

It is important to understand a key limitation of the algorithm: t-SNE cannot be applied on new data the way that one can apply PCA, nor can it be reversed. t-SNE is an iterative algorithm, fit by moving around the low dimensional data points until they "match" the high dimensional ones in the particular mathematical sense that t-SNE cares about - it does not yield any notion of what the reduced components are, and as such, cannot be applied on new data that wasn't a part of the initial iterative process.

In general, t-SNE does well when the goal is to visualize data for the researcher's own intuitive understanding of the structure, particularly natural strata of the data. For example, we will show later in this paper that t-SNE can effectively reveal the clustered nature of the MNIST dataset. However, t-SNE is not particularly well-suited for dimensionality reduction preceding the fitting of a model. This is because it cannot be applied on new data that the model predicts. Additionally, because it does not yield a notion of what the components are, t-SNE is not as well suited for feature engineering as other dimensionality reduction techniques.

4 Algorithm Description

t-SNE is based on the intuition that if data points are relatively close together in high dimensional space, they should also be relatively close in low dimensional space. In particular, t-SNE uses a notion of "affinity" between data points, which can be interpreted as a likelihood that two points are neighbors. The closer the points are, the higher this affinity, increasing in a non-linear way. The t-SNE algorithm takes place in two phases:

1. Compute the pairwise affinities of the high dimensional points
2. After randomly initializing them, move the low dimensional points around until the low dimensional points affinities are similar to the high dimensional ones. Essentially, points that are too close compared to their distance in high dimensions move away from each other, and points that are too far away move closer.

To gain some basic intuition on the process, we have created an animation of the t-SNE fitting process that shows the low dimensional points being moved around: https://github.com/epswartz/t-SNE/blob/main/animations/t-sne_mnist_500iter_full.gif

The full version of the t-SNE algorithm follows. For all of these computations: * X is the dataset, with x_i being some point i * Y is the dataset's low dimensional embedding, with y_i being some embedded point i corresponding to x_i

4.0.1 1: Binary search and high dimensional affinities

First, we must compute pairwise affinities in high dimensional space. The user gives a value called "perplexity", which is a measure of how many points each point should consider to be its neighbors - the higher the perplexity, the slower the drop off in affinity based on distance between points. Note that this affinity is asymmetric, as a particular point may choose another point as its neighbor, but that point may have much closer points. The perplexity is defined as:

$$Perp(P_i) = 2^{H(P_i)}$$

where P_i is a probability distribution of how likely point i is to pick any other point as its neighbor, and $H(P_i)$ is the Shannon entropy of P_i, defined as:

$$H(P_i) = -\sum_j p_{j|i} log_2 p_{j|i}$$

The distribution $p_{j|i}$ is based on the pairwise distances between points i and j, and is calculated as follows:

$$p_{j|i} = \frac{exp(-\|x_i - x_j\|^2/2\sigma_i^2)}{\sum_{i \neq k} exp(-\|x_i - x_k\|^2/2\sigma_i^2)}$$

Where x_i, x_j and x_k are data points, and σ_i is the standard deviation of a probability distribution centered at x, which defines its affinity to the other data points. At the beginning of the algorithm, the σ_i values are unknown - the first step of t-SNE is to perform a binary search across the space of all possible σ_i values, choosing a sigma for each point that gives its affinity distribution the user's desired perplexity.

4.0.2 2: Iterative fitting

t-SNE is an iterative algorithm, fit via gradient descent. The core idea of t-SNE is that the affinity distribution should be similar in high and low dimensional space, and as such, the cost function is defined as the sum of KL-Divergences across the low and high affinity distributions of every point:

$$C = \sum_i KL(P_i||Q_i) = \sum_i \sum_j p_{j|i} log \frac{p_{j|i}}{q_{j|i}}$$

Here, P_i is the distribution of affinities in high dimensional space for point i, and Q_i is the distribution of lower dimensional affinities by the same point i. Q is defined similarly to P, however, it is symmetric:

$$q_{ij} = \frac{exp(-\|y_i - y_j\|^2)}{\sum_{k \neq l} exp(-\|y_k - y_l\|^2)}$$

We found this notation slightly unclear - note that k and l in the denominator are simply two indices - the sum is across all pairwise square norm differences where the two points are not the same.

After computing q_{ij}, we can compute the gradient of the cost function and fit the algorithm:

$$\frac{\delta C}{\delta y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)$$

Note that this uses p_{ij}, a symmetric version of the high dimensional pairwise affinity distribution that we computed in the preceding section. These values are defined as $p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n}$, where n is the number of data points.

After this calculation, we are ready to update Y as follows:

$$Y^{(t)} = Y^{(t-1)} + \eta \frac{\delta C}{\delta Y} + \alpha^{(t)}(Y^{(t-1)} - Y^{(t-2)})$$

Where η and α are the learning rate and momentum coefficient, respectively.

We run this iterative loop for some set number of iterations, or until the cost function is minimized.

5 Performance Optimization

5.0.1 Vectorization

Our primary method of optimization was to create vectorized versions of all the subcomponents of t-SNE. This version vastly outperformed our original version, due to the massive parallelization of vector operations. We were able to achieve over 100x speedup through vectorization, shown in the plots below.

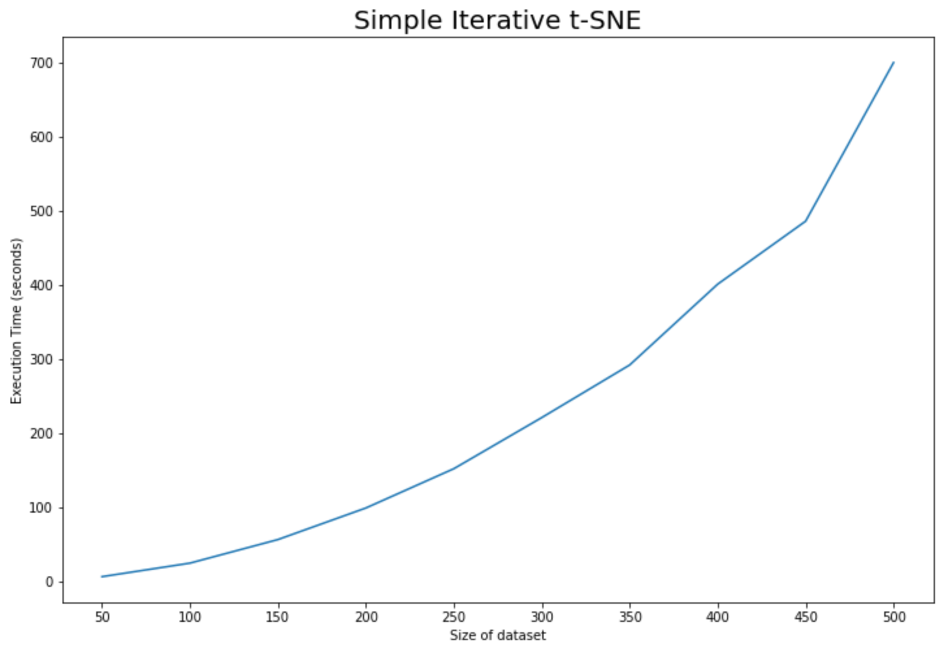
5.0.2 JIT compilation with `numba`

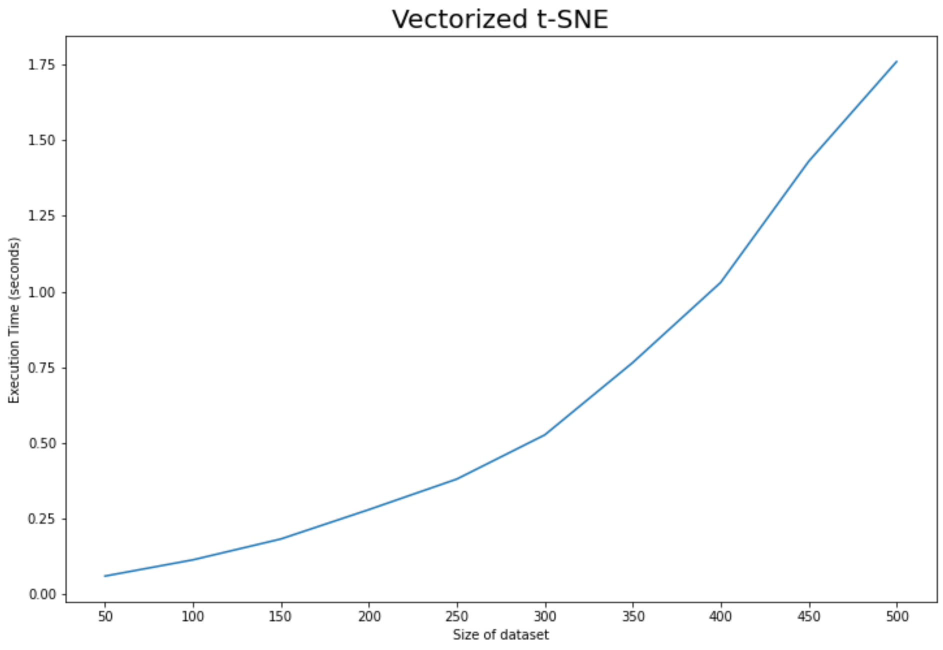
In addition to vectorization, we applied just-in-time compilation with numba in an attempt to speed things up. However, we saw no increase in performance.

5.0.3 Reducing iterations needed

We apply a technique called *early exaggeration*, which is simply a multiplication of the high dimensional pairwise affinities in the early stages of the iterative portion of the algorithm. This makes the randomly generated q_{ij} values too small to model their corresponding p_{ij} values, and thus makes the data spread out to try to match. This leads to better separated clusters, and because users will run the algorithm for as long as it takes to get separated points, this is essentially a performance improvement.

5.0.4 Performance vs. Size of Dataset





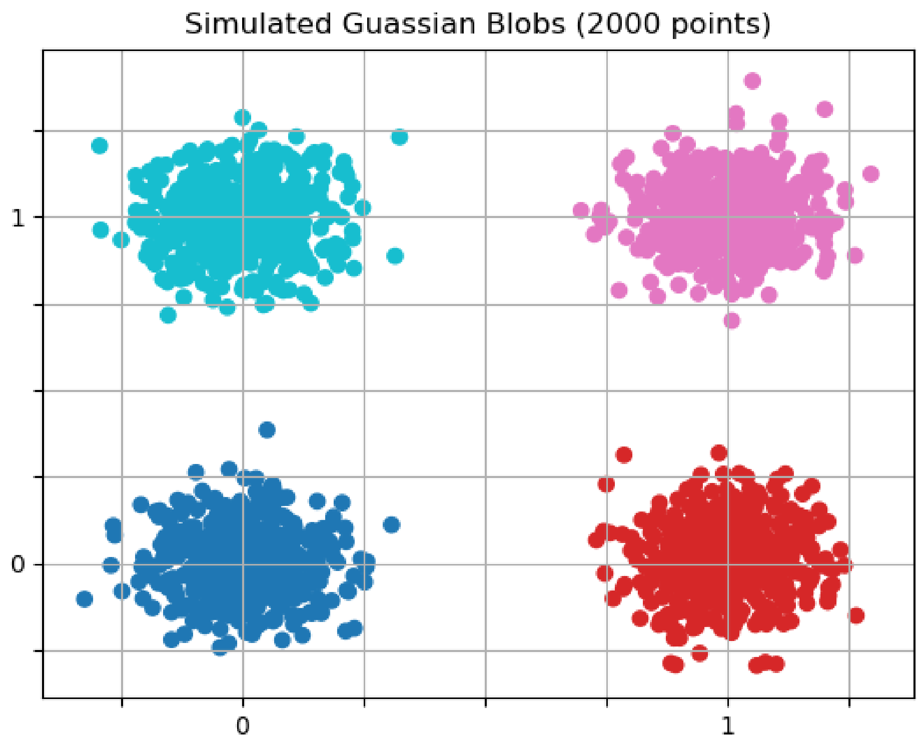
6 Performance on Data and Comparative Analysis

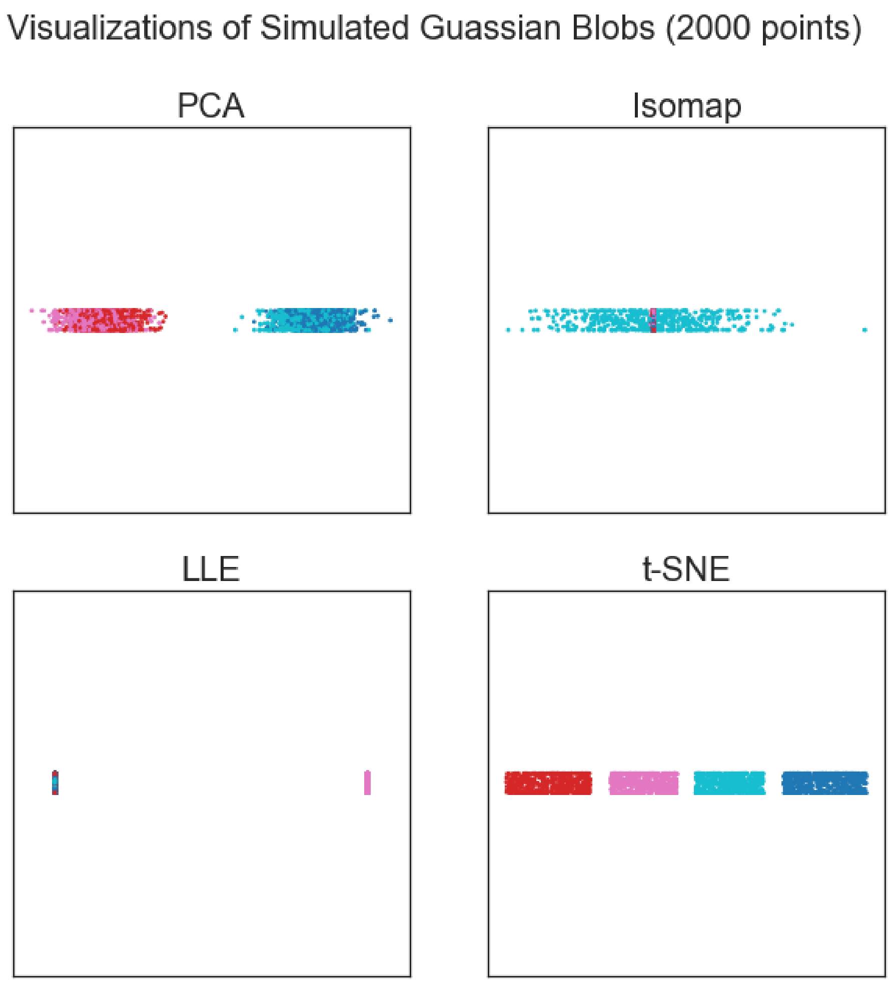
t-SNE is used to visually investigate the structure of high dimensional datasets, and in particular, to spot clusters and strata of the data. We present two comparisons.

Note that since the primary purpose of t-SNE is visualization, these are not rigorous numerical comparisons. However, the results from t-SNE definitely seem sensical on both datasets.

6.1 Simulated Data: `sklearn.datasets.make_blobs`

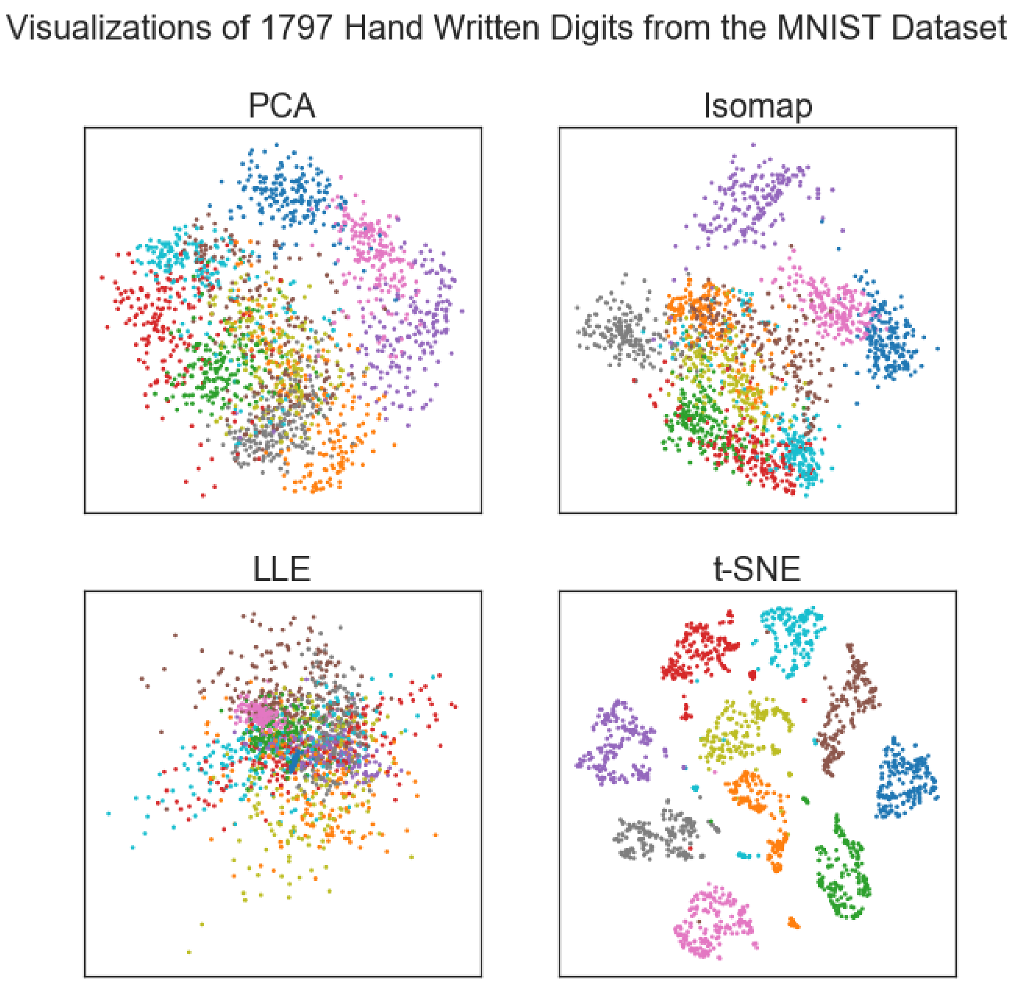
The first comparison is on a simulated dataset made of 4 Gaussian blobs, which are centered at the points $(0, 0)$, $(1, 0)$, $(0, 1)$, and $(1, 1)$. On this task, t-SNE is able to separate the blobs in 1 dimension, even though a single axis cannot separate them. Compare this to PCA, which, as it is linear, cannot separate the blobs. In this test, t-SNE is embedding 2-dimensional data into 1-dimension (some jitter is added in the vertical dimension to help see the embedded data).





6.2 Real Data: 8x8 MNIST

The second comparison is on the MNIST handwritten digits dataset. On this data, t-SNE is able to separate the clusters representing numbers, and even suggests some clusters that aren't immediately obvious. For example, the ones are almost always split into two groups - we suspect this is because some people draw ones with a "hat" on top, and some just draw them as a single vertical line.



7 Discussion/Conclusion

In general, we think t-SNE is a great tool for visualization - it performs well on many datasets that are difficult for other techniques to separate meaningfully. However, since there isn't an interpretable notion of what the dimensions are, there isn't always a useful way to interpret t-SNE's insight other than "it looks like there are X clusters". One idea to expand on t-SNE would be to fit a regression from the high dimensional dataset to t-SNE's embedded dimensions - this may be able to reveal that some of the discovered dimensions have useful interpretations, depending on the data.

Another way to expand on t-SNE would be to use more different types of initialization. Since we implemented the algorithm's original version from the paper, we simply drew the initial low dimensional representation from a multivariate normal distribution. In other implementations, people often initialize the representation using PCA. There's no reason that you couldn't use any other technique, and it would be interesting to see which perform better.

8 References

L. van der Maaten and G. Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

L. van der Maaten, t-SNE implementation at https://lvdmaaten.github.io/tsne/.

Liam Schonoveld, *In Raw Numpy: t-SNE*, https://nlml.github.io/in-raw-numpy/in-raw-numpy-t-sne/.

Alexander Fabisch, Christoper Moody, Nick Travers, *scikit-learn t-SNE implementation*, https://github.com/scikit-learn/scikit-learn/blob/main/sklearn/manifold/_t_sne.py.