Non-linear dimensionality reduction methods: Uniform Manifold Approximation and Projection (UMAP) versus t-Distributed Stochastic Neighbor Embedding (t-SNE)

Ivan Dewerpe

Supervisor: Korbinian Strimmer

14th of May 2021

Abstract

Ever since the introduction of UMAP in 2018, two camps have emerged in the space of non-linear dimensionality reduction algorithms: one that advocates for UMAP's use and the other that defends t-SNE. I begin by describing the range of linear and non-linear dimensionality reduction algorithms: focusing on t-SNE and UMAP and their respective optimisations. I continue by describing state-of-the-art t-SNE and UMAP pipelines and evaluate their embedding's local & global structure preservation, runtime performance, and visualisation on four datasets: a simulated RNA-Seq dataset, a circle dataset, MNIST Digits and MNIST Fashion. I find the most striking difference between t-SNE and UMAP being; UMAP constructs denser clusters with additional whitespace, however, there is little evidence to favour one technique over the other. I conclude by debunking UMAP's superiority over t-SNE and recommend continuing the use of both algorithms to explore which provides a more faithful representation of high-dimensional data.

Acknowledgments

I want to use this section to thank my supervisor Professor Korbinian Strimmer. I would not have been introduced to this topic area nor been able to explore it so thoroughly without his guidance.

Contents

List of Figures

Chapter 1

Introduction

The interpretation of high-dimensional data exists as a substantial hurdle in a wide range of applications such as genomics, cancer research, and bioinformatics. Dimension reduction techniques allow the visualisation of high-dimensional data in two to three dimensions, thus aiding our comprehension of interactions within the data and providing better accuracy for classification models. A common approach to project high-dimensional data onto the Euclidean space is to use PCA: an orthogonal linear transformation. Linear dimensionality reduction approaches (such as PCA) do not faithfully preserve the local interactions between points in the datasets. On the other hand, Non-Linear Dimensionality Reduction (NLDR) methods can maintain the local and global structure of high-dimensional data by extracting significant interactions within its local neighbourhoods (Birjandtalab et al. (2016)).

t-stochastic neighbour embedding (t-SNE), created by Laurens Van Der Maaten and Geoffrey Hinton in November 2008, "minimises the difference between high-dimension and low dimension join distributions" (Birjandtalab et al. (2016)). It rapidly became one of the most widely used NLDR techniques and a benchmark technique for visualising RNA-Seq Data and other high-dimensional biological data. t-SNE was unrivalled until 2018 when McInnes, Healy and Melville (2018) introduced Uniform Manifold Approximation Projection (UMAP). In the original paper, the authors declared that "the UMAP algorithm is competitive with t-SNE for visualisation quality, and arguably preserves more of the global structure with superior run time performance." Becht et al. in 2019 then demonstrated that UMAP could produce more faithful low-dimensional embeddings than t-SNE on single-cell data. However, Kobak and Berens (2019), and Kobak and Linderman (2021) attributed the findings to the wrong parametrisation and initialisation of the t-SNE algorithm, arguing that t-SNE is at least equivalent to UMAP. By comparing the two algorithms, UMAP and t-SNE, they found that the most significant difference occurring is that UMAP produced denser cluster structures with additional white spaces. It is important to note that both Kobak and McInnes have a clear motivation to defend t-SNE and UMAP: the former is the author of OpenTSNE (Python package) and an avid defender of the algorithm, and the latter of UMAP and its Python package.

In the current landscape, it is unclear whether one should use t-SNE or UMAP. To address this uncertainty, I provide in this report an analysis of the four contentious differences: the local structure preservation, global structure preservation, runtime efficiency and meaningfulness of the visualisation. To explain the theoretical differences, I will first describe the methods: PCA, t-SNE and its improvements, UMAP, and finally HDBSCAN for my clustering analysis. To visualise and compare embeddings resulting from t-SNE and UMAP, I present the results

of my experiments on different datasets: first on simulated data to mimic RNA-Seq data, then on simulated data of a circle, and finally on the MNIST Digits and Fashion datasets. t-SNE and UMAP are respectively initialised and parametrised following the latest state-of-the-art pipelines in single-cell analysis literature to offer a fair comparison. I then present a discussion of my findings with a more critical lens to identify the most meaningful differences between UMAP and t-SNE.

In this report, I discovered that McInnes, Healy and Melville (2018)'s original claims favouring UMAP are no longer correct: t-SNE and UMAP's visualisation capabilities and runtime performance are equivalent. Additionally, with careful hyper-parametrisation and initialisation of t-SNE, I obtained higher quantitative scores on all my datasets for faithfully preserving the global and local structure of high-dimensional data than on UMAP. Finally, I corroborate Kobak and Berens (2019)'s claim that the most striking difference between the two techniques is that UMAP creates denser clusters with additional whitespace to separate unrelated points. I also find that using the UMAP structure as a basis in clustering analysis produces higher accuracy than t-SNE without exaggeration.

Chapter 2

Methods

In this chapter, we will describe in detail two principal methods in the family of non-linear dimensional reduction algorithms: Uniform Manifold Approximation and Projection (UMAP) and t-Distributed Stochastic Neighbor Embedding (t-SNE)

2.1 Nonlinear Dimensionality Reduction

High-dimensional data is difficult to interpret and computationally expensive to process. By reducing high dimensional data to lower dimensions, we enable humans to visualise high-dimensional in two or three dimensions. The technique of nonlinear dimensionality reduction can be formulated as follows. Let's assume some data X consists of n data points $x_i, 1 \leq i \leq n$ of dimension p. The intrinsic dimensionality of X, given by d, where $d << N$, can be thought of as "points in the dataset $[X]$ ... lying on or near a manifold with dimensionality d that is embedded in the N dimensional space"(van der Maaten et al. (2007)). In essence, dimensionality reduction techniques transform dataset X into a new dataset Y with dimensionality d, whilst retaining important as much as possible geometric properties of the data. The notion of intrinsic dimensionality is crucial to understand the limitations of dimensionality reduction techniques. We are certain to lose some of the geometric properties of a data sample if we reduce it to a much lower dimension than its intrinsic dimensionality.

2.1.1 PCA

Principal Component Analysis (PCA), is a benchmark linear dimensionality reduction technique invented by Karl Pearson in 1901. In the upcoming chapters, PCA will be used as an initialisation condition, a key characteristic in our comparison of t-SNE and UMAP. We will use the same data matrix X expressed above. The aim of PCA is to maximally compress information from X into latent variables $y \in \{y_1, y_2, ...\}$. These latent variables are called "principal components." PCA is an orthogonal transformation of data matrix X so that the resulting components are orthogonal:

$$T = XU$$

Where T are our principal components, and U is an orthogonal matrix. To understand the compression properties of PCA, we note that for principal component $\mathbf{t}^{PCA} \in T$, $Var(\mathbf{t}^{PCA}) = \Lambda$, where Λ is a diagonal matrix with $diag(\Lambda) = \{\lambda_1, \lambda_2, ..., \lambda_p\}$. Hence the variance of principal component t_i is equal to eigenvalue λ_i. The total variation is $\mathbf{Tr}(Var(\mathbf{t}^{PCA}))$

which is equal to $\mathbf{Tr}(\Lambda) = \sum_{i=1}^{p} \lambda_i$. To understand how much variation each principal component encapsulates, we use $\frac{\lambda_i}{\sum_{i=1}^{p} \lambda_i}$. Thus, components with low variation may be discarded, leading to a reduction in dimension.

2.2 t-SNE

The algorithm t-stochastic neighbor embedding (van der Maaten and Hinton (2008)) is a non-linear dimensional reduction method created by Laurens Van Der Maaten and Geoffrey Hinton and used for the visualisation of high dimensional data. It works by giving each datapoint a location in a two or three dimensional map. The technique is derived from Stochastic Neighbor Embedding (SNE) (Hinton and Roweis (2003)). However, SNE is hampered by two problems that t-SNE solves: a cost function that is difficult to optimise and by what Van Der Maaten refers to as the "crowding problem."

Given a set of N high-dimensional objects $x_1, ..., x_N$, t-SNE starts by converting the high-dimensional Euclidean distances between datapoints into conditional probabilities p_{ij} that represent similarities. We can describe the similarity between data point x_j and x_i as the conditional probability, $p_{j|i}$. In other words, $p_{j|i}$ is the probability that "x_i would pick x_j as its neighbour if neighbours were picked in proportion to their probability density under a Gaussian centered at x_i."(van der Maaten and Hinton (2008)) $p_{i|j}$ is given by:

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

Because we are only interested in pairwise similarity, note that $p_{i|i} = 0$. We will explain how to set the bandwith of the Gaussian kernels σ_i at the end of the section. The first problem with SNE arises at this stage. Namely that for a high-dimensional outlier datapoint x_i, the values of p_{ij} are extremely small for all j. For this x_i, "the location of its low-dimensional map point y_i has very little effect on the cost function."(van der Maaten and Hinton (2008)) This results in the inability to determine the location of the current map point from the positions of the other map points. To remedy this, Van Der Maaten et al. defined symetrised conditional probabilies p_{ij} in high-dimensional space by:

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N}$$

where $p_{ij} = p_{ji}$, and $p_{ii} = 0$. This was shown to create a simpler form of SNE's gradient and hence improve its accuracy and efficiency.

As I explained in 2.1, data has an intrinsic dimensionality that reflects the low-dimensional counterpart of x_i and x_j namely y_i and y_j with ($y_i \in \mathbb{R}^d$, and $d << N$). To understand the emergence of t-SNE over SNE, I now discuss the "crowding problem." Assume that we have a manifold with 8 intrinsic dimensions which is itself embedded into a space of much higher dimensionality, let's say 28. The Gaussian kernel used to calculate the p_{ij} uses Euclidean distance, which is inherently affected by the curse of dimensionality. The curse of dimensionality can be defined as a problem in high dimensional data when distance measures are inadequate in discriminating. Going back to our example, it is possible that in eight dimensions, we have nine points that are mutually equidistant. Yet it is impossible to faithfully model this relationship using a two-dimensional map. Van Der Maaten et al. found that the symmetric property of symmetric SNE "matched the joint probabilities of pairs of datapoints in the high dimensional and low-dimensional spaces *rather* than their

distances." This allowed for a natural way to eradicate the "crowding problem." Instead of using a Gaussian distribution in both the high and low dimensional space, in t-SNE we employ a Student t-distribution with one degree of freedom (Cauchy Distribution). The heavier tails in this distribution as opposed to the Gaussian one, allows high-dimensional space to be modelled faithfully: dissimilar points can be modelled further apart in the map. t-SNE defines q_{ij} for $i \neq j$ and $q_{ii} = 0$ as:

$$q_{ij} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_k \sum_{l \neq k} (1 + \|y_k - y_l\|^2)^{-1}}$$

The goal for t-SNE is to minimise the Kullback-Leibler divergence (Kullback and Leibler (1951)) of the distribution Q and P to make them as similar as possible. This is given by:

$$C = KL(P||Q) = \sum_i \sum_j p_{ij} log \frac{p_{ij}}{q_{ji}}$$

The gradient of the Kullback-Leibler divergence between Q and P is given by:

$$\frac{\partial C}{\partial y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + \|y_i - y_j\|^2)^{-1}$$

Intuitively, we think of this optimisation problem as finding the most faithful low dimensional embedding to the high dimensional data. We can now talk about the bandwidths σ_i for the Gaussian Kernels. It is difficult to pick an "optimal" σ_i since the density of the data is likely to vary for all the data points (Poličar (2020)). To understand how this is done in t-SNE, we have to introduce the notion of perplexity. Perplexity is a smooth measure of the k number of close neighbours (van der Maaten and Hinton (2008)). The value usually ranges between 5 and 50 according to the authors. It can be mathematically defined as follows:

$$\text{Perplexity}(P_i) = 2^{H(P_i)}$$

where $H(P_i)$ is the Shannon entropy of P_i measured in bits

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

The above is the initial version of t-SNE. A direct implementation is quite slow. For example, computing p_{ij} adn q_{ij} requires computing all the pair-wise interactions between points and has a time complexity $O(n^2)$.(Poličar (2020)) In the subsequent decade, most papers addressing improvements within t-SNE, provided with optimisations in its implementation. More specifically, optimisations in the computation of p_{ij} and q_{ij}. To talk about these optimisations, it is helpful to rewrite the gradient update rule in the following form(van der Maaten and Hinton (2008)):

$$\frac{\partial C}{\partial y_i} = 4 * \left(\sum_{j \neq i} p_{ij} q_{ij} Z(y_i - y_j) - \sum_{j \neq i} q_{ij}^2 Z(y_i - y_j) \right)$$

or

$$\frac{1}{4} * \frac{\partial C}{\partial y_i} = \mathcal{F}_{attr,i} - \mathcal{F}_{rep,i}$$

Where we define Z to be the normalisation constant present in q_{ij}, namely:

$$Z = \sum_{k \neq l} (1 + \|y_k - y_l\|^2)^{-1}$$

We have split the gradient to improve our intuition of the "repulsive" and "attractive" forces between points in the embedding. Namely, the first term $\mathcal{F}_{attr,i}$, promotes local neighbours to remain close to one another, hence preserving the local structure of the high-dimensional data. The second term represents the repulsive forces: t-SNE pushes unrelated points further from one another. When using t-SNE, one often has to try different parameters to produce the optimal embedding (see Section 3.1). The optimisations that follow in computing the attractive and repulsive forces are instrumental in making t-SNE a competitive method compared to UMAP.

2.2.1 Attractive Force Optimisations

In a 2014 paper entitled "Accelerating t-SNE using Tree-Based Algorithms," (van der Maaten (2014)) Van Der Maaten et al. demonstrate that input similarities p_{ij}, can be approximated by only using k nearest neighbours. The probabilities associated with unrelated objects i and j are infinitesimally small (van der Maaten (2014)). As such, they developed a sparse approximation to the probabilities p_{ij} by finding the $\lfloor 3 * \text{Perplexity} \rfloor$ nearest neighbours for each of the n input objects. They computed these approximations using Vantage Point trees, thus reducing the original $O(n^2)$ quadratic time complexity to $O(n \log n)$ (Poličar (2020)).

In a 2017 paper entitled "Efficient algorithms for t-distributed stochastic neighbourhood embedding,"(Linderman et al. (2017)) Linderman et al. refined van der Maaten (2014)'s optimisation by approximating nearest neighbours instead of using exact nearest neighbours. The authors explain that using a new theoretical advance from Jaffe et al. (2017), connecting every point to its, for example, $k = 50$ nearest neighbours is not more accurate than connecting them to 2 randomly chosen points out of the 50 nearest neighbours. We refer the reader to the original paper for details. Briefly, the randomised procedure executed on point clouds lying on manifolds creates expander graphs at the local scale (Linderman et al. (2017)). These graphs represent the local geometry of the data accurately (Linderman et al. (2017)) and justify the use of computing approximate nearest neighbours instead of the exact ones, using a randomised nearest neighbour method called Annoy (Bernhardsson (2017)). Linderman et al.'s optimisation enable OpenTSNE (a popular Python package for t-SNE) to forego the use of Vantage Point trees and lower the time complexity of computing p_{ij} to below $O(n \log n)$.

2.2.2 Repulsive Force Optimisations

Barnes-Hut t-SNE is the first major improvement in repulsive force optimisation. Stemming in 2014 by van der Maaten (2014), it draws from particle simulations. The authors use space partitioning (metric) trees to approximate repulsive forces. They found that for two clusters Q and W, for any points $q_1, q_2 \in Q$ and $w \in W$ then $|q_1 - w| \approx |q_2 - w| >> |q_1 - q_2|$ i.e. the repulsive force from q_1 onto w is roughly equivalent to the repulsive force exerted from q_2 onto w (van der Maaten (2014)). We can hence find a cluster's **centre of mass** c_q for Q

and c_w for W as a point that summarises the interactions between a particular cluster and unrelated points: summarising the cluster's contribution to $\mathcal{F}_{rep,i}$. The authors use the Barnes-Hut tree algorithm (Barnes and Hut (1986)) to follow this intuition: "constructing a quadtree ... on the current embedding, (2) traversing the quadtree using depth first-search and (3) at every node ..., deciding whether a node's corresponding cell can be used as a summary" (van der Maaten (2014)). The authors define a condition θ to determine whether a cell is admissible as a "summary" for all the points in the cell:

$$\frac{r_{cell}}{|q_1 - q_{cell}|} < \theta$$

Where r_{cell} is the diagonal distance of the cell and q_{cell} is the centre of mass of the cell. The value θ represents our sensitivity to summarising the number of cells generated by the quadtree: a higher value of θ improves the speed of t-SNE, the Barnes-Hut algorithm summarises more cells but worsens its accuracy. Note that Barnes-Hut t-SNE with $\theta = 0$ computes all the pairwise interactions. t-SNE's time complexity improves from $O(n^2)$ complexity to $O(n \log n)$ on average. When using t-SNE in Python under the OpenTSNE (Policar et al. (2019)) package, we can use the `'bh'` modifier for this method.

The newest method to compute repulsive forces "Fast Fourier transform (`'fft'`)-accelerated interpolation-based t-SNE" (FIt-SNE) by Linderman et al. (2017) computes the repulsive forces in $O(2pn)$. It takes a small number p of interpolation points that control the interaction between nodes Poličar (2020). Then, it calculates the distance between the interpolation points (p) and the n nodes ($O(pn)$) and between each node ($O(n \log n)$ with FFTs) (Linderman et al. (2017). "Finally,... [they] interpolate from the p interpolation nodes to all the original points (also pn computations)" (Linderman et al. (2017)). The mathematical explanation of this method is beyond the scope of this report. An in-depth discussion is available in the original paper for the interested reader. To use this optimisation on t-SNE in Python under the OpenTSNE package, we can use the `'fft'` modifier for this method.

2.3 UMAP

The algorithm: Uniform Manifold Approximation and Projection (McInnes, Healy and Melville (2018)) is a non-linear dimensionality reduction method created by Leland McInnes, John Healy and James Melville and used for the visualisation of high-dimensional data. UMAP's foundation exists within topological data analysis and category theory. At a high level, UMAP uses local manifold approximations and patches together their local fuzzy simplicial set representations to construct a topological representation of the high dimensional data. Essentially, properties in topological data analysis and category theory prove that a fuzzy topological representation of the data will capture the topology of the manifold underlying the data. The algorithm then proceeds to convert this high-dimensional representation of the data to a low dimension. To explain UMAP, we proceed to show the algorithm and then explaining the intuition behind each step. As can be seen in Algorithm 1, there are two steps to UMAP: constructing a weighted graph motivated to capture the topology of the manifold and then finding a low dimensional representation for this graph.

The core of the UMAP algorithm, and key differentiation factor with t-SNE, is constructing local fuzzy simplicial sets to create a fuzzy simplicial complex: a weighted graph, see Algorithm 2. A simplex is a combinatorial building block to create a k-dimensional object by taking the convex hull of $k + 1$ independent point (McInnes (2018)). It is ultimately an

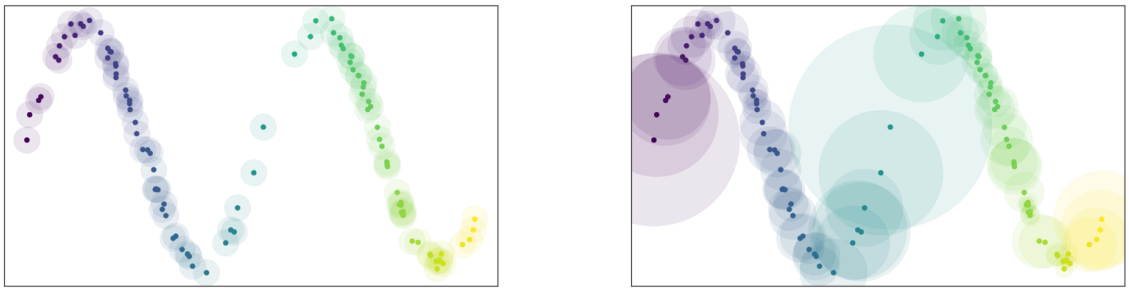
Algorithm 1 UMAP Algorithm

function UMAP(X, n, d, min-dist, n-epochs)

#Construct the weighted graph
for all $x \in X$ **do**
fs-set $\leftarrow$ LocalFuzzySimplicialSet(X, x, n)
end for
top-rep $\leftarrow \bigcup_{x \in X}$ fs-set$[x]$

 # Perform optimisation of the graph layout
$Y \leftarrow$ SpectralEmbedding(top-rep, d)
$Y \leftarrow$ OptimizeEmbedding(top-rep, Y, min-dist, n-epochs)
return Y
end function

arbitrary set of $k+1$ objects with faces given by a subset. We can agglomerate this simplex into a "simplicial complex." A simplicial complex $\mathcal{K}$ is a set of simplices such that any face of any simplex in $\mathcal{K}$ is also in $\mathcal{K}$: it is a way to ensure all faces exist. To build a simplicial complex from a topological space, the authors construct a Čech Complex given an open cover of a topological space (McInnes (2018)). An open cover C over a topological space $\mathcal{S}$ is defined as a collection of open subsets U_α where $C = U_\alpha : \alpha \in A$. A Čech Complex intuitively is a combinatorial way to convert the cover C into a simplicial complex (McInnes (2018)). The Čech Complex works as follows: let each subset U_α in the open cover be a 0-simplex (a point); create a 1-simplex (line) between the two open sets if their intersection is non-empty; and so on for three intersecting sets, etc. (McInnes (2018)). This process is guaranteed to produce a meaningful topological space by the Nerve Theorem see the original paper for more detail on the topological justifications McInnes, Healy and Melville (2018). The Nerve theorem especially guarantees that the simplicial complex generated is equivalent to the union of the cover. Think of the "cover" as a ball with radii r_α. If the data underlying the manifold is non-uniformly distributed, then covers of fixed radii do not adjust to the data see Figure 2.1b. We do not cover the underlying manifold. We took the two Figures in 2.1 from (McInnes (2018)) following fair use.



(a) Open cover with fixed radius.

(b) Open cover with local metric.

Figure 2.1: Open cover examples. (McInnes (2018))

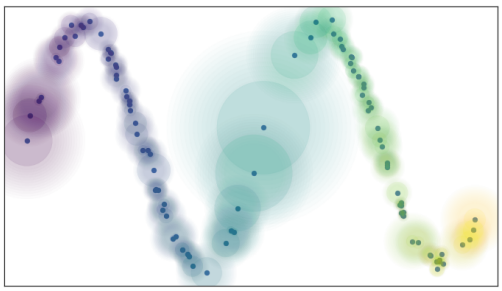
UMAP assumes that the data underlying the manifold is uniformly distributed. In the case that the data is indeed uniformly distributed: full connectivity between open covers ensues. On the other hand, the authors state that a varying notion of distance within the manifold explains non-uniformly distributed data. The algorithm approximates a local sense of distance for each point using the two following assumptions that we state without proof:

(1) the high-dimensional data is uniformly distributed on the Riemannian manifold and (2) the Riemannian metric is locally constant (McInnes (2018)). The mathematics involved to explain the inner workings of Riemannian Geometry is out of the scope of this project: we refer the interested reader to McInnes, Healy and Melville (2018). The assumptions from Riemannian Geometry allow UMAP to stretch the "covers" (or balls with radii r_α) to the k-th nearest neighbour, see Figure 2.1b (McInnes (2018)). Stretching the open cover introduces the notion of local metric and intuition of estimating the Riemannian metric: we no longer look at a fixed area but a varying one depending on the distance to the k-nearest-neighbour. In Algorithm 2, the construction of the k-neighbour graph encapsulates this theoretical development. A manifold may not be fully connected: it is wrong to assume otherwise. Using varying metrics across the manifold enables UMAP to preserve the local structure of the data as it is unlikely that points are isolated from the rest of the data.

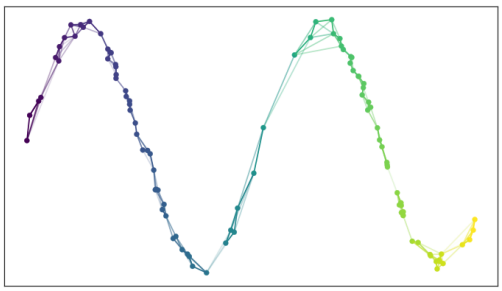
Algorithm 2 Constructing a local fuzzy simplicial set

function LocalFuzzySimplicialSet(X,x,n)
  knn, knn-dists $\leftarrow$ ApproxNearestNeighbors(X, x, n)
  $\rho \leftarrow$ knn-dists[1]
  $\sigma \leftarrow$ Binary search for σ such that $\sum_{i=1}^{n} \exp(-\text{knn-dists}_i - \rho)/\sigma = \log_2(n)$
  fs-set$_0 \leftarrow X$
  fs-set$_1 \leftarrow \{([x, y], 0) | y \in X\}$
  for all $y \in knn$ **do**
    $d_{x,y} \leftarrow \max\{0, \text{dist}(x, y) - \rho\}/\sigma$
    fs-set$_1 \leftarrow$ fs-set$_1 \cup ([x, y], \exp(-d_{x,y}))$
  end for
  return fs-set
end function

UMAP weighs the edges of the k-neighbour graph according to their distance away from one another. The authors define the varying weights as working in a fuzzy topology: in the graph, the edges are no longer binary (1 -connected, 0 - not), but "fuzzy" values between 0 and 1 (McInnes (2018)). However, they found that a fuzzy cover could leave out points isolated. To ensure a minimum of local connectivity, the authors introduced strict confidence within the open sets to the *first* nearest neighbour and having fuzzy confidence decay beyond that initial nearest neighbour for all data points see Figure 2.2a. We took the two Figures in 2.2 from (McInnes (2018)) following fair use. Again, this minimum boundary on local connectivity does not force a fully-connectedness condition on the embedding: it ensures that isolated points are at least in one local neighbourhood as the contrary is unlikely.



(a) Local connectivity and fuzzy open sets.



(b) Graph with combined edge weights.

Figure 2.2: Local connectivity & fuzzy open sets and final graph. (McInnes (2018))

Currently, the mathematical idea behind local metric solves a lot of problems to embed the high-dimensional data. However, the local metrics are incompatible. Let a and b two neighbouring points on the manifold: from point a to b the distance could be 0.3, whereas from b to a it might be 0.7. The initial intuition behind simplicial sets helps us here. Since we have a family of fuzzy simplicial sets, we take the union of those sets to merge differing metrics (McInnes (2018)). There are multiple ways to define the union of simplicial sets: the authors define the union between two disagreeing edges as $a + b - a \cdot b$ (McInnes, Healy and Melville (2018)). This final intuition allows UMAP to compute a proper k-neighbour graph as seen in Algorithm 2. Figure 2.2b provides a visualisation of the graph.

In the real world, to find a low dimensional representation in the euclidean space, the distance on the manifold has to be a standard Euclidean distance for the global coordinate system and not a varying metric (McInnes (2018)). Additionally, the distance between the nearest neighbour also has to be globally comparable across the manifold to work towards a low dimensional representation (McInnes (2018)). In the Python implementation of UMAP (McInnes, Healy, Saul and Grossberger (2018)), `min-dist` represents the minimum distance between nearest neighbours.

In the UMAP algorithm 1, the final step is to find a *good* low dimensional representation of the fuzzy simplicial complex. The authors formulate this as an optimisation problem: "finding the low dimensional representation with the closest fuzzy topological structure" (McInnes (2018)). Ultimately, the weights in the k-nearest neighbour graph and its resulting union provide UMAP probabilities of the simplex existing. The low-dimensional and high-dimensional topological structures have the same points (0-simplices): UMAP compares their distance (1-simplices). The 1-simplices are probabilities: they exist or not. The optimisation problem of finding a faithful representation of the high-dimensional topological structure is modelled using Bernouilli variables and cross-entropy loss (McInnes (2018)).

Let E be the set of all possible 1-simplices (or graph edges). We have functions $\mu(e)$ for the weights of the 1-simplex e in the high dimension and $\upsilon(e)$ for its low dimensional representation, we minimise the following cross-entropy loss (McInnes, Healy and Melville (2018)):

$$\sum_{e \in E} \mu(e) \log\left(\frac{\mu(e)}{\upsilon(e)}\right) + (1 - \mu(e)) \log\left(\frac{1 - \mu(e)}{1 - \upsilon(e))}\right)$$

or

$$\sum_{e \in E} \mathcal{F}_{attractive}(e) + \mathcal{F}_{repulsive}(e)$$

Similarly to in t-SNE, we can deconstruct UMAP into its attractive forces ($\mathcal{F}_{attractive}(e)$) and repulsive forces ($\mathcal{F}_{repulsive}(e)$). As υ grows (the points are most dissimilar), the attractive forces shrink, and as the weights in the high dimensional case μ grow, the attractive force grows. The second term ($\mathcal{F}_{repulsive}(e)$) is thought of as the repulsive force between the points span by e when $\mu(e)$ is small (McInnes (2018)).

To improve UMAP's accuracy and runtime speed, the authors of the Python package (McInnes, Healy, Saul and Grossberger (2018)) made several changes to the algorithm. First, the authors noticed that beyond the *first*-nearest neighbour, the fuzzy set membership decay became infinitesimally small. The authors profit from Dong et al. (2011)'s work on an efficient Nearest Neighbour algorithm to only compute the set membership for the nearest neighbour

Algorithm 3 Spectral embedding for initialisation

function SpectralEmbedding(top-rep, d)
  $A \leftarrow$ 1-skeleton of top-representation expressed as a weighted adjacency matrix
  $D \leftarrow$ degree matrix for the graph A
  $L \leftarrow D^{\frac{1}{2}}(D - A)D^{\frac{1}{2}}$
  evec $\leftarrow$ Eigenvectors of L (sorted)
  $Y \leftarrow$ evec$[1..d + 1]$ **return** Y
end function

of each point (McInnes (2018)). Second, the loss function is re-arranged using a smooth approximation of the membership weight function v for the low dimensional representation using the family of curves of the form $\frac{1}{1+ax^{2b}}$. Finally, spectral embedding is performed by considering the 1-skeleton of the global fuzzy topological representation as a weighted graph and using Laplacian eigenmaps (Belkin and Niyogi (2003)) see Algorithm 3 to initialise the fuzzy simplicial complex (McInnes, Healy and Melville (2018)).

The nearest neighbour search phase forms an upper bound on UMAP's time complexity. In the worst case, UMAP has a time complexity of $O(n^2)$. Leland McInnes estimates that the empirical average is more around $O(d \cdot n^{1.14})$, where the dimension of the data is given by d (McInnes, Healy and Melville (2018)). However, in the average case, we can consider that UMAP scales at $O(n \log(n))$.

2.4 HDBSCAN

In this report, we offer a comparison of clustering with t-SNE and UMAP and how they improve clustering of high-dimensional data. To do so, we need a clustering algorithm. Many clustering algorithms are available, such as K-Means, Gaussian Mixture Models (GMM), Hierarchical Clustering and Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCA). In this report, we use HDBSCAN over other alternatives as it is an efficient and relatively accurate algorithm with the additional benefit of removing noisy data. It is important to note that probabilistic clustering models, such as using GMM, are slightly slower but produce more accurate results and can discover noisy data. We use HDBSCAN for simplicity. In this report, we provide an intuitive explanation (Leland McInnes (2017)) of HDBSCAN and direct the interested reader to the original paper for more details (Campello et al. (2013)).

Leland McInnes (2017) describes HDBSCAN in five steps:

- Transform the space according to the density of the data and build a distance weighted graph.
- Construct the minimum spanning tree of the distance weighted graph.
- Construct a cluster hierarchy of connected components.
- Condense the cluster hierarchy based on the minimum cluster size.
- Extract the stable clusters from the condensed tree.

The core benefit of HDBSCAN is to find clusters by finding higher density data clusters amongst sparser noise: removing outliers and noise. To estimate density within a dataset

efficiently, HBDSCAN computes the distance between k nearest neighbors and builds a distance matrix. In HDBSCAN, the distance metric is defined as **core distance** for parameter k for a point $x \in X$, denoted by $\text{core}_k(x)$. To emphasize similar points (low core distance) and spread apart points with low density, Campello et al. (2013) defines the **mutual reachability distance** for point $a, b \in X$ as follows:

$$d_{mreach-k}(a, b) = max\{\text{core}_k(a), \text{core}_k(b), d(a, b)\}$$

where $d(a, b)$ is the original metric distance. We refer the interested reader to the following literature (Eldridge et al. (2015)) to understand the benefits of using **mutual reachability distance** to allow single-linkage in hierarchical clustering to more closely approximate the true density distribution of our points (Leland McInnes (2017)).

The next step is to construct the minimum spanning tree of the distance weighted graph. The HDBSCAN algorithm considers the data from the embedding as a weighted graph with edges between points representing their "mutual reachability" (Leland McInnes (2017)). The algorithm builds the minimum spanning tree using Prim's algorithm (Prim (1957)) to obtain a hierarchy of connected components at varying threshold levels: dropping edges with values below a prescribed threshold value.

Then the algorithm builds a cluster hierarchy: iterating through the edges by ascending magnitude of distance in the tree and creating a new merged cluster for each edge (Leland McInnes (2017)). At this stage, we obtain a single-linkage hierarchical clustering structure that provides information about the interactions of the data in the dataset.

The next step is to condense the cluster hierarchy based on minimum cluster size. The algorithm introduces a hyper-parameter called **minimum cluster size**. The minimum cluster size represents a minimum threshold value for the algorithm to separate a cluster from the larger pool of data points. The algorithm walks through the clustering hierarchy: comparing the size of a cluster at each iteration to the minimum cluster size. If a cluster happens to be smaller than the minimum cluster size, then HDBSCAN treats the cluster as retaining the cluster identity of the parent (Leland McInnes (2017)). If the split created two clusters at least as large as the minimum cluster size then HDBSCAN treats them as true clusters (Leland McInnes (2017)). The condensing process ensures that HDBSCAN creates a condensed hierarchical structure and removes incentives from branching off.

Finally, to extract the stable clusters from the condensed tree, we compute the stability of each cluster. To consider the persistence of clusters, the algorithm uses a new distance metric $\lambda = \frac{1}{\text{distance}}$ (Leland McInnes (2017)). For any given cluster, the authors define the values λ_{birth} and λ_{death}: representing the lambda value when the cluster split off and became its cluster and its lambda value when it split into smaller clusters (if any) respectively. For any given cluster, we define for a point p in the cluster the value λ_p as the value at which that point 'fell out of the cluster'. We compute the stability of the cluster as:

$$\sum_{p \in \text{cluster}} (\lambda_p - \lambda_{birth})$$

To discover the clusters the algorithm starts at the leaf nodes. If the sum of the children clusters is greater than the parent cluster's stability then the cluster's stability becomes that sum. On the other hand, if the parent cluster is more stable then we declare the cluster as selected and unselect its descendants (labelling them as noise) (Leland McInnes (2017)). HDBSCAN returns the root node.

Chapter 3

Results

3.1 Simulation

3.1.1 Gaussian Distribution Synthetic Data

To begin this results chapter, we start with using PCA, t-SNE and UMAP on a simulation of synthetic data to motivate the use of t-SNE and UMAP over the traditional use of PCA. PCA is a traditional linear dimensionality method to use to embed high-dimensional data into the euclidean space (in our case). PCA is a very efficient method that enables us to capture the global structure of data proficiently.

To numerically measure the quality and faithfulness of a given embedding, we used three different metrics; **KNN**, **KCN** and **CPD**. These are standard metrics in the space of comparing dimensionality reduction methods as seen in Kobak and Berens (2019), Lee and Verleysen (2009) and Becht et al. (2019):

- **KNN** quantifies the preservation of the local structure. We use k-nearest neighbour embedding to embed the original data. In our case we use $k = 10$. Then we compare this embedding with the k-nearest neighbour embedding of the embedding from one of our methods (PCA, t-SNE and UMAP). We create a score as the fraction of k-nearest neighbours in the original data preserved as k-nearest neighbours in the embedding (Lee and Verleysen (2009)).
- **KCN** quantifies the preservation of the 'mesoscopic structure" or medium-sized structure. We use k-nearest *class* means to embed the original data. In our case, we use $k = 4$ for the synthetic dataset. In the later MNIST experiments, we use $k = 10$. Then we compare this embedding with the k-nearest class means in the embedding from one of our methods (PCA, t-SNE and UMAP). Creating a score as the fraction of k-nearest class means in the original data preserved as k-nearest class means in the embedding.
- **CPD** quantifies the preservation of the macroscopic or global structure. CPD is the Spearman correlation between pairwise distances in the high-dimensional space and the embedding (Becht et al. (2019)). We compute the CPD across all the pairs among 1,000 randomly chosen points.

First, we explain the first synthetic dataset X_1. X_1 contains 15,550 50 dimensional data points x_i. We create the first synthetic dataset based on Kobak and Berens (2019)'s first simulation. We sample the data from 15 50-dimensional spherical Gaussian distributions.

We split the data into three distinct classes: one with five samples of size 2000, another with five samples of size 1000 and finally, one with five samples of size 100. The classes of size 1000 and 100 do not have overlapping points with the other classes. The samples in the other class ($n = 2000$) have some overlapping points. This simulated data has the additional benefit of exhibiting hierarchical structure. This type of structure is prevalent in RNA-Seq data in single-cell analysis, a major use case of t-SNE and UMAP (Kobak and Berens (2019)). Using no-overlap would result in trivial analysis. The interested reader can find the code in Section A.1 (Package Import (A.1.1), Synthetic Data Creation (A.1.2), Metrics Design (A.1.3), Methods Design (A.1.4), Metrics Computation (A.1.5), Plotting Embeddings (A.1.6)).

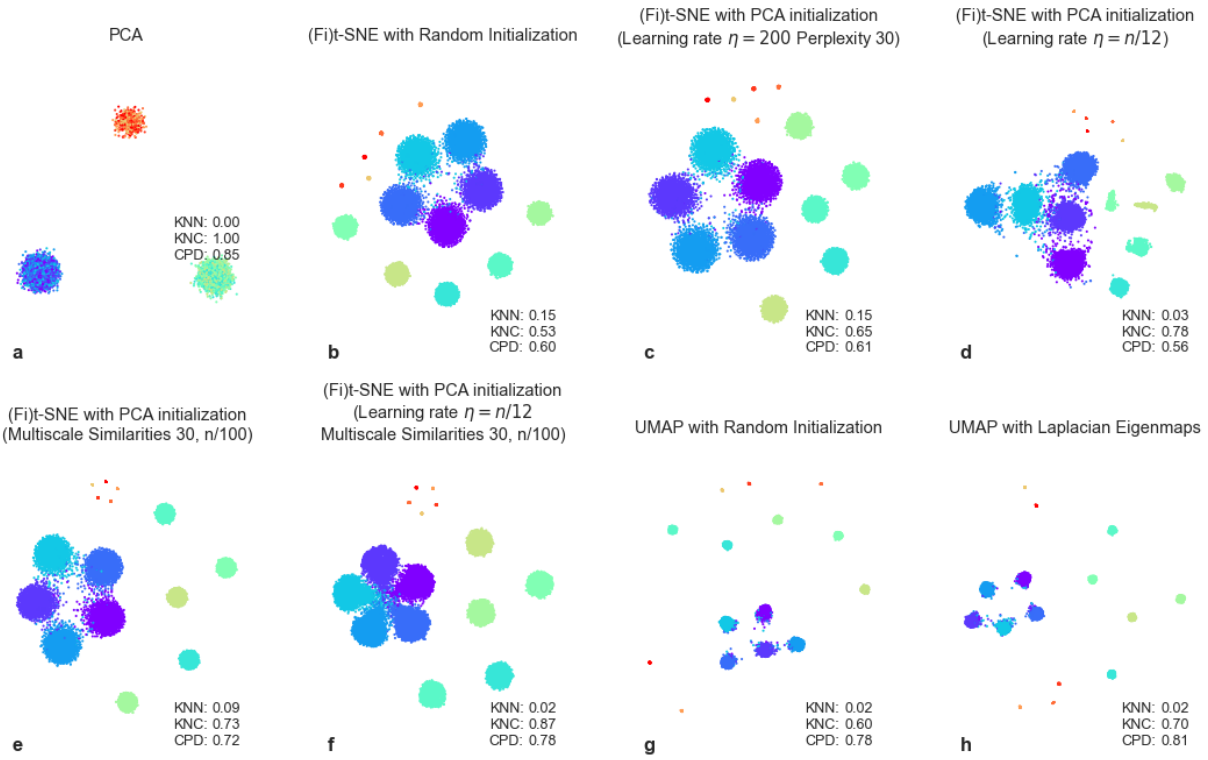


Figure 3.1: Simulations using synthetic data. Each color represents a different class with larger dot representing the size of the cluster.

The results can be found in Figure 3.1. To understand the differences, we detail the different methods used in **a**, **b**, **c**, **d**, **e**, **f**, **g**, and **h**. Method **a** uses PCA to embed the 50 dimensional data in X into two principal components represented in the euclidean space. It can be seen from the high KNC and CPD values, 1.0 and 0.85 respectively, that PCA is good at preserving the mesoscopic and global structure of the data at the expense of the local structure. From the figure, we can only discern three clusters.

Method **b** uses t-SNE with the random initialisation, a default perplexity of 30 and a learning rate of 200. Larger perplexity values are associated with larger ranging attractive forces and produce different results as we will see later. In figure **b** we clearly see 15 clusters. This method performs better than PCA in preserving the local structure (KNN 0.0 vs 0.15), however, less faithfully preserves the global structure (KNC 1.00 vs 0.53 and CPD 0.85 vs 0.60).

UMAP with a random initialisation (**g**), performs better than traditional t-SNE in preserving

the global structure (KNC 0.6 vs KNC 0.53 and CPD 0.78 vs CPD 0.6) but in this simulation, this comes at the expense of losing information regarding the local structure (KNN 0.02 vs 0.15).

In literature, it is unusual to find analysis using the 'standard settings' and random initialisations. To provide a more faithful t-SNE visualisation three main ideas exist: using PCA initialisation, increasing the learning rate η (Belkina et al. (2019)) and using Multi-scale similarities (Lee et al. (2015)). t-SNE using PCA initialisation can be seen in Figure **c**. PCA initialisation allows us to boost t-SNE's capabilities to preserve global geometry by injecting the data's global structure. This remains relatively unchanged as t-SNE optimises the local structure. In this report, we provide various versions of t-SNE to provide a faithful comparison of the two methods. This new version of t-SNE better preserves the mesoscopic and macroscopic structures than the one in **b** (KNC 0.65 vs 0.53 and CPD 0.61 vs 0.60). However, compared to the random initialisation of UMAP (**g**) preserves less the global structure.

In **d** we use t-SNE with PCA initialization and increase the learning rate to $\eta = n/12$. The default learning rate in OpenTSNE is 200, however, as noticed in Belkina et al. (2019), this is often too low for larger datasets. A low-learning rate causes t-SNE to poorly converge to an optimum or converge to a sub-optimal local minimum. It has become standard in single-cell data analysis amongst researchers using t-SNE to use a learning rate of $\eta = n/12$ as a rule of thumb as it ensures convergence. The results in **d** show improvements in the mesoscopic structure over t-SNE in **c** (KNC 0.78 vs 0.65), however, the metric for both the local and global structure decrease (KNN 0.03 vs 0.15 and CPD 0.056 vs 0.60).

In **e** we make use of "Multi-scale Similarities." t-SNE with Multiscale similarities uses a multiscale kernel which accounts for two different perplexity values instead of a typical Gaussian kernel (Poličar (2020)). Enabling us to obtain a better separation between clusters using Perplexity $= n/100 = 155$ whilst maintaining much of the global structure using Perplexity $= 30$. As a result, we obtain in **e** a slightly less faithful representation of the local structure than in our t-SNE with PCA approach (**c**) (KNN 0.09 vs 0.15) but preserve the global structure much more accurately (KNC 0.73 vs 0.65 and CPD 0.72 vs 0.61).

Method **f**, combines all the improvements (PCA, learning rate and multiscale similarities) as in the pipeline suggested by Kobak and Berens (2019). On this dataset, we find that the combination leads to a less faithful representation of the local structure (KNN 0.2 vs 0.09 (**e**) vs 0.15 (**c**)) but a faithful representation of the global structure (KNC 0.87 vs 0.73 (**e**) and CPD 0.78 vs 0.72 (**e**)).

	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**
Runtime (s)	0.439	28.4	28.1	23.0	90.8	87.9	9.65	5.93

Table 3.1: Runtime for the simulations - CPU (8-Core 9th Generation Intel Core i9 2.3GHz-4.8GHz)

We introduce UMAP with Laplacian Eigenmaps. Laplacian Eigenmaps improve the quality, efficiency and faithfulness of a given embedding using UMAP. We apply the method to our synthetic data and show the results in **h**. Upon inspection, we see 15 clusters more spread apart than for our t-SNE methods. We find that this UMAP method provides a slightly better embedding (than **g**) with regards to the global structure of the data (CPD 0.81 vs 0.78 (**f**)) but a comparable local embedding to **f** (KNN 0.02).

Finally, we look at the runtime of each of these methods on this dataset, see Table 3.1. Clearly on this dataset PCA is the most efficient, (0.439s). UMAP appears to be substantially faster than any of the t-SNE methods (5.93s (**g**) and 9.65s (**h**) vs 23.0s (**d**) when using a high-learning rate in t-SNE). It appears that using multi-scale similarities greatly slows down our computation of the embedding (90.8s (**e**) and 87.9s (**f**)). All the t-SNE methods were computed using interpolation. To evaluate the efficiency of t-SNE vs UMAP we will conduct more in-depth analysis, see 3.3. For the rest of our experiments, we use methods **a**, **c**, **e**, **f**, **g**, and **h**.

3.1.2 Circle

In Kobak and Linderman (2021), the authors use a circle to demonstrate the difference in embedding between t-SNE and UMAP methods with random and specialised initialisation: PCA for t-SNE and LE for UMAP. In this simulation, we use two circles: one with 7k dots, see Figure 3.2, and the other with 25k dots, see Figure 3.3. We explore the example exposed in Kobak and Linderman (2021) further by selecting a larger dataset (25k samples) and comparing the different UMAP and t-SNE pipelines we introduced in 3.1.1. This simple example shows the importance of initialisation and provides an ideal visualisation to discuss differences between the two methods. The interested reader can find the code in Section A.2 (Package Import (A.2.1), Synthetic Data Creation (A.2.2), Metrics Design (A.2.3), Methods Design (A.2.4), Metrics Computation (A.2.5), Plotting Embeddings (A.2.6)): note that for the 25,000 dataset we replace $n = 7000$ with $n = 25000$.

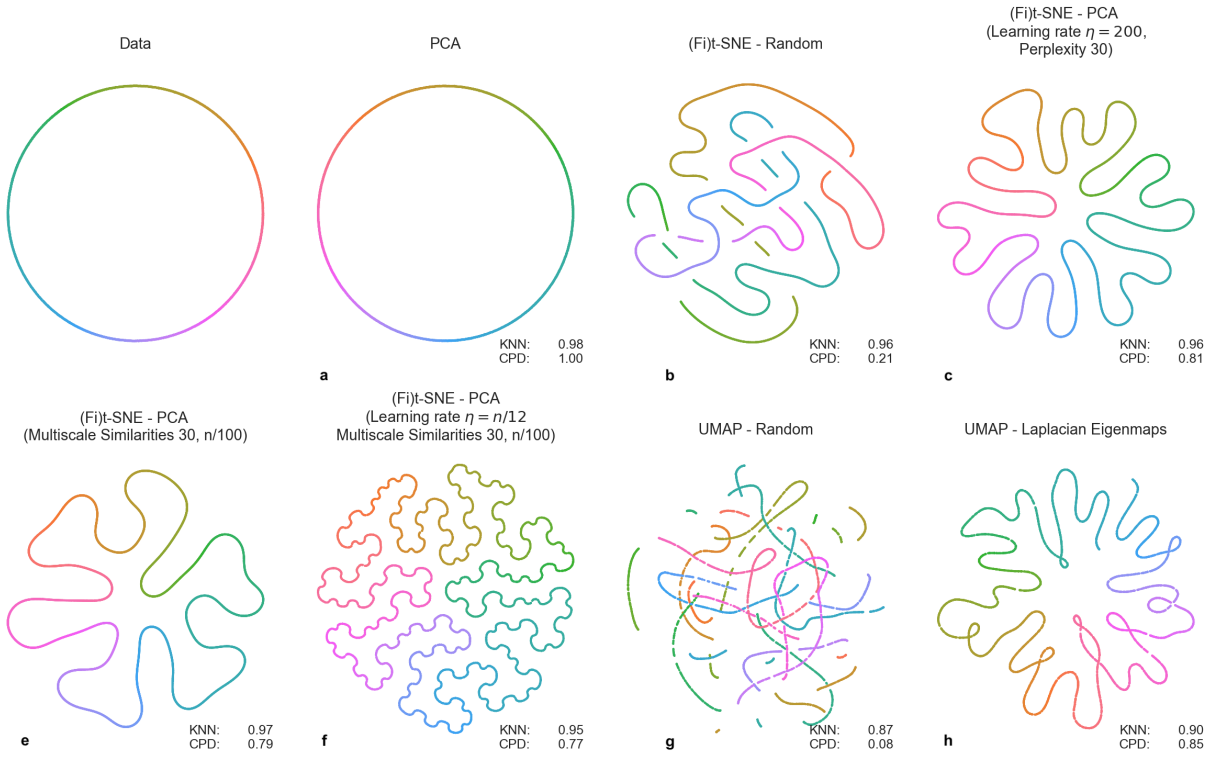


Figure 3.2: Circle with 7000 samples

Since a circle is a two-dimensional object, PCA accurately and faithfully represents the circle data on both datasets. In the smaller dataset, PCA preserves the local structure extensively (KNN 0.98) and the global structure wholly (CPD 1.0). Its accuracy falls on the larger dataset concerning preserving the local structure, KNN 0.86, however, it persists to preserve

the global structure completely, CPD 1.0. PCA with two components captures 100% of the variance in dataset X in its two principal components it obtains high scores on the KNN and CPD metrics.

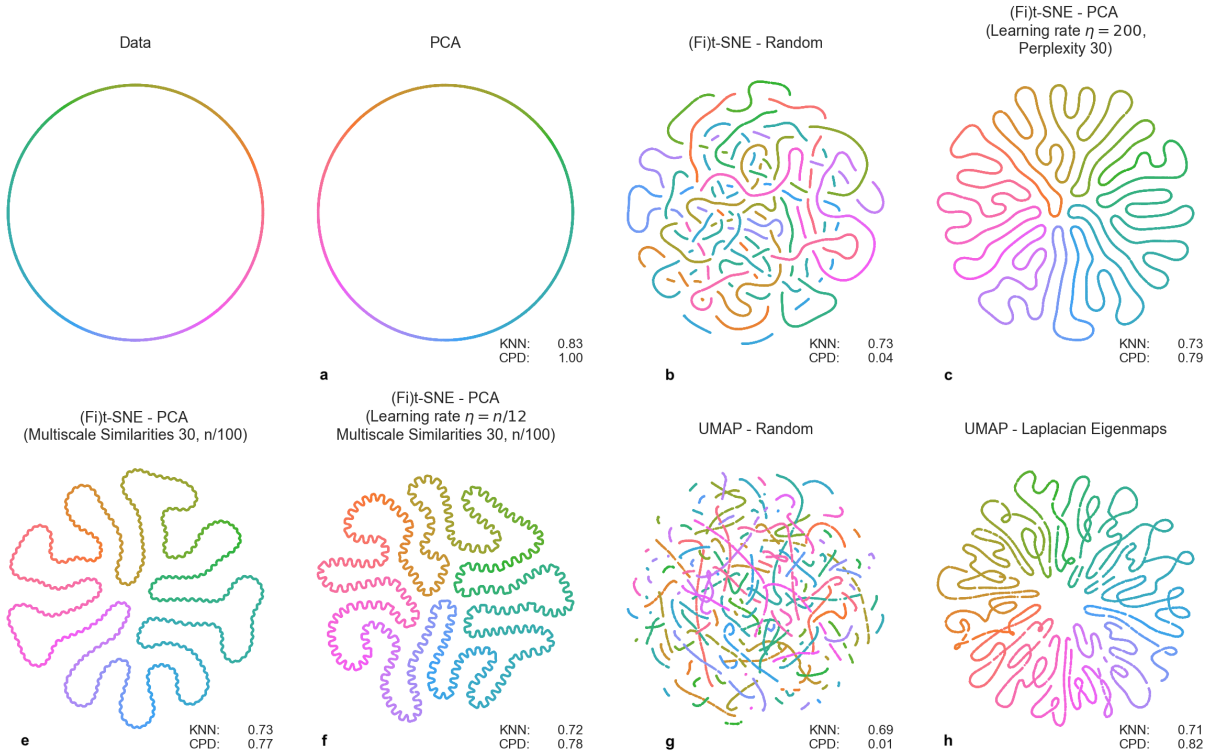


Figure 3.3: Circle with 25,000 samples.

In t-SNE, we discovered that the version using a random initialisation badly preserves the global structure of the data in the Euclidean space. The best version with regards to the KNN and CPD metrics on both datasets are (Fi)t-SNE with PCA initialisation with Perplexity = 30 and Learning Rate = 200, with KNN 0.96 & 0.73 and CPD 0.81 & 0.79 for the 7k and 25k datasets respectively. We found that t-SNE with a random initialisation produces an entangled embedding: poorly preserving the global structure with a CPD score of 0.21 (7k) and 0.04 (25k). However, it preserves the local relationship of the data equally to our best t-SNE method: KNN score of 0.96 vs 0.96 (7k) and 0.73 vs 0.73 (25k). PCA initialisation enables the default setting t-SNE to effectively preserve the global structure of the circle with a CPD of 0.81. PCA acts by injecting the global interactions within our data in the t-SNE algorithm resulting in a more faithful embedding.

Using UMAP, we discovered the same as per t-SNE: the version with random initialisation did not capture the global structure of the embedding. The best version of UMAP uses Laplacian Eigenmaps initialisation, producing scores on the KNN metric of 0.90 & 0.85 and the CPD metric of 0.79 & 0.82 for the 7k and 25k datasets respectively. The entanglement resulting from the random initialisation of UMAP is more pronounced than in the case of t-SNE. But, the same observation remains: randomly initialised UMAP poorly preserves the global structure of the circle data on both datasets (CPD of 0.08 (7k) and 0.01 (25k)). We find that switching from the random initialisation of UMAP to the LE version improves the preservation of the local structure of the data (KNN 0.87 vs 0.90 (7k) and 0.69 vs 0.71 (25k)). LE, in this dataset, not only injects the global structure of the circle in the UMAP algorithm but also improves the preservation of the local structure.

	a	b	c	e	f	g	h
Runtime (s)							
7k	0.00475	32.7	29.1	43.7	23.3	8.34	8.32
25k	0.0102	79	72	208	396	25.7	96.0

Table 3.2: Runtime for the circle simulations - CPU (8-Core 9th Generation Intel Core i9 2.3GHz-4.8GHz)

Even though X is two dimensional, the intuition behind using PCA to initialise the data in the t-SNE case and LE in UMAP becomes clear. The important take-away from this simulation is that PCA and LE are not a diversion from t-SNE or UMAP, nor do they change the essence of the methods. They are simply initialisation tools that improve t-SNE and UMAP by aiding the preservation of the global structure of the data. We also note that we are yet to see striking differences in the KNN, KCN, and CPD metrics between the t-SNE and UMAP variants. On both datasets, UMAP (LE) preserves more of the global structure than any other t-SNE methods and t-SNE preserves more of the local structure. One of the biggest difference between the two methods is the visualisation embedding they produce in 2-D. Looking at Table 3.2 for details on runtime, we notice that UMAP is substantially faster than the more informed t-SNE methods (**e** and **f**).

3.2 MNIST

Until now, we compared PCA, UMAP and t-SNE holistically on the three metrics KNN, KNC and CPD, and assuming they have the same importance. In Section 2.1, we introduced the notion of intrinsic dimensionality, stating that every dataset X of dimension D has a lower intrinsic dimensionality $d << D$. This has gone unmentioned in our analysis so far. It is important to restate that unless $d = 2$, it might not be possible to faithfully transcribe the notion of distance from high dimensions into the euclidean space. As such, it is almost impossible to guarantee the global structure presented in the euclidean space is faithful to its higher dimensionality representation. t-SNE and UMAP focus on local neighbourhoods embedding as it is the only information we can realistically transcribe into the euclidean space. As such, KNN's importance surges over the two other metrics. It does not mean that the other metrics are redundant per se: they give us a way to quantify how PCA, t-SNE and UMAP preserve the mesoscopic and macroscopic structures. However, if the intrinsic dimensionality of the data is much higher than two, it is not possible to have a method "better" preserve the macroscopic structure. One must be careful not to over-interpret structure or distance as we can interpret clusters that might not exist in higher dimensions.

3.2.1 Hand Digit Recognition

The MNIST digits dataset (Lecun et al. (1998)) is a dataset composed of 60,000 training images and 10,000 images of handwritten digits. It is a subset of the larger NIST dataset that has been size-normalised and centred. The dataset contains ten classes, each representing one of the ten digits (0-9). Each digit is a grayscale image of dimension 28 by 28. The MNIST digits dataset is a benchmark in machine learning, the field of computer vision, and manifold learning. We flatten the images into a single 784 vector: treating the dataset as a 70,000 by 784 matrix. The interested reader can find the code for the experiments on MNIST

Digits in Section A.3 with details regarding each step (Methods and MNIST Digits (784) import(A.3.1, A.3.3), Metrics Design (A.3.2), Methods Design (A.3.4), Metrics computation (A.3.5), Plotting the Embedding (A.3.6), Computing the Clusters (A.3.7), and Plotting the Clusters (A.3.8)).

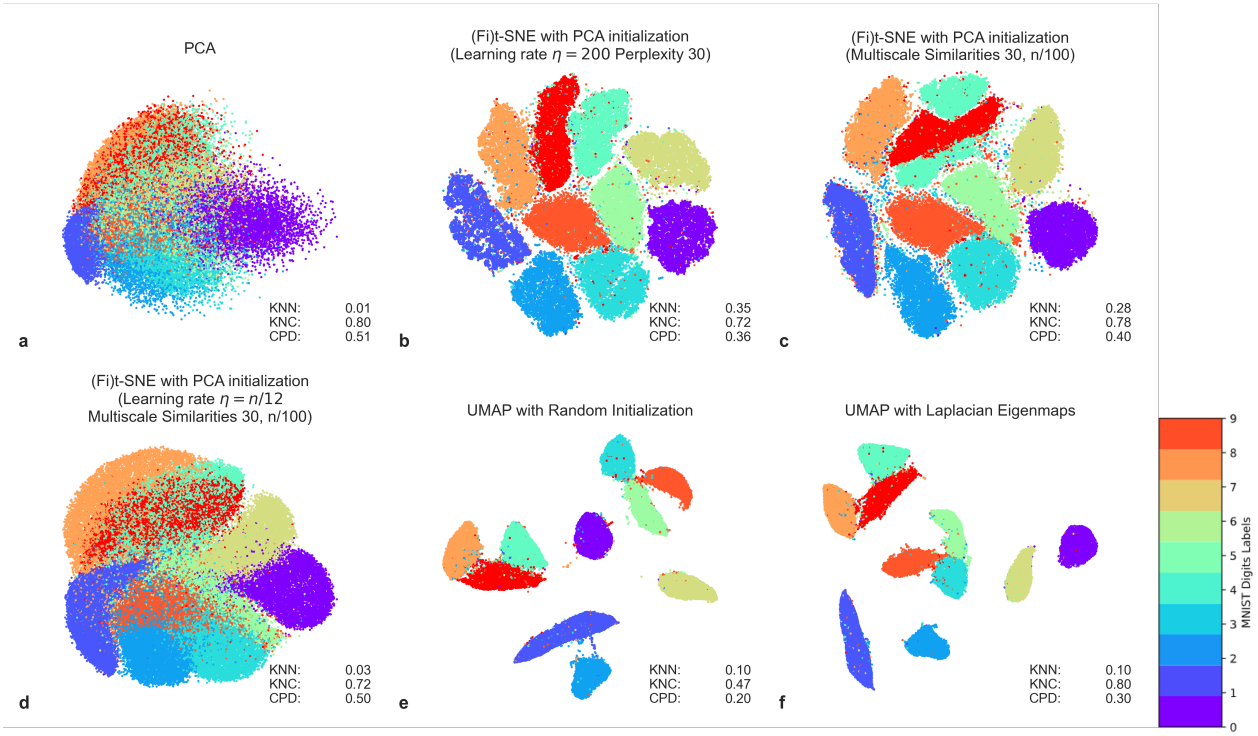


Figure 3.4: Embedding using methods **a**, **c**, **e**, **f**, **g**, and **h** of the MNIST Digits dataset.

We make use of our analysis on the synthetic data by only using the methods that performed best. PCA on the MNIST Digits dataset provides a faithful presentation of the mesoscopic structure (KNC 0.8) and the global structure (KNC 0.5). But PCA fails to faithfully preserve the high dimensional data's local structure (KNN 0.01). The method that best presents the local structure, highest KNN, is method **b** with a KNN of 0.35. Within the group of t-SNE methods, we observe that adding tools to interpret the global structure further, multi-scale perplexity and a higher learning rate (**c** and **d**), degrades the preservation of the local structure. By adding multi-scale similarities of 30 and $n/100 = 700$, the embedding in **c** better preserves the mesoscopic (KNC 0.78 vs 0.72) and the macroscopic structures (CPD 0.39 vs 0.34) of the MNIST data at the expense of weaker preservation of the local structure (KNN 0.28 vs 0.35). In increasing the learning rate from $\eta = 200$ to $\eta = n/12 = 5,833$ using method **d** we obtain an embedding that preserves better the macroscopic structure (CPD 0.46 vs 0.39) at the expense of a weaker preservation of the mesoscopic structure (KNC 0.72 vs 0.78) and almost no preservation of the local structure (KNN 0.03 vs 0.28).

We skip over UMAP with random initialisation and discuss UMAP with Laplacian Eigenmaps initialisation. UMAP (random) performs worse than UMAP (LE) on all metrics (KNN, KNC, CPD). Visually, the embedding shows a rather clear separation between the ten classes. However, quantitatively, the faithfulness of the preservation of the microscopic (KNN 0.1 vs 0.35 (**c**)) and macroscopic structure (CPD 0.32 vs 0.34 (**c**)) is worse than the t-SNE methods detailed in **b** and **c**. The higher KNC in UMAP (KNC 0.8 vs 0.72 **b** and 0.78 **c**) exemplifies that UMAP better marks separation in cluster than its t-SNE

counterparts. We expect this embedding to display high clustering accuracy with HDBSCAN as it separates classes in MNIST digits more clearly.

To evaluate the accuracy of the clusters created with regards to the true labels, we introduce two metrics: Adjusted Mutual Information (MI) and Adjusted Rand Index (ARI):

- **(A)MI** The mutual information score computes the similarity between two clusterings U and V. It is a measure of the similarity between two labels of the same data. We adjust this score for chance accounting for MI's tendency to assign a higher score for two clusterings with a larger number of clusters regardless of the amount of information shared (scikit-learn developers (2007-2020a)).
- **ARI** The Rand Index score computes the similarities between two clusterings by considering all the pairs of samples and counting pairs assigned to the wrong cluster in the predicted clusterings from the true labels (scikit-learn developers (2007-2020b)). We adjust this score for chance by subtracting the expected Rand Index.

To obtain the first scatter plot "HBDSCAN" in **a**, we let HBDSCAN run on the MNIST Digits dataset X acting as a baseline. It enables us to analyse how t-SNE and UMAP embeddings can aid HBDSCAN in uncovering the underlying local and global structure of the high dimensional representation and cluster accordingly. Remember from Section 2.4 that HDBSCAN works by only clustering data points it is confident are not noise. In the baseline, it only clusters 16.88% of the data. The amount of data clustered in PCA and for t-SNE with a high learning rate are low (50.59% and 86.61%), we disregard them in this analysis.

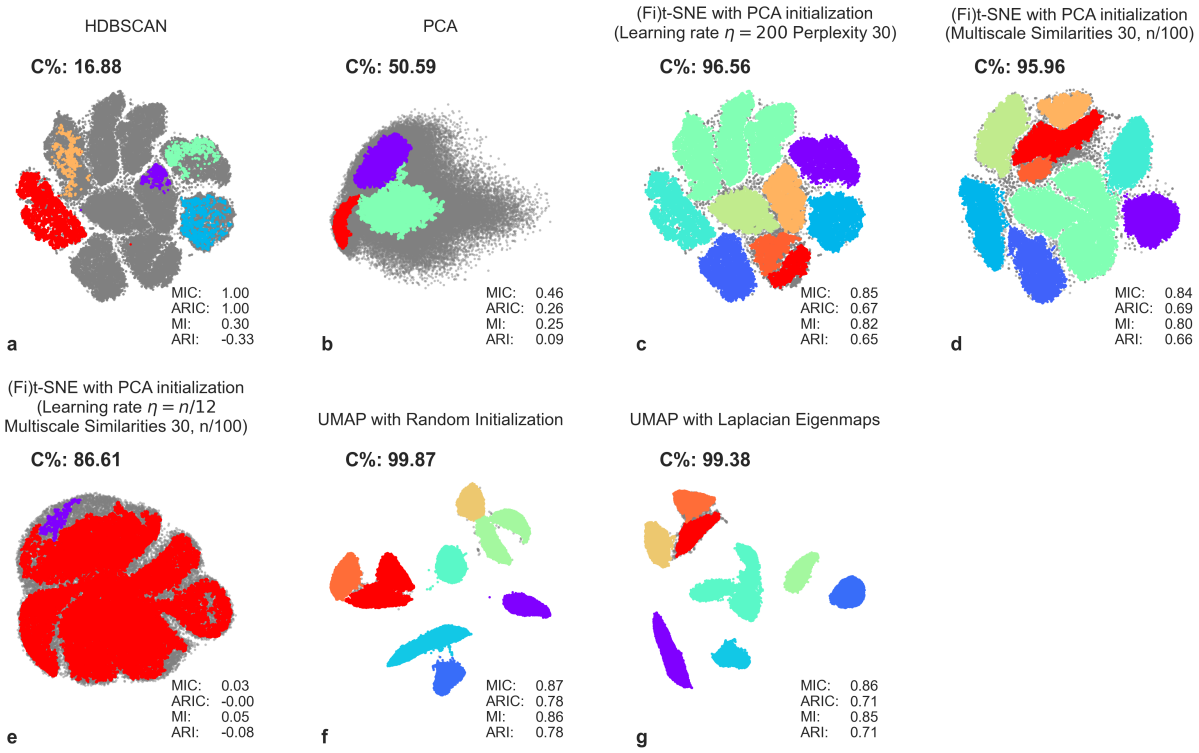


Figure 3.5: Clustering MNIST Digits using HDBSCAN.

Out of the two UMAP methods, UMAP with a random initialisation obtains higher global

scores than the UMAP with LE initialisation (MI 0.86 vs 0.85 and ARI 0.78 vs 0.71). HDBSCAN in both embeddings only uncovers 8 clusters (versus the ten classes). For UMAP (LE), the error lies in clustering the middle blob together: composed of 3 classes (the digits 2,5, & 8). For UMAP (random), the red (4 & 9) and green blob (5 & 8) group four different classes.

HDBSCAN on the the t-SNE methods (**c** and **d**) perform more poorly. Since the clusters are more clumped up, HDBSCAN classifies the data points separating the big clusters as noise: the t-SNE methods have a lower clustering rate (96.58% and 95.96%). In **c**, we see that HDBSCAN merged three different clusters (the digits 2,5, & 8) at the top of the cluster graph thus accounting for the majority of the errors. It is possible that with exaggeration, we can obtain an embedding more suited for HDBSCAN clustering. We also obtain two clusters for digit 2(red and orange) instead of one at the bottom of the plot. The two t-SNE methods provide similar quantitative results (MI 0.82 **c** vs MI 0.80 **d** and ARI 0.65 **c** vs 0.66 **d**).

Overall, the clearer clusters in **f** and **g** allow HDBSCAN on UMAP to outperform HDBSCAN on t-SNE. It is important to note that it is possible to exaggerate the t-SNE plots by increasing the repulsive forces between unrelated points.

3.2.2 Fashion

The MNIST Fashion dataset (Xiao et al. (2017)) comprises 70,000 28 by 28 grayscale images split into ten categories: T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle boot. With the additional advances in computer vision since 1998, a team of researchers envisioned MNIST fashion as a more complex replacement benchmark to the MNIST digits dataset. The images are substantially more complex than their digits counterpart. As for the MNIST digits dataset, we flatten the images into a single 784 vector: treating the dataset as a 70,000 by 784 matrix. The interested reader can find the code for the experiments on MNIST Fashion in Section A.4 with details regarding each step (Methods and MNIST Fashion (784) import(A.4.1, A.4.3), Metrics Design (A.4.2), Methods Design (A.4.4), Metrics computation (A.4.5), Plotting the Embedding (A.4.6), Computing the Clusters (A.4.7), and Plotting the Clusters (A.4.8)).

We again create the embeddings using the methods we found in the simulations to be most proficient at preserving the microscopic, mesoscopic and macroscopic structures. The embeddings are in Figure 3.6. PCA quantitatively performs the worst at preserving the local structure (KNN 0.01). However, it best preserves the mesoscopic and macroscopic structure of the MNIST Fashion dataset (KNC 0.95, CPD 0.88). The poor visualisation offered by PCA (lack of interpretation of cluster separation) demonstrates that the global structure metrics are misleading. They are quantitatively correct, yet we cannot solely refer to them to describe the visualisation offered by PCA, t-SNE and UMAP.

In the t-SNE methods, we find that t-SNE parametrised with a learning rate $\eta = 200$ and Perplexity = 30 best preserves the local structure (KNN 0.31). Interpreting the embedding further in t-SNE: adding Multiscale Similarities (**c**) and a higher learning rate $\eta = n/12$ (**d**)) improves the preservation of the mesoscopic and macroscopic structures. However, it degrades the preservation of the local structure. From method **b** to **c**, the KNN score worsens from 0.31 to 0.26, the KNC score remains the same, and the CPD score improves from 0.65 to 0.69. By using a higher learning rate in t-SNE (**d**) we exacerbate the trade-off between local and global structure preservation further: we do not faithfully preserve the local structure anymore (KNN 0.05). However, we obtain an embedding that more faithfully preserves the

global structure (KNC 0.88 and CPD 0.75).

In the UMAP methods, we find that UMAP (LE) quantitatively performs better than UMAP (random). UMAP (LE) preserves more of the mesoscopic and macroscopic structure than UMAP (random) (KNC 0.85 vs 0.72 and CPD 0.59 vs 0.43) and equally preserves the local structure (KNN 0.11 vs 0.11). UMAP (LE - **e**) performs as well as the best t-SNE method (**c**) in preserving the mesoscopic structure but quantitatively worse on the local and global metric. Visually, UMAP is pleasing as it provides separation between clusters, aiding the visualisation of possible interactions.

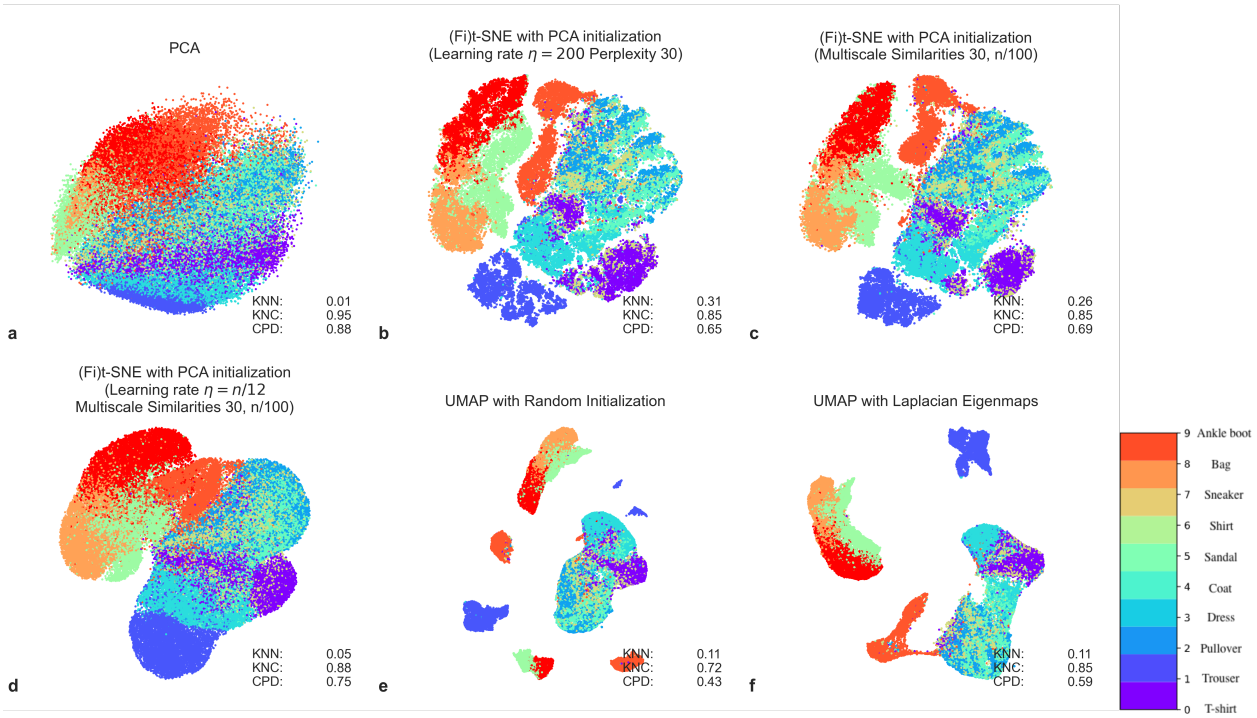


Figure 3.6: Embedding using methods **a**, **c**, **e**, **f**, **g**, and **h** of the MNIST Fashion dataset.

To perform clustering on these embeddings, we use the same process detailed for MNIST Digits: setting up a baseline. In the embedding analysis earlier, we found that method **b** preserved more faithfully the local and global structure of the MNIST Fashion dataset. As such, we use **b** as the initial embedding from which we build the baseline HDBSCAN clusters. On the benchmark, HBDSCAN clustered with high accuracy the data not classified as noise (MI 0.83 and ARI 0.86). However, it considered only 10.94% of the data. On the overall dataset, it performed poorly (MI 0.17 and ARI 0.19). Using the PCA embedding as a starting point, it performed clustering with an ordinary accuracy on the data it was confident (MI 0.53 and ARI 0.36). However, this only accounts for 62.02% of the data. Adding the noise (non-clustered data) to the computation of our two metrics, we arrive at a global score of MI 0.37 and ARI 0.19.

The additional white spaces in the UMAP embedding provide for a preferred embedding for the HBDSCAN algorithm. We focus on UMAP (LE), **g**. HBDSCAN confidently (99.04%) creates 6 clusters leading to a global MI score of 0.64 and an ARI score of 0.4. The ARI score is lower than the MI score because HDBSCAN fails to identify finer clusters. For example, the blue island contains in the original embedding three distinct classes (). The greater separation between clusters that UMAP offers, with the risk of over-interpreting the global structure, aids HBDSCAN in finding more clusters. However, the greater quantity of clusters

does not equate to a better separation in the data. In the middle-bottom of the UMAP (LE) plot, HDBSCAN misclassifies the handbag class into three distinct classes (red, orange and light blue). HDBSCAN misclassifies three classes (boots, sandals, and sneakers) into the light blue blob in the top right. HDBSCAN does discover the class "Trouser" as its standalone cluster. Finally, the separation in the bottom right is unfaithful to the embedding: it does not delimit two separate clusters; it cuts off the turquoise blob in the embedding (coat) into two.

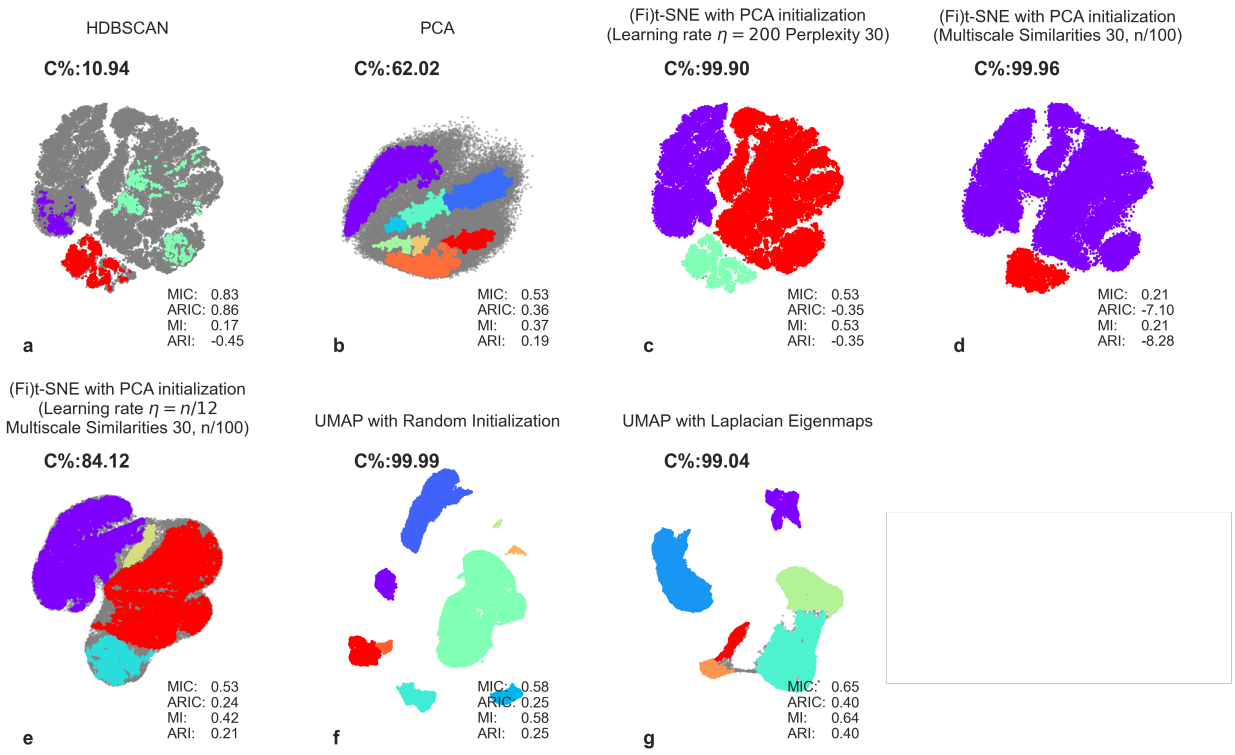


Figure 3.7: Clustering MNIST Fashion using HDBSCAN.

HDBSCAN on the faithful t-SNE embeddings that offers a good trade-off between local and global structure preservation (**c** and **d**) performs poorly. t-SNE generates denser embeddings without additional whitespace between clusters. As such, for method **c**, HDBSCAN confidently (99.90%) creates 3 clusters leading to a global score of MI 0.53 and ARI -0.35. ARI accounts for "chance" by discounting the expected RI from the RI. The adjusted MI score is artificially inflated by the fact that the three "islands" each have clusters of different points, as such by chance HBDSCAN obtains an adequately high MI score. ARI provides us with a better representation of HDBSCAN's behaviour on t-SNE embeddings as it appears to account more for the chance factor. In the t-SNE method **d**, HDBSCAN confidently (99.96%) creates two clusters leading to a global score of MI 0.21 and ARI -8.28. The negative ARI score is indicative of the algorithm's poor performance on this embedding and that HBDSCAN obtains an average MI score by chance for the same reasons as for **c**. In method **e**, the more faithful representation of the global structure appears to make HDBSCAN perform an adequate job at clustering. HBDSCAN confidently (84.12%) creates four clusters, obtaining a MI score of 0.42 and an ARI of 0.21 globally. The lower ARI score provides insight into the big red and purple area, indicating that much information about the finer clusters is lost. The latter t-SNE embedding provides us with more information about the interactions between objects in MNSIT Fashion: we obtain four clear groupings; shoewear

(purple), handbag (yellow), tops (red), and trousers (blue). We cluster less data than using the UMAP (LE) embedding (84.12% vs 99.04%), and as per UMAP (LE), we do not obtain the finer class structures. We delve deeper into the results in Section 4.3.

3.3 Speed

To analyse the difference in speed between t-SNE and UMAP, we use the MNIST digit data set and the MNIST fashion data set. They are traditional benchmarks to evaluate the efficiency of algorithms in manifold learning. We use t-SNE in the OpenTSNE Python package. As discussed in the optimisation section of the t-SNE method, there are two ways to approximate pairwise repulsive interactions: the Barnes-Hut method (Barnes-Hut t-SNE) and the Interpolation method (Fit-SNE). To evaluate if our theoretical hypothesis that the Barnes-Hut method performs best on a smaller dataset we include four variants of t-SNE. Two using the Barnes-Hut (`bh`) method: one with a random initialisation (**b**) and the other with PCA initialisation (**c**). The other two will use the interpolation method (`fft`) on both method (**b**) and (**c**). For the UMAP method, we use two variants: one with random initialisation (**g**) and the other with Laplacian Eigenmaps (LE) (**f**). In the experiment, we constructed samples of size [100, 1000, 2500, 10000, 25000, 70000, 100000] taken from the MNIST digits and MNIST fashion datasets and ran each method five times to obtain average time estimates. The results for the MNIST digits and fashion speed comparison test are in Figure 3.8 and Figure 3.9 respectively. Since the two graphs are very similar, we compare them alongside on-another. We did not analyse PCA as we know it has a small runtime even with large datasets. To ensure that different starting points in the optimisation do not impact the speed experiment, we use the same random seed, 42, for the OpenTSNE and UMAP packages. Finally, it is important to note we conducted all experiments on the same number of cores (8) on a computer with no other applications running to ensure a controlled environment. The interested reader can find the code for the speed benchmark experiment in Section A.5 with details regarding each step (Methods and Datasets import (A.5.1, A.5.3), Experiment design (A.5.2), Methods Design (A.5.4), Experiment run (A.5.5) and Graph Plotting (A.5.6)).

In the small sample size range (0-10,000), we find that both UMAP methods are comparable: attributing the variance between runs to the runtime environment. Both variants of UMAP create an embedding in sub-50 seconds, corroborating our findings in the two simulations. For the four t-SNE methods, we distinguish that methods using Barnes-Hut t-SNE were slower (180s and 200s on 100 and 1,000 samples). The methods using interpolation-based optimisation timed 140s and 150s on 100 and 1,000 samples. In OpenTSNE, there is an initial burden on the runtime performance of the algorithm caused by an early exaggeration phase before the conventional embedding regime. The early exaggeration phase typically runs for 250 iterations with a higher value of exaggeration Belkina et al. (2019) (usually 8-12 times larger), allowing for points to disperse before affining the embedding using the conventional regime (usually 750 iterations with attractive forces restored to their original values).

In the medium range (20,000-70,000), we find that whilst the random initialisation of UMAP's embedding time grows linearly, the LE initialisation of UMAP starts to grow faster. For UMAP (LE), on the MNIST digits dataset, the algorithm runtime grows linearly, whilst on the MNIST fashion dataset, it grows exponentially. The former is the more likely: the nearest neighbour search results in faster growth in runtime for UMAP (LE) versus UMAP (random). For the t-SNE methods, there is now a stark difference between Barnes-Hut t-SNE

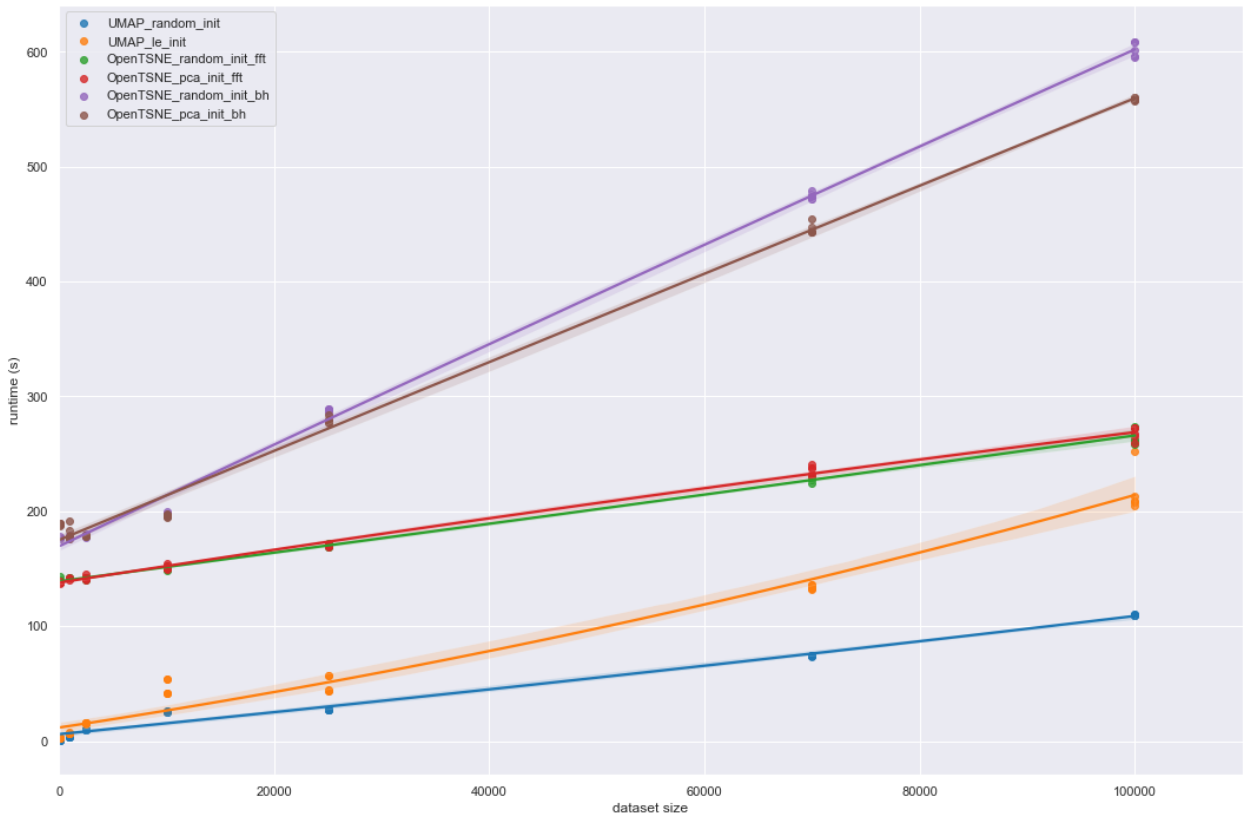


Figure 3.8: MNIST digits speed experiment

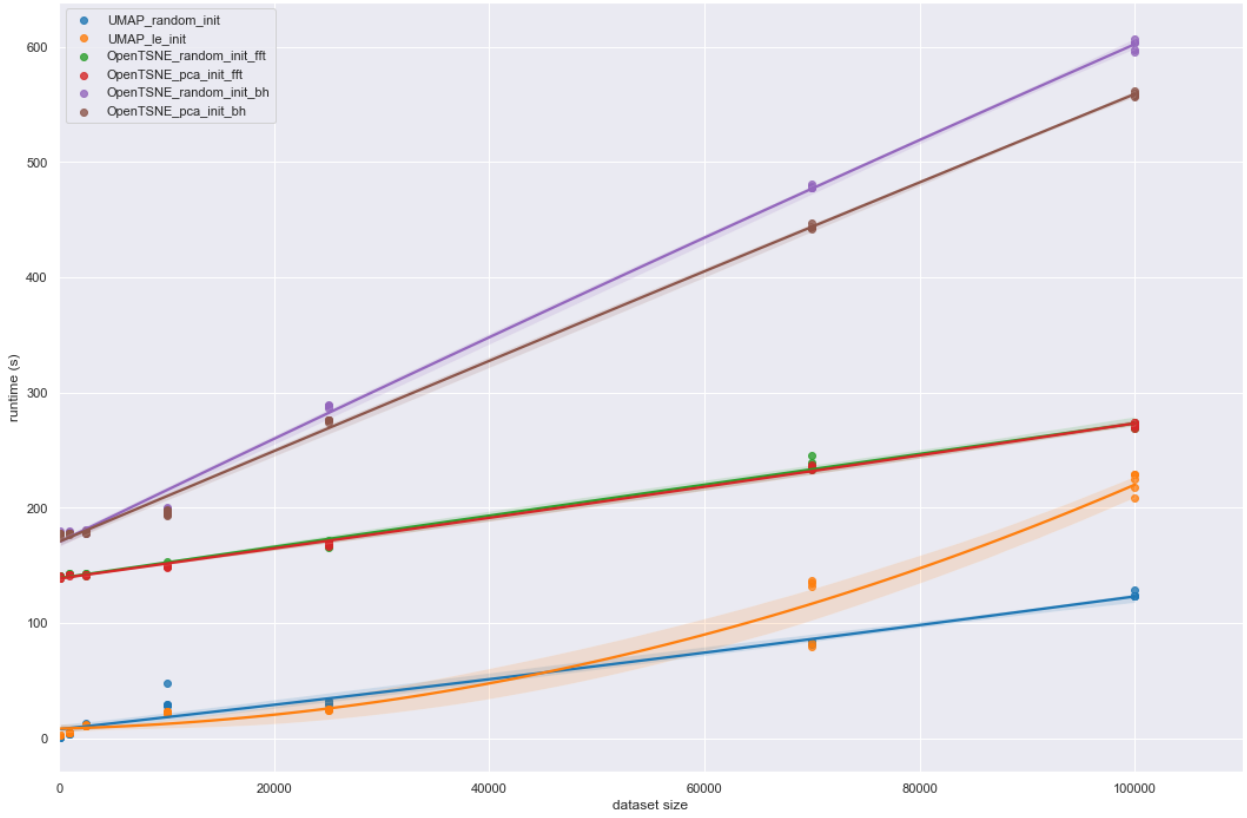


Figure 3.9: MNIST digits speed experiment

and Interpolation t-SNE, with the latter having a much slower growth rate. From our initial hypothesis, we expected Barnes-Hut t-SNE to outperform Fit-SNE on the smaller datasets however, it appears to be a globally slower alternative. On the mid-range, Fit-SNE records times of 170s and 230s on average for samples of size 25,000 and 70,000. In the same range, the best Barnes-Hut t-SNE (PCA) records times of 285s and 540s on average for samples of size 25,000 and 70,000. Fit-SNE's growth rate with both a PCA and a random initialisation are equivalent: it is also lower than UMAP's.

In the large range (100,000), UMAP with LE initialisation and Fit-SNE almost record equal runtime performances. In the MNIST digits speed plot, we can see that one of UMAP's (LE) run produced the same performance as Fit-SNE. We notice that whilst Fit-SNE on all runs appears to record very similar times, UMAP using LE initialisation varies more. UMAP with random initialisation does perform better than all other methods, however, as we saw in our simulations this method does not produce embeddings comparable to those produced by t-SNE with PCA initialisation. We concentrate our attention on the comparison of t-SNE with PCA initialisation (red line) and UMAP with LE initialisation (orange line). We can extrapolate the results to higher sample sizes where we expect UMAP and t-SNE to have comparable runtimes.

Chapter 4

Discussion

4.1 Synthetic Data 1 - Gaussian

In the first analysis of simulated data, see Section 3.1.1, we wanted to expose the reader to the variants of t-SNE and UMAP that are prevalent in literature and provide an intuition of the differences between t-SNE and UMAP. We found through the course of this simulation that PCA most faithfully preserved the mesoscopic and macroscopic structure of the data out of all the methods presented. However, PCA produced an embedding that did not faithfully represent the local structure of the data (KNN 0). Section 3.2 introduced our mathematical justification in more readily focusing on the importance of dimensionality reduction methods in preserving the local structure of high dimensional data. As such, it is paramount to find embeddings that faithfully present the microscopic structure of high-dimensional data over other metrics. We uncovered that t-SNE with Random initialisation and t-SNE with PCA initialisation best preserved the local structure of the simulated data (KNN 0.15). McInnes et al.'s initially claimed that UMAP outperformed t-SNE in preserving the global structure of the data. However, on this data, UMAP with LE initialisation and t-SNE (PCA, multi-scale, high learning rate) produced similar results in their representation of the global structure (CPD 0.81, 0.78 and KNC 0.70 and 0.87) at the detriment of explaining the local structure (KNN 0.02). These results demonstrate that it is possible to manipulate t-SNE to perform equally to UMAP on its global structure claim. Due to the obvious mathematical limitations of preserving the high-dimensional global structure in the euclidean space, it is possible that the latter t-SNE method and UMAP over-interpret the global constitution. Putting our metrics aside, when comparing the resulting embeddings the most striking difference is that UMAP produces denser clusters than t-SNE with additional white space between them. To construct a very similar embedding with t-SNE, we can increase the repulsive forces, spreading dissimilar points and condensing our clusters. In OpenTSNE, this is done by introducing a constant scaling factor and optimizing the pre-existing embedding with k iterations (usually 125). Finally, we discovered that hyper-parameter selection is crucial to produce faithful embeddings.

4.2 Synthetic Data 2 - Circle

The purpose of the second analysis on simulated data, the circle example (Section 3.1.2), was to discuss the importance of initialisation in the UMAP and t-SNE methods. Until now, we have broadly explored the effectiveness of t-SNE and UMAP in preserving the local and

global structure of high-dimensional data to the euclidean space (in this report). One of the main critique addressed in Kobak and Linderman (2021) is the lack of fair analysis between t-SNE and UMAP. The authors state in their paper that much of the literature comparing t-SNE and UMAP do so unfaithfully: authors compare t-SNE with a random initialisation against UMAP with LE initialisation and vice versa, see Becht et al. (2019). In our results in the circle dataset, we saw that these comparisons are senseless. The range of different embeddings on such a simple dataset motivates the need to think carefully about hyper-parameter selection and initialisation. In the last simulation, we saw that a higher learning rate was detrimental to providing a clearer cluster embedding. PCA performed exceedingly well in providing an accurate visualization because the dataset was only two dimensional. As such, its two principal components captured 100% of the variation in the data. In this sample, we discovered that a random initialisation of t-SNE and UMAP portrayed a tangled circle: a very poor visualization of the actual data. Using LE and PCA initialisation injected within UMAP and t-SNE a notion of the global structure thus enabling the visualization of the embedding to be similar to a circle. Within the different t-SNE methods, there is a range of different embeddings depending on the hyper-parameter chosen. We found that in both circle simulations that the multi-scale version of t-SNE provided a better visualization even though it performed worst on the global structure metric than either UMAP (LE) or t-SNE (PCA): CPD 0.79 and 0.77 vs 0.85 and 0.82 vs 0.81 and 0.79. Interestingly, UMAP (LE) that scored the highest CPD produced a confused image: many overlaps and loops. From the two circle simulations, we highlight the importance of using PCA and LE initialisation to better t-SNE and UMAP's faithfulness in presenting the global structure of high-dimensional data. Additionally, we show that t-SNE and UMAP produce different embeddings on very similar data: they are roughly equivalent when measured quantitatively. t-SNE generally performs better in the local metrics, whilst UMAP pulls ahead in the global metrics.

4.3 MNIST Digits & Fashion

The MNIST Digits (see Section 3.2.1) and Fashion (see Section 3.2.2) datasets provide for a more complex environment to compare t-SNE and UMAP. In this report, we group our comparison of the embeddings and clustering for the two datasets within a single section. On both datasets, we discovered that PCA recovers the global structure to a high degree (CPD 0.5 (Digits) 0.88 (Fashion), but as per our simulations, we find that it does not accurately represent the local interactions between local points. It becomes clear that PCA is not appropriate to obtain a faithful representation of high dimensional data's local structure in the euclidean space. Within the realm of t-SNE, we found that our best methods retained a low learning rate of 200. In literature, enhancing t-SNE with a high learning rate of $\eta = n/12$ usually prevents slow convergence and getting stuck at a local optimum. The technique with a higher learning rate appears to provide a more faithful representation of the global structure of the MNIST Digits and Fashion datasets but at the expense of preserving the local interactions. We hypothesize that this may occur as there exists a trade-off between global and local structure optimization. The additional dimensions in high dimensional data cause the volume of space to increase rapidly and for available data to become sparse. It becomes impossible to preserve the global structure of the data. However, t-SNE with a higher learning rate converges rapidly to a non-optimal solution influenced by the PCA initialisation. In both datasets, this t-SNE method looks similar to the PCA embedding but with better preservation of the local structure (KNN 0.03 vs 0.01 (Digits) and 0.05 vs 0.01 (Fashion)). We disregard UMAP with a random initialisation from the embedding comparison as motivated by our circle dataset results and discussion. The other two t-SNE methods are similar.

The PCA initialisation version (non-multi-scale) provides a more faithfully representation of the local structure (KNN 0.35 vs 0.28 (Digits) and 0.31 vs 0.26 (Fashion), whereas the multi-scale version preserves the global structure better (CPD 0.35 vs 0.28 and 0.69 vs 0.65 (Fashion)). Although the two t-SNE methods outperform UMAP (LE) on all metrics (except KNC), they have different embeddings to UMAP. The UMAP (LE) method offers a "clearer" visualization: the clusters are denser and dissimilar points pushed further apart.

Focusing specifically on MNIST Digits, we see six clear groupings. The two prominent clusters, each with three different colours, group the numbers 2,5,8 (light blue, green and orange) and 4,6,9 (salmon, turquoise and red). It is a fair grouping: the numbers 7 and 9 have the most similarity, and depending on how one writes the number 4, it can be hard to discern with the number 9. Notice that the cluster with 9 is the linking factor in that bigger cluster. The other bigger cluster is less interconnected: only a few number shapes likely exist in common. We can conclude that UMAP gives us excellent visualization of the dataset and the relationship between digits. The same interaction exists within t-SNE: at the top of the graph, Figure ..., we have 7,9,4 (with a link between 9 and 4 as per UMAP), and in the middle (5,8,3) with a single common touching point. We can exaggerate the embedding to obtain a clearer visualization.

Similar patterns emerge in their embedding of the MNIST Fashion dataset. In the t-SNE embedding, on the left; boot, sandal, sneaker (red, green, and salmon), in the top-middle; bag (orange), in the top-bottom; trouser (blue), in the bottom-right t-shirt (purple) and the messy cluster; dress, pullover, shirt, coat (turquoise, light blue, yellow, green-turquoise). t-SNE successfully groups the item by category: footwear, accessories, pants, and over-garments. UMAP constructs the same categorization in its embedding. It becomes clear that whilst t-SNE and UMAP differ in the metrics (by a small margin): visually, they produce the same result. They both portray the interaction between items in the dataset proficiently.

Using HDBSCAN with t-SNE and UMAP embeddings is not the principal result of this report. It is an additional study into the effect of UMAP and t-SNE embeddings and how they separate the data. We found that UMAP provides better support for HDBSCAN to achieve a high clustering accuracy. HDBSCAN works by hierarchically clustering by k-nearest neighbours. UMAP produces dense clusters with additional white space to separate them: an ideal ground for HDBSCAN to perform better clustering. However, we found the extra accuracy of using UMAP (LE) over t-SNE on the MNIST dataset to be minute: MI 0.85 vs 0.82 and ARI 0.65 vs 0.71. By running 125 iterations of exaggeration by a factor of 4 (scale the repulsive forces by 4), HDBSCAN on the best t-SNE embedding improves its accuracy in clustering: MI 0.871 vs 0.85 and ARI 0.828 vs 0.71.

On the MNIST Fashion dataset, UMAP (LE) performs better than t-SNE. The link between the greater cluster density and separation and HDBSCAN accuracy is apparent. HDBSCAN with UMAP obtains an MI score of 0.64 vs 0.53 and an ARI score of 0.4 vs -0.35. The tight cluster structure of t-SNE in MNIST Fashion misleads the HDBSCAN algorithm to produce three clusters. By replicating the process of using exaggeration with t-SNE, we found that by exaggerating by a factor of 4 for 125 iterations, HBDSCAN (99.99%) obtains a global MI score of 0.653 and an ARI score of 0.454. The results are slightly better than those obtained using UMAP. However, compared to the high accuracy in the MNIST digits dataset, none of the methods are proficient in MNIST Fashion at constructing an ideal embedding for HBDSCAN's clustering. The embedding's quality limits HDBSCAN's accuracy: HDBSCAN clusters almost identically along the groups built by UMAP and t-SNE.

4.4 Speed

In the speed performance results (see Section 3.3), we showed that t-SNE had an initial performance burden over UMAP, but their performance converged on larger datasets. The various efficiency optimisations for t-SNE over the years, detailed in Section 2.2.1 and 2.2.2, closed the efficiency gap between UMAP and t-SNE on larger datasets. It is important to note that this comes as a burden on smaller datasets. However, the time difference is insignificant (tens of seconds) and does not make for a solid argument for picking UMAP over t-SNE anymore. Additionally, a common approach to speed up t-SNE that has risen in popularity in single-cell analysis that deals with large datasets, with sometimes over 2 million samples, is to model visualization with a subset of random or curated points (van Unen et al. (2017)) from the neighbour nearest graph (Belkina et al. (2019). We then use the smaller embedding to embed the whole sample. This technique can neglect rare data points and hide interactions within the data. For example, in single-cell data analysis, rare data points can provide essential information to determine the effect of cells in the research context. However, in general, this is a very efficient method to produce accurate results. The method relies on extracting core cluster points in the initial embedding and recomputing nearest-neighbour approximation as we add more data. The UMAP Python package does not currently support this operation, but we expect this trick to be transferable.

Chapter 5

Conclusion

In this report, we wanted to objectively verify the original claims of McInnes, Healy and Melville (2018): "[the UMAP]... algorithm is competitive with t-SNE for visualisation quality and arguably preserves more of the global structure with superior runtime performance." This report followed a logical progression to address the main four points of comparison: visualisation quality, preservation of the global and local structure, and runtime performance.

To compare the visualisation performance and UMAP, we used three synthetic data samples, MNIST Digits and MNIST Fashion. The latter two have a human interpretable underlying structure. We found that indeed t-SNE and UMAP had comparable visualisation quality. In the MNIST Digits dataset, the embeddings were visually different but proved to display similar underlying patterns in the data. We, however, found instances where the UMAP embedding was less explicit: t-SNE demonstrated relationships between the digits (5 and 6) and (1 and 9); UMAP's additional white space made these relationships less evident. The similarities were more prominent in the MNIST Fashion dataset, where the sole differentiating factor was UMAP's creation of denser clusters with additional white space separating them. From our datasets, neither UMAP nor t-SNE pulls ahead in providing a visualisation of the high dimensional data. We conclude that they indeed construct comparable visualisation with minute differences about the relationship of the data.

To assess the local and global faithfulness of t-SNE and UMAP embeddings of high dimensional data translated to the euclidean space we used three metrics: KNN (microscopic), KNC (mesoscopic) and CPD (macroscopic). We found in all experiments that t-SNE vastly outperforms UMAP in preserving the local structure of the high dimensional data. Notably, on the MNIST digits dataset, t-SNE's KNN score was three orders of magnitude higher than UMAP's, and on the synthetic dataset, it was seven orders of magnitude higher. In the global structure metrics (CPD and KNC), UMAP performed better than any t-SNE methods on the circle dataset and gaussian synthetic dataset. However, we discovered the difference to be minute. On the MNIST Digits and Fashion dataset, all the t-SNE pipelines performed better quantitatively than UMAP in the KNN and CPD metrics. For the synthetic dataset, we produced a t-SNE method that performed comparably to UMAP on the global metrics: outperforming UMAP on the KNC metric (0.87 vs 0.7) and obtaining a slightly smaller CPD score (0.78 vs 0.81). We conclude that we can select hyper-parameters that produce similar to better results than UMAP in the quantitative metrics that exemplify the local and global structure of the data. It was seen in Becht et al. (2019), the authors found UMAP to provide superior embeddings but used default settings to produce the embedding and failed to construct t-SNE with PCA initialisation. Kobak and Berens (2019) demonstrated on the same

datasets that the proper initialisation and selection of hyper-parameters for t-SNE produce comparable if not better results to t-SNE in scRNA-seq studies. Our study of t-SNE constructs better representations of the local structure of the high-dimensional data. We found that local structure preservation was the only mathematical guarantee. Kobak and Berens (2019) found that UMAP misrepresented some aspects of the global structure on the Cao et al. (2019) datasets. Further analysis into UMAP could confirm that it over-interprets the global structure of the data as we saw in these smaller experiments.

Finally, with the introduction of "Fast interpolation-based t-SNE" or Fit-SNE, we observed that the statement:"[UMAP has]...superior run time performance" is no longer accurate. We discovered that Barnes-Hut t-SNE performs equally or worse than Fit-SNE. The `ftt` option should be favoured when using t-SNE. Comparing Fit-SNE and UMAP, we find that UMAP performs better on small datasets. However, their performance converges on larger datasets (starting at 100k). Better performance on smaller datasets is insignificant when looking at the use cases of these methods: analysis on massive datasets. We conclude that McInnes, Healy and Melville (2018)'s observation that UMAP achieves better run-time performance is no longer the case on significant sample sizes.

In the current state of research, there are no mathematical explanations regarding UMAP's different embedding structure. Future works in Machine Learning have to concentrate on understanding UMAP's creation of denser clusters with additional white space. This could be critical in understanding if UMAP over-interprets the global structure of high-dimensional to the detriment of the local structure. UMAP and t-SNE are only as good as the information they visually provide. Future works in fields such as single-cell data analysis have to continue to compare the use of both methods to analyse which algorithm provides a more faithful representation of high-dimensional data in the Euclidean space. It becomes especially significant as UMAP and t-SNE can extract different structures within the same dataset to explain different biological phenomenons (McInnes, Healy and Melville (2018)).

Finally, UMAP is a much more novel method than t-SNE. Further optimisations UMAP may allow the algorithm to produce better results than t-SNE in the future as it exploits topological sound mathematics. However, because we are limited by the "Curse of Dimensionality" and the restrictions imposed by the intrinsic dimensionality of any particular dataset, it may not be possible to produce significant improvements of current methods. A new method by Ding et al. (2018) named scvis, promises to provide robust and faithful representations of the local and global structure superior to UMAP's and t-SNE's, however, this method is out of the scope of this report. But so far it has attracted less attention than UMAP or t-SNE.

Chapter 6

Bibliography

Barnes, J. and Hut, P. (1986). A hierarchical o (n log n) force-calculation algorithm, *nature* **324**(6096): 446–449.
URL: *https://www.nature.com/articles/324446a0*

Becht, E., Leland McInnes, J. H., Charles-Antoine Dutertre, I. W. H. K., Lai Guan Ng, F. G. and Newell, E. W. (2019). Dimensionality reduction for visualizing single-cell data using UMAP, *Nature Biotechnology* pp. 28–44.
URL: *https://doi.org/10.1038/nbt.4314*

Belkin, M. and Niyogi, P. (2003). Laplacian eigenmaps for dimensionality reduction and data representation, *Neural Computation* **15**(6): 1373–1396.
URL: *https://ieeexplore.ieee.org/document/6789755*

Belkina, A. C., Ciccolella, C. O., Anno, R., Halpert, R., Spidlen, J. and Snyder-Cappione, J. E. (2019). Automated optimized parameters for t-distributed stochastic neighbor embedding improve visualization and allow analysis of large datasets, *bioRxiv* .
URL: *https://www.biorxiv.org/content/early/2019/05/17/451690*

Bernhardsson, E. (2017). Annoy: Approximate nearest neighbors in c++/python optimized for memory usage and loading/saving to disk.
URL: *https://github.com/spotify/annoy*

Birjandtalab, J., Pouyan, M. B. and Nourani, M. (2016). Nonlinear dimension reduction for EEG-based epileptic seizure detection, *2016 IEEE-EMBS International Conference on Biomedical and Health Informatics (BHI)*, pp. 595–598.
URL: *http://doi.org/10.1109/BHI.2016.7455968*

Campello, R. J. G. B., Moulavi, D. and Sander, J. (2013). Density-Based Clustering Based on Hierarchical Density Estimates, *in* J. Pei, V. S. Tseng, L. Cao, H. Motoda and G. Xu (eds), *Advances in Knowledge Discovery and Data Mining*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 160–172.
URL: *https://link.springer.com/chapter/10.1007/978-3-642-37456-2_14*

Cao, J., Spielmann, M., Qiu, X., Huang, X., Ibrahim, D. M., Hill, A. J., Zhang, F., Mundlos, S., Christiansen, L., Steemers, F. J. et al. (2019). The single-cell transcriptional landscape of mammalian organogenesis, *Nature* **566**(7745): 496–502.
URL: *https://www.nature.com/articles/s41586-019-0969-x*

Ding, J., Condon, A. and Shah, S. (2018). Interpretable dimensionality reduction of single cell transcriptome data with deep generative models, *Nature Communications* **9**(2002).
URL: *https://doi.org/10.1038/s41467-018-04368-5*

Dong, W., Moses, C. and Li, K. (2011). Efficient K-Nearest Neighbor Graph Construction for Generic Similarity Measures", *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, Association for Computing Machinery, New York, NY, USA, p. 577–586.
URL: *https://doi.org/10.1145/1963405.1963487*

Eldridge, J., Belkin, M. and Wang, Y. (2015). Beyond hartigan consistency: Merge distortion metric for hierarchical clustering, *Conference on Learning Theory*, PMLR, pp. 588–606.
URL: *http://proceedings.mlr.press/v40/Eldridge15.html*

Hinton, G. E. and Roweis, S. (2003). Stochastic Neighbor Embedding, *in* S. Becker, S. Thrun and K. Obermayer (eds), *Advances in Neural Information Processing Systems*, Vol. 15, MIT Press.
URL: *https://proceedings.neurips.cc/paper/2002/file/6150ccc6069bea6b5716254057a194ef-Paper.pdf*

Jaffe, A., Kluger, Y., Linderman, G. C., Mishne, G. and Steinerberger, S. (2017). Randomized near-neighbor graphs, giant components and applications in data science, *Journal of applied probability* **57**(2): 458–476.
URL: *https://doi.org/10.1017/jpr.2020.21*

Kobak, D. and Berens, P. (2019). The art of using t-SNE for single-cell transcriptomics, *Nature Communications* **10**(5416).
URL: *https://doi.org/10.1038/s41467-019-13056-x*

Kobak, D. and Linderman, G. (2021). Initialization is critical for preserving global data structure in both t-SNE and UMAP, *Nature Biotechnology* **39**: 1–2.
URL: *https://doi.org/10.1038/s41587-020-00809-z*

Kullback, S. and Leibler, R. A. (1951). On Information and Sufficiency, *The Annals of Mathematical Statistics* **22**(1): 79 – 86.
URL: *https://doi.org/10.1214/aoms/1177729694*

Lecun, Y., Bottou, L., Bengio, Y. and Haffner, P. (1998). Gradient-based learning applied to document recognition, *Proceedings of the IEEE* **86**(11): 2278–2324.
URL: *https://doi.org/10.1109/5.726791*

Lee, J. A. and Verleysen, M. (2009). Quality assessment of dimensionality reduction: Rank-based criteria, *Neurocomputing* **72**(7): 1431–1443. Advances in Machine Learning and Computational Intelligence.
URL: *https://www.sciencedirect.com/science/article/pii/S0925231209000101*

Lee, J., Peluffo, D. and Verleysen, M. (2015). Multi-scale similarities in stochastic neighbour embedding: Reducing dimensionality while preserving both local and global structure, *Neurocomputing* **169**.
URL: *https://doi.org/10.1016/j.neucom.2014.12.095*

Leland McInnes, John Healy, S. A. (2017). How HDBSCAN Works.
URL: *https://hdbscan.readthedocs.io/en/latest/how_hdbscan_works.html#how-hdbscan-works*

Linderman, G. C., Rachh, M., Hoskins, J. G., Steinerberger, S. and Kluger, Y. (2017). Fast interpolation-based t-sne for improved visualization of single-cell rna-seq data, *Nature Methods* **16**(3): 243–245.
URL: *http://dx.doi.org/10.1038/s41592-018-0308-4*

McInnes, L. (2018). How UMAP Works.
URL: *https://umap-learn.readthedocs.io/en/latest/how_umap_works.html*

McInnes, L., Healy, J. and Melville, J. (2018). UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction, *arXiv preprint arXiv:1802.03426* .
URL: *https://arxiv.org/abs/1802.03426*

McInnes, L., Healy, J., Saul, N. and Grossberger, L. (2018). Umap: Uniform manifold approximation and projection, *The Journal of Open Source Software* **3**(29): 861.
URL: *https://umap-learn.readthedocs.io/en/latest/index.html*

Policar, P. G., Strazar, M. and Zupan, B. (2019). openTSNE: a modular Python library for t-SNE dimensionality reduction and embedding, *bioRxiv* .
URL: *https://www.biorxiv.org/content/early/2019/08/13/731877*

Poličar, P. (2020). How t-SNE works.
URL: *https://opentsne.readthedocs.io/en/latest/tsne_algorithm.html*

Prim, R. C. (1957). Shortest connection networks and some generalizations, *The Bell System Technical Journal* **36**(6): 1389–1401.
URL: *https://doi.org/10.1002%2Fj.1538-7305.1957.tb01515.x*

scikit-learn developers (2007-2020a). sklearn.metrics.adjusted_mutual_info_score.
URL: *https://scikit-learn.org/stable/modules/generated/sklearn.metrics.adjusted_mutual_info_score.ht*

scikit-learn developers (2007-2020b). sklearn.metrics.adjusted_rand_score.
URL: *https://scikit-learn.org/stable/modules/generated/sklearn.metrics.adjusted_rand_score.html*

van der Maaten, L. (2014). Accelerating t-SNE using Tree-Based Algorithms, *Journal of Machine Learning Research* **15**(93): 3221–3245.
URL: *http://jmlr.org/papers/v15/vandermaaten14a.html*

van der Maaten, L. and Hinton, G. (2008). Visualizing Data using t-SNE, *Journal of Machine Learning Research* **9**(86): 2579–2605.
URL: *http://jmlr.org/papers/v9/vandermaaten08a.html*

van der Maaten, L., Postma, E. and Herik, H. (2007). Dimensionality Reduction: A Comparative Review, *Journal of Machine Learning Research - JMLR* **10**.
URL: *http://www.math.chalmers.se/Stat/Grundutb/GU/MSA220/S18/DimRed2.pdf*

van Unen, V., Thomas Höllt, N. P., Na Li, M. J. T. R., Elmar Eisemann, F. K. and Anna Vilanova, B. P. F. L. (2017). Visual analysis of mass cytometry data by hierarchical stochastic neighbour embedding reveals rare cell types.
URL: *https://doi.org/10.1038/s41467-017-01689-9*

Xiao, H., Rasul, K. and Vollgraf, R. (2017). Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms, *arXiv preprint arXiv:1708.07747* .
URL: *https://arxiv.org/abs/1708.07747*

Appendix A

A.1 Simulation 1 - 15 samples

A.1.1 Imports

[ ]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import tensorflow as tf
import matplotlib

from sklearn.datasets import load_digits
from sklearn.datasets import fetch_openml
import sklearn.datasets as dt
import scipy

from sklearn.decomposition import PCA
from sklearn.neighbors import NearestNeighbors
from scipy.spatial.distance import pdist
from sklearn.utils import resample
from tensorflow.keras.datasets.fashion_mnist import load_data
from umap import UMAP

#Import all the algorithms
from umap import UMAP
import openTSNE
from openTSNE import TSNE as OpenTSNE
from openTSNE import affinity, initialization, TSNEEmbedding
from MulticoreTSNE import MulticoreTSNE

import time
%matplotlib inline
```

A.1.2 Synthetic data creation

Code for the synthetic data used in Section 3.1.1.

```
[ ]: #Simulation data -- https://github.com/berenslab/rna-seq-tsne/blob/
      ↪master/toy-example.ipynb
     #Clusters size
     nn = [2000, 2000, 2000, 2000, 2000,  1000, 1000, 1000, 1000, 1000,  100,␣
      ↪100, 100, 100, 100]
     d1 = [4,   4,   4,   4,   4,    10,  10,  10,  10,  10,   10,  10,  10, ␣
      ↪10,  10]
     d2 = [20,  20,  20,  20,  20,   20,  20,  20,  20,  20,   20,  20,  20, ␣
      ↪20,  20]
     cl = [0,   0,   0,   0,   0,    1,   1,   1,   1,   1,    2,   2,   2,  ␣
      ↪2,   2]

     p = 50   # dimensionality

     np.random.seed(42)
     X = []
     y = []

     #
     for i,n in enumerate(nn):
         Xpart = np.random.randn(n,p)
         Xpart[:, i] += d1[i]
         Xpart[:,20+cl[i]] += d2[i]
         X.append(Xpart)

         ypart = np.ones(n) * i
         y.append(ypart)

     X = np.concatenate(X)
     y = np.concatenate(y)
```

A.1.3 Metrics Design (KNN, KNC & CPD)

Code used to produce the KNN, KNC and CPD score in Section 3.1.1's experiments.

```
[ ]: def embedding_metrics(X, Z, classes, knn=10, knn_classes=10,␣
      ↪subsetsize=1000):
         intersections_knn = 0
         intersections_knc = 0

         ## KNN ##
         #Compute the set of KNN on the true and embedded data
         neighbor_true_data_knn = NearestNeighbors(n_neighbors=knn).fit(X)
         set_true_knn = set(neighbor_true_data_knn.
      ↪kneighbors(return_distance=False))
         neighbor_embed_data_knn = NearestNeighbors(n_neighbors=knn).fit(Z)
         set_embed_knn = set(neighbor_embed_data_knn.
      ↪kneighbors(return_distance=False))
```

```
    #Compute the intersections between true and embedding
    for i in range(X.shape[0]):
        intersections_knn += len(set_true_knn[i] & set_embed_knn[i])
    KNN = intersections_knn / (X.shape[0] * knn)

    ## KNC ##
    #Build the class means
    cl, cl_inv = np.unique(classes, return_inverse=True)
    C = cl.size
    mu1 = np.zeros((C, X.shape[1]))
    mu2 = np.zeros((C, Z.shape[1]))
    for c in range(C):
        mu1[c,:] = np.mean(X.iloc[cl_inv==c,:], axis=0)
        mu2[c,:] = np.mean(Z[cl_inv==c,:], axis=0)

    #KNN on the class means
    neighbor_true_data_knc = NearestNeighbors(n_neighbors=knn_classes).
↪fit(mu1)
    set_true_knc = neighbor_true_data_knc.kneighbors(return_distance=False)
    neighbor_embed_data_knc = NearestNeighbors(n_neighbors=knn_classes).
↪fit(mu2)
    set_embed_knc = neighbor_embed_data_knc.
↪kneighbors(return_distance=False)

    #Compute the intersection between true and embedding
    for i in range(C):
        intersections_knc += len(set(set_true_knc[i]) &␣
↪set(set_embed_knc[i]))
    KNC = intersections_knc / (C * knn_classes)

    ## CPD ##
    #Pick a subset of pairwise points
    subset = np.random.choice(X.shape[0], size=subsetsize, replace=False)
    distance_true = pdist(X[subset,:])
    distance_embed = pdist(Z[subset,:])
    CPD = scipy.stats.spearmanr(distance_true[:,None],
            distance_embed[:,None]).correlation

    return KNN, KNC, CPD
```

A.1.4 Embeddings

Code used to produce the PCA, t-SNE and UMAP embeddings shown in Section 3.1.1's experiments.

[ ]:
```
embed =[]
# Methods
```

```
#PCA - 2 components
%time embed.append(PCA(n_components=2).fit_transform(X))
#t-SNE random initialization
%time embed.append(OpenTSNE(n_jobs=-1, random_state=42,
        initialization='random',negative_gradient_method="bh").fit(X))
#t-SNE PCA initialization
%time embed.append(OpenTSNE(n_jobs=-1, random_state=42,
        negative_gradient_method="bh").fit(X))
#Multi-Scale kernel
%time affinities_multiscale_mix = affinity.Multiscale(X,
        perplexities=[30, int(X.shape[0]/100)], n_jobs=-1, random_state=42)
%time pca_init = initialization.pca(X,
        random_state=42)
%time embedding0 = TSNEEmbedding(pca_init,affinities_multiscale_mix,
        negative_gradient_method="bh",n_jobs=8,random_state=42)
%time embedding1 = embedding0.optimize(n_iter=250, exaggeration=12,
        momentum=0.5,random_state=42)
%time embed.append(embedding1.optimize(n_iter=750, exaggeration=1,
        momentum=0.5, random_state=42))
#Learning Rate adjusted
%time affinities_multiscale_mix = affinity.Multiscale(X,
        perplexities=[30, int(X.shape[0]/100)], n_jobs=-1, random_state=42)
%time pca_init = initialization.pca(X, random_state=42)
%time embedding0 = TSNEEmbedding(pca_init,affinities_multiscale_mix,n/12,
        negative_gradient_method="bh",n_jobs=8,random_state=42)
%time embedding1 = embedding0.optimize(n_iter=250, exaggeration=12,
        momentum=0.5,random_state=42)
%time embed.append(embedding1.optimize(n_iter=750, exaggeration=1,
        momentum=0.5,random_state=42))
#UMAP random initialization
%time embed.append(UMAP(random_state=42, init='random').fit_transform(X))
#UMAP LE initialization
%time embed.append(UMAP(random_state=42).fit_transform(X))
```

A.1.5 Metrics Computation

Code used to compute the KNN, KNC and CPD scores in Section 3.1.1's experiments.

[ ]:
```
%%time
#Calculate KNN and CPD
metrics = []
for Z in embed:
    knn, cpd = embedding_metrics(X, Z, knn=10, subsetsize=1000)
    metrics.append((knn,cpd))
```

A.1.6 Plotting Embeddings

Code used to produce a visualisation of the embeddings produced by PCA, t-SNE and UMAP on the synthetic data shown in Figure 3.1.

```
[ ]: titles = ["PCA", "(Fi)t-SNE with Random Initialization", "(Fi)t-SNE with␣
     ↪PCA initialization\n(Learning rate $\eta=200$ Perplexity 30)",␣
     ↪"(Fi)t-SNE with PCA initialization\n(Learning rate $\eta=n/12$)",␣
     ↪"(Fi)t-SNE with PCA initialization\n(Multiscale Similarities 30, n/
     ↪100)", "(Fi)t-SNE with PCA initialization\n(Learning rate $\eta=n/
     ↪12$\nMultiscale Similarities 30, n/100)", "UMAP with Random␣
     ↪Initialization", "UMAP with Laplacian Eigenmaps"]

     letters = 'abcdefgh'

     plt.figure(figsize=(7.2, 4.5))

     for i,Z in enumerate(embed):
         plt.subplot(2,4,1+i)
         plt.gca().set_aspect('equal', adjustable='datalim')
         #Scatter plots of embeddings
         rand_order = np.random.permutation(Z.shape[0])
         plt.scatter(Z[rand_order,0], Z[rand_order,1], s=1, c=y[rand_order],␣
     ↪cmap='rainbow',
                     rasterized=True, edgecolor='none')
         plt.title(titles[i], va='center')
         #Metrics text
         plt.text(0.70,.02,'KNN:\nKNC:\nCPD:', transform=plt.gca().transAxes,␣
     ↪fontsize=6)
         plt.text(0.87,.02,'{:.2f}\n{:.2f}\n{:.2f}'.format(
             metrics[i][0], metrics[i][1], metrics[i][2]), transform=plt.gca().
     ↪transAxes, fontsize=6)
         plt.text(0, 0, letters[i], transform = plt.gca().transAxes,␣
     ↪fontsize=8, fontweight='bold')
         #Remove numbers on x/y axis
         plt.xticks([])
         plt.yticks([])

     sns.despine(left=True, bottom=True)
     plt.tight_layout()
     plt.savefig('simulation/toy-simulation.png', dpi=150)
```

A.2 Simulation 2 - Circle

A.2.1 Imports

[ ]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import tensorflow as tf
import matplotlib

from sklearn.datasets import load_digits
from sklearn.datasets import fetch_openml
from matplotlib.colors import ListedColormap
import sklearn.datasets as dt
import scipy

from sklearn.decomposition import PCA
from sklearn.neighbors import NearestNeighbors
from scipy.spatial.distance import pdist
from sklearn.utils import resample
from tensorflow.keras.datasets.fashion_mnist import load_data
from umap import UMAP

#Import all the algorithms
from umap import UMAP
import openTSNE
from openTSNE import TSNE as OpenTSNE
from openTSNE import affinity, initialization, TSNEEmbedding
from MulticoreTSNE import MulticoreTSNE

import time
```

A.2.2 Circle data creation

Data used to create the circle: let $n = 25000$ for the circle with 25k dots. See Section 3.1.2.

[ ]:
```
#Circle data -- https://github.com/dkobak/tsne-umap-init/blob/master/
 ↪tsne-umap-circle.ipynb
n = 7000
np.random.seed(42)
X = np.random.randn(n,3) / 1000
X[:,0] += np.cos(np.arange(n)*2*np.pi/n)
X[:,1] += np.sin(np.arange(n)*2*np.pi/n)
```

A.2.3 Metrics Design (KNN & CPD)

To evaluate the circle we only used the KNN and CPD metrics. The code below details this adaptation.

```
[ ]: def embedding_metrics(X, Z, knn=10, subsetsize=1000):
         intersections = 0

         #Compute the set of KNN on the true and embedded data
         neighbor_true_data = NearestNeighbors(n_neighbors=knn).fit(X)
         set_true = set(neighbor_true_data.kneighbors(return_distance=False))
         neighbor_embed_data = NearestNeighbors(n_neighbors=knn).fit(Z)
         set_embed = set(neighbor_embed_data.kneighbors(return_distance=False))

         #Compute the intersections between true and embedding
         for i in range(X.shape[0]):
             intersections += len(set_true[i] & set_embed[i])
         KNN = intersections / (X.shape[0] * knn)

         #Pick a subset of pairwise points
         subset = np.random.choice(X.shape[0], size=subsetsize, replace=False)
         distance_true = pdist(X[subset,:])
         distance_embed = pdist(Z[subset,:])
         CPD = scipy.stats.spearmanr(distance_true[:,None],
                 distance_embed[:,None]).correlation

         return KNN, CPD
```

A.2.4 Embeddings

Code used to initialise the PCA, t-SNE and UMAP methods and create their associated embeddings.

```
[ ]: embed =[]
     # Methods
     #PCA - 2 components
     %time embed.append(PCA(n_components=2).fit_transform(X))
     #t-SNE random initialization
     %time embed.append(OpenTSNE(n_jobs=-1, random_state=42,
             initialization='random',negative_gradient_method="bh").fit(X))
     #t-SNE PCA initialization
     %time embed.append(OpenTSNE(n_jobs=-1, random_state=42,
             negative_gradient_method="bh").fit(X))
     #Multi-Scale kernel
     %time affinities_multiscale_mix = affinity.Multiscale(X,
             perplexities=[30, int(X.shape[0]/100)], n_jobs=-1, random_state=42)
     %time pca_init = initialization.pca(X,
             random_state=42)
     %time embedding0 = TSNEEmbedding(pca_init,affinities_multiscale_mix,
             negative_gradient_method="bh",n_jobs=8,random_state=42)
     %time embedding1 = embedding0.optimize(n_iter=250, exaggeration=12,
             momentum=0.5,random_state=42)
     %time embed.append(embedding1.optimize(n_iter=750, exaggeration=1,
             momentum=0.5, random_state=42))
```

```
#Learning Rate adjusted
%time affinities_multiscale_mix = affinity.Multiscale(X,
        perplexities=[30, int(X.shape[0]/100)], n_jobs=-1, random_state=42)
%time pca_init = initialization.pca(X, random_state=42)
%time embedding0 = TSNEEmbedding(pca_init,affinities_multiscale_mix,n/12,
        negative_gradient_method="bh",n_jobs=8,random_state=42)
%time embedding1 = embedding0.optimize(n_iter=250, exaggeration=12,
        momentum=0.5,random_state=42)
%time embed.append(embedding1.optimize(n_iter=750, exaggeration=1,
        momentum=0.5,random_state=42))
#UMAP random initialization
%time embed.append(UMAP(random_state=42, init='random').fit_transform(X))
#UMAP LE initialization
%time embed.append(UMAP(random_state=42).fit_transform(X))
```

A.2.5 Metrics Computation

Code used to compute the metrics based on the embedding created in A.2.4.

[ ]:
```
%%time
#Calculate KNN and CPD
metrics = []
for Z in embed:
    knn, cpd = embedding_metrics(X, Z, knn=10, subsetsize=1000)
    metrics.append((knn,cpd))

#Insert the true data
embed.insert(0, X)
```

A.2.6 Plotting Embeddings

Code used the plot the circles in Figure 3.2 and 3.3.

[ ]:
```
cmap = ListedColormap(sns.husl_palette(n))

titles = ['Data', "PCA", "(Fi)t-SNE - Random", "(Fi)t-SNE - PCA␣
 ↪\n(Learning rate $\eta=200$,\nPerplexity 30)", "(Fi)t-SNE - PCA␣
 ↪\n(Multiscale Similarities 30, n/100)", "(Fi)t-SNE - PCA\n(Learning␣
 ↪rate $\eta=n/12$\nMultiscale Similarities 30, n/100)", "UMAP - Random",␣
 ↪"UMAP - Laplacian Eigenmaps"]

plt.figure(figsize=(8,5))

for i,Z in enumerate(embed):
    plt.subplot(2,4,i+1)
    plt.gca().set_aspect('equal', adjustable='datalim')
    plt.scatter(Z[:,0], Z[:,1], s=1, c=np.arange(n), cmap=cmap,␣
 ↪edgecolor='none', rasterized=True)
    plt.title(titles[i], va='center')
```

```
    if i > 0:
        plt.text(0.69,.02,'KNN:\nCPD:', transform=plt.gca().transAxes,␣
 ↪fontsize=6)
        plt.text(0.9,.02,'{:.2f}\n{:.2f}'.format(
            metrics[i-1][0], metrics[i-1][1]), transform=plt.gca().
 ↪transAxes, fontsize=6)
    plt.xticks([])
    plt.yticks([])

sns.despine(left=True, bottom=True)
plt.tight_layout()

plt.savefig('tsne-umap-circle-7kfull.png', dpi=300)
```

A.3 MNIST Digits

A.3.1 Packages Import

[ ]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import tensorflow as tf
import matplotlib

from sklearn.datasets import load_digits
from sklearn.datasets import fetch_openml
from sklearn.datasets import load_digits
from sklearn.metrics import adjusted_rand_score, adjusted_mutual_info_score
import sklearn.datasets as dt
import scipy
import hdbscan

from sklearn.decomposition import PCA
from sklearn.neighbors import NearestNeighbors
from scipy.spatial.distance import pdist
from sklearn.utils import resample
from tensorflow.keras.datasets.fashion_mnist import load_data

#Import all the algorithms
from umap import UMAP
import openTSNE
from openTSNE import TSNE as OpenTSNE
from openTSNE import affinity, initialization, TSNEEmbedding

import time
```

A.3.2 Metrics Design (KNN, KNC & CPD)

Code used to produce the KNN, KNC and CPD score in Section 3.2.1's experiments

[ ]:
```
def embedding_metrics(X, Z, classes, knn=10, knn_classes=10,␣
 ↪subsetsize=1000):
    intersections_knn = 0
    intersections_knc = 0

    ## KNN ##
    #Compute the set of KNN on the true and embedded data
    neighbor_true_data_knn = NearestNeighbors(n_neighbors=knn).fit(X)
    set_true_knn = set(neighbor_true_data_knn.
 ↪kneighbors(return_distance=False))
    neighbor_embed_data_knn = NearestNeighbors(n_neighbors=knn).fit(Z)
```

```
    set_embed_knn = set(neighbor_embed_data_knn.
↪kneighbors(return_distance=False))

    #Compute the intersections between true and embedding
    for i in range(X.shape[0]):
        intersections_knn += len(set_true_knn[i] & set_embed_knn[i])
    KNN = intersections_knn / (X.shape[0] * knn)

    ## KNC ##
    #Build the class means
    cl, cl_inv = np.unique(classes, return_inverse=True)
    C = cl.size
    mu1 = np.zeros((C, X.shape[1]))
    mu2 = np.zeros((C, Z.shape[1]))
    for c in range(C):
        mu1[c,:] = np.mean(X.iloc[cl_inv==c,:], axis=0)
        mu2[c,:] = np.mean(Z[cl_inv==c,:], axis=0)

    #KNN on the class means
    neighbor_true_data_knc = NearestNeighbors(n_neighbors=knn_classes).
↪fit(mu1)
    set_true_knc = neighbor_true_data_knc.kneighbors(return_distance=False)
    neighbor_embed_data_knc = NearestNeighbors(n_neighbors=knn_classes).
↪fit(mu2)
    set_embed_knc = neighbor_embed_data_knc.
↪kneighbors(return_distance=False)

    #Compute the intersection between true and embedding
    for i in range(C):
        intersections_knc += len(set(set_true_knc[i]) &␣
↪set(set_embed_knc[i]))
    KNC = intersections_knc / (C * knn_classes)

    ## CPD ##
    #Pick a subset of pairwise points
    subset = np.random.choice(X.shape[0], size=subsetsize, replace=False)
    distance_true = pdist(X[subset,:])
    distance_embed = pdist(Z[subset,:])
    CPD = scipy.stats.spearmanr(distance_true[:,None],
            distance_embed[:,None]).correlation

    return KNN, KNC, CPD
```

A.3.3 MNIST Digits Import and PCA initalisation

Code used to import the MNIST Digits flattened dataset (70,000 x 784 Matrix). PCA Initialisation for the t-SNE embeddings.

```
[ ]: from sklearn.datasets import fetch_openml
     #Import MNIST Digits
     mnist = fetch_openml('mnist_784')
     X = mnist.data
     y = mnist.target.astype('int')
     #PCA
     X_whitened = X - X.mean(axis=0)
     U, s, V = np.linalg.svd(X_whitened, full_matrices=False)
     X784 = np.dot(U, np.diag(s))[:,:784]
     n = X784.shape[0]n = X784.shape[0]
```

A.3.4 Embeddings

Code used to initialise the PCA, t-SNE and UMAP methods and create their associated embeddings.

```
[ ]: embed =[]
     # Methods
     #PCA - 2 components
     %time embed.append(PCA(n_components=2).fit_transform(X))
     #t-SNE random initialization
     %time embed.append(OpenTSNE(n_jobs=-1, random_state=42,
             initialization='random',negative_gradient_method="bh").fit(X784))
     #t-SNE PCA initialization
     %time embed.append(OpenTSNE(n_jobs=-1, random_state=42,
             negative_gradient_method="bh").fit(X784))
     #Multi-Scale kernel
     %time affinities_multiscale_mix = affinity.Multiscale(X784,
             perplexities=[30, int(X784.shape[0]/100)], n_jobs=-1,␣
      ↪random_state=42)
     %time pca_init = initialization.pca(X,
             random_state=42)
     %time embedding0 = TSNEEmbedding(pca_init,affinities_multiscale_mix,
             negative_gradient_method="bh",n_jobs=8,random_state=42)
     %time embedding1 = embedding0.optimize(n_iter=250, exaggeration=12,
             momentum=0.5,random_state=42)
     %time embed.append(embedding1.optimize(n_iter=750, exaggeration=1,
             momentum=0.5, random_state=42))
     #Learning Rate adjusted
     %time affinities_multiscale_mix = affinity.Multiscale(X784,
             perplexities=[30, int(X784.shape[0]/100)], n_jobs=-1,␣
      ↪random_state=42)
     %time pca_init = initialization.pca(X, random_state=42)
     %time embedding0 = TSNEEmbedding(pca_init,affinities_multiscale_mix,n/12,
             negative_gradient_method="bh",n_jobs=8,random_state=42)
     %time embedding1 = embedding0.optimize(n_iter=250, exaggeration=12,
             momentum=0.5,random_state=42)
     %time embed.append(embedding1.optimize(n_iter=750, exaggeration=1,
             momentum=0.5,random_state=42))
```

```
#UMAP random initialization
%time embed.append(UMAP(random_state=42, init='random').fit_transform(X))
#UMAP LE initialization
%time embed.append(UMAP(random_state=42).fit_transform(X))
```

A.3.5 Metrics Computation

Code used to compute the metrics seen in Figure 3.4.

```
[ ]: %%time
#Calculate KNN, KNC, and CPD
metrics = []
for i in range(len(embed)):
    knn, knc, cpd = embedding_metrics(X, embed[i], y, knn=10,␣
 ↪knn_classes=4, subsetsize=1000)
    metrics.append((knn, knc, cpd))
```

A.3.6 Plotting Embeddings

Code used to plot the embeddings from Section A.3.4 resutling in the plots shown in Figure 3.4.

```
[ ]: titles = ["PCA", "(Fi)t-SNE with PCA initialization\n(Learning rate␣
 ↪$\eta=200$ Perplexity 30)",
          "(Fi)t-SNE with PCA initialization\n(Multiscale Similarities 30,␣
 ↪n/100)",
          "(Fi)t-SNE with PCA initialization\n(Learning rate $\eta=n/
 ↪12$\nMultiscale Similarities 30, n/100)", "UMAP with Random␣
 ↪Initialization",
          "UMAP with Laplacian Eigenmaps","e"]

letters = 'abcdefgh'

plt.figure(figsize=(7.2, 4.5))

# Add plots
for i,Z in enumerate(embed):
    plt.subplot(2,3,1+i)
    plt.gca().set_aspect('equal', adjustable='datalim')
    rand_order = np.random.permutation(Z.shape[0])
    plt.scatter(Z[rand_order,0], Z[rand_order,1], s=1, c=y[rand_order],␣
 ↪cmap='rainbow',
                rasterized=True, edgecolor='none')
    plt.title(titles[i], va='center')
    plt.text(0.69,.02,'KNN:\nKNC:\nCPD:', transform=plt.gca().transAxes,␣
 ↪fontsize=6)
    plt.text(0.9,.02,'{:.2f}\n{:.2f}\n{:.2f}'.format(
```

```
        metrics[i][0], metrics[i][1], metrics[i][2]), transform=plt.gca().
↪transAxes, fontsize=6)
    plt.text(0, 0, letters[i], transform = plt.gca().transAxes,␣
↪fontsize=8, fontweight='bold')
    plt.xticks([])
    plt.yticks([])

sns.despine(left=True, bottom=True)
plt.tight_layout()
plt.savefig('digits.png', dpi=600)
```

A.3.7 Clustering with HDBSCAN

Code used to cluster using HDBSCAN. To improve efficiency of computing the baseline we cluster on the first 50 principal components that capture 95%+ of the variance. For the other clusterings, we use the methods in Section A.3.4's embeddings.

[ ]:
```
X_whitened = X - X.mean(axis=0)
U, s, V = np.linalg.svd(X_whitened, full_matrices=False)
X784 = np.dot(U, np.diag(s))[:,:784]
```

[ ]:
```
#Prepare data to cluster
embeddings_to_cluster = [X]
for Z in embed:
    embeddings_to_cluster.append(Z)

clustering_labels = []
clustered_labels = []
rand_score_clustered = []
mi_score_clustered = []
rand_score = []
mi_score = []
total_capture = []

for i in range(len(embeddings_to_cluster)):
    print(i)
    if i == 0:
        clustering_labels.append(hdbscan.
↪HDBSCAN(min_samples=10,min_cluster_size=500).fit_predict(X784))
    else:
        clustering_labels.append(hdbscan.
↪HDBSCAN(min_samples=10,min_cluster_size=500).
↪fit_predict(embeddings_to_cluster[i]))
    clustered = (clustering_labels[i] >= 0)
    clustered_labels.append(clustered)
    #Total Score
    rand_score.append(adjusted_rand_score(y, clustering_labels[i]))
    mi_score.append(adjusted_mutual_info_score(y, clustering_labels[i]))
```

```
    #Scores based on what was actually clustered
    rand_score_clustered.append(adjusted_rand_score(y[clustered],
                                    clustering_labels[i][clustered]))
    mi_score_clustered.append(adjusted_mutual_info_score(y[clustered],
                                    clustering_labels[i][clustered]))
    #Total capture
    total_capture.append(np.sum(clustered) / X.shape[0])

#Add best KNN/KNC/CPD trade off embedding as benchmark for HDBSCAN
embed_for_cluster = [embed[1]]
for Z in embed:
    embed_for_cluster.append(Z)
```

A.3.8 Plotting the Clusters

Code used to plot the clusters in Figure 3.5.

[ ]:
```
titles = ["HDBSCAN", "PCA", "(Fi)t-SNE with PCA initialization\n(Learning␣
 ↪rate $\eta=200$ Perplexity 30)",
          "(Fi)t-SNE with PCA initialization\n(Multiscale Similarities 30,␣
 ↪n/100)",
          "(Fi)t-SNE with PCA initialization\n(Learning rate $\eta=n/
 ↪12$\nMultiscale Similarities 30, n/100)", "UMAP with Random␣
 ↪Initialization",
          "UMAP with Laplacian Eigenmaps"]

letters = 'abcdefgh'

plt.figure(figsize=(7.2, 4.5))

#Plot clustering on top of the embedding
for i,Z in enumerate(embed_for_cluster):
    plt.subplot(2,4,1+i)
    plt.gca().set_aspect('equal', adjustable='datalim')
    #PLot clustering on top of the non-clustered data
    plt.scatter(Z[~clustered_labels[i],0], Z[~clustered_labels[i],1], s=1,␣
 ↪color=(.5,.5,.5), alpha=0.5,
                rasterized=True, edgecolor='none') #not clustered
    plt.scatter(Z[clustered_labels[i],0], Z[clustered_labels[i],1], s=1,␣
 ↪c=clustering_labels[i][clustered_labels[i]], cmap='rainbow',
                rasterized=True, edgecolor='none') #clustered
    plt.title(titles[i], va='center')
    #Add clustering scores
    plt.text(0.69,.02,'MIC:\nARIC:\nMI:\nARI:', transform=plt.gca().
 ↪transAxes, fontsize=6)
    plt.text(0.90,.02,'{:.2f}\n{:.2f}\n{:.2f}\n{:.2f}'.format(
        mi_score_clustered[i], rand_score_clustered[i], mi_score[i],␣
 ↪rand_score[i]),
```

```
        transform=plt.gca().transAxes, fontsize=6)
    #Add capture %
    plt.text(0.1,0.9, 'C%:', transform = plt.gca().transAxes, fontsize=8,␣
 ↪fontweight='bold')
    plt.text(0.28,0.9, '{:.2f}'.format(total_capture[i]*100), transform =␣
 ↪plt.gca().transAxes, fontsize=8, fontweight='bold')
    plt.text(0, 0, letters[i], transform = plt.gca().transAxes,␣
 ↪fontsize=8, fontweight='bold')
    #Remove x/y axis numbers
    plt.xticks([])
    plt.yticks([])
sns.despine(left=True, bottom=True)
plt.tight_layout()
plt.savefig('clustering/report/cluster_digits.png', dpi=600)
```

A.4 MNIST Fashion

A.4.1 Packages Import

[ ]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import tensorflow as tf
import matplotlib
import hdbscan

from sklearn.datasets import load_digits
from sklearn.metrics import adjusted_rand_score, adjusted_mutual_info_score
from sklearn.datasets import load_digits
from sklearn.datasets import fetch_openml
import sklearn.datasets as dt
import scipy

from sklearn.decomposition import PCA
from sklearn.neighbors import NearestNeighbors
from scipy.spatial.distance import pdist
from sklearn.utils import resample
from tensorflow.keras.datasets.fashion_mnist import load_data

#Import all the algorithms
from umap import UMAP
import openTSNE
from openTSNE import TSNE as OpenTSNE
from openTSNE import affinity, initialization, TSNEEmbedding

import time
```

A.4.2 Metrics Design (KNN, KNC & CPD)

Code used to produce the KNN, KNC and CPD score in Section 3.2.2's experiments

[ ]:
```
def embedding_metrics(X, Z, classes, knn=10, knn_classes=10,␣
↪subsetsize=1000):
    intersections_knn = 0
    intersections_knc = 0

    ## KNN ##
    #Compute the set of KNN on the true and embedded data
    neighbor_true_data_knn = NearestNeighbors(n_neighbors=knn).fit(X)
    set_true_knn = set(neighbor_true_data_knn.
↪kneighbors(return_distance=False))
    neighbor_embed_data_knn = NearestNeighbors(n_neighbors=knn).fit(Z)
```

```
    set_embed_knn = set(neighbor_embed_data_knn.
↪kneighbors(return_distance=False))

    #Compute the intersections between true and embedding
    for i in range(X.shape[0]):
        intersections_knn += len(set_true_knn[i] & set_embed_knn[i])
    KNN = intersections_knn / (X.shape[0] * knn)

    ## KNC ##
    #Build the class means
    cl, cl_inv = np.unique(classes, return_inverse=True)
    C = cl.size
    mu1 = np.zeros((C, X.shape[1]))
    mu2 = np.zeros((C, Z.shape[1]))
    for c in range(C):
        mu1[c,:] = np.mean(X.iloc[cl_inv==c,:], axis=0)
        mu2[c,:] = np.mean(Z[cl_inv==c,:], axis=0)

    #KNN on the class means
    neighbor_true_data_knc = NearestNeighbors(n_neighbors=knn_classes).
↪fit(mu1)
    set_true_knc = neighbor_true_data_knc.kneighbors(return_distance=False)
    neighbor_embed_data_knc = NearestNeighbors(n_neighbors=knn_classes).
↪fit(mu2)
    set_embed_knc = neighbor_embed_data_knc.
↪kneighbors(return_distance=False)

    #Compute the intersection between true and embedding
    for i in range(C):
        intersections_knc += len(set(set_true_knc[i]) &␣
↪set(set_embed_knc[i]))
    KNC = intersections_knc / (C * knn_classes)

    ## CPD ##
    #Pick a subset of pairwise points
    subset = np.random.choice(X.shape[0], size=subsetsize, replace=False)
    distance_true = pdist(X[subset,:])
    distance_embed = pdist(Z[subset,:])
    CPD = scipy.stats.spearmanr(distance_true[:,None],
            distance_embed[:,None]).correlation

    return KNN, KNC, CPD
```

A.4.3 MNIST Fashion Import and PCA initalisation

Code used to import the MNIST Fashion flattened dataset (70,000 x 784 Matrix). PCA initialisation for the t-SNE embeddings.

```
[ ]: #Load MNIST Fashion
fashion = fetch_openml('Fashion-MNIST', version=1)
X = fashion.data
y = fashion.target.astype('int')
#PCA
X_whitened = X - X.mean(axis=0)
U, s, V = np.linalg.svd(X_whitened, full_matrices=False)
X784 = np.dot(U, np.diag(s))[:,:784]
n = X784.shape[0]n = X784.shape[0]
```

A.4.4 Embeddings

Code used to initialise the PCA, t-SNE and UMAP methods and create their associated embeddings.

```
[ ]: embed =[]
# Methods
#PCA - 2 components
%time embed.append(PCA(n_components=2).fit_transform(X))
#t-SNE random initialization
%time embed.append(OpenTSNE(n_jobs=-1, random_state=42,
        initialization='random',negative_gradient_method="bh").fit(X784))
#t-SNE PCA initialization
%time embed.append(OpenTSNE(n_jobs=-1, random_state=42,
        negative_gradient_method="bh").fit(X784))
#Multi-Scale kernel
%time affinities_multiscale_mix = affinity.Multiscale(X784,
        perplexities=[30, int(X784.shape[0]/100)], n_jobs=-1,␣
 ↪random_state=42)
%time pca_init = initialization.pca(X,
        random_state=42)
%time embedding0 = TSNEEmbedding(pca_init,affinities_multiscale_mix,
        negative_gradient_method="bh",n_jobs=8,random_state=42)
%time embedding1 = embedding0.optimize(n_iter=250, exaggeration=12,
        momentum=0.5,random_state=42)
%time embed.append(embedding1.optimize(n_iter=750, exaggeration=1,
        momentum=0.5, random_state=42))
#Learning Rate adjusted
%time affinities_multiscale_mix = affinity.Multiscale(X784,
        perplexities=[30, int(X784.shape[0]/100)], n_jobs=-1,␣
 ↪random_state=42)
%time pca_init = initialization.pca(X, random_state=42)
%time embedding0 = TSNEEmbedding(pca_init,affinities_multiscale_mix,n/12,
        negative_gradient_method="bh",n_jobs=8,random_state=42)
%time embedding1 = embedding0.optimize(n_iter=250, exaggeration=12,
        momentum=0.5,random_state=42)
%time embed.append(embedding1.optimize(n_iter=750, exaggeration=1,
        momentum=0.5,random_state=42))
#UMAP random initialization
```

```
%time embed.append(UMAP(random_state=42, init='random').fit_transform(X))
#UMAP LE initialization
%time embed.append(UMAP(random_state=42).fit_transform(X))
```

A.4.5 Metrics Computation

Code used to compute the metrics seen in Figure 3.6.

[ ]:
```
%%time
#Calculate KNN, KNC, and CPD
metrics = []
for i in range(len(embed)):
    knn, knc, cpd = embedding_metrics(X, embed[i], y, knn=10,␣
 ↪knn_classes=4, subsetsize=1000)
    metrics.append((knn, knc, cpd))
```

A.4.6 Plotting Embeddings

Code used to plot the embeddings from Section A.4.4 resutling in the plots shown in Figure 3.6.

[ ]:
```
titles = ["PCA", "(Fi)t-SNE with PCA initialization\n(Learning rate␣
 ↪$\eta=200$ Perplexity 30)",
          "(Fi)t-SNE with PCA initialization\n(Multiscale Similarities 30,␣
 ↪n/100)",
          "(Fi)t-SNE with PCA initialization\n(Learning rate $\eta=n/
 ↪12$\nMultiscale Similarities 30, n/100)", "UMAP with Random␣
 ↪Initialization",
          "UMAP with Laplacian Eigenmaps"]

letters = 'abcdefgh'

plt.figure(figsize=(7.2, 4.5))

# Add plots
for i,Z in enumerate(embed):
    plt.subplot(2,3,1+i)
    plt.gca().set_aspect('equal', adjustable='datalim')
    rand_order = np.random.permutation(Z.shape[0])
    plt.scatter(Z[rand_order,0], Z[rand_order,1], s=1, c=y[rand_order],␣
 ↪cmap='rainbow',
                rasterized=True, edgecolor='none')
    plt.title(titles[i], va='center')
    plt.text(0.69,.02,'KNN:\nKNC:\nCPD:', transform=plt.gca().transAxes,␣
 ↪fontsize=6)
    plt.text(0.90,.02,'{:.2f}\n{:.2f}\n{:.2f}'.format(
        metrics[i][0], metrics[i][1], metrics[i][2]), transform=plt.gca().
 ↪transAxes, fontsize=6)
```

```
    plt.text(0, 0, letters[i], transform = plt.gca().transAxes,␣
 ↪fontsize=8, fontweight='bold')
    plt.xticks([])
    plt.yticks([])

sns.despine(left=True, bottom=True)
plt.tight_layout()
plt.savefig('clustering/mnist-fash-simulation.png', dpi=600)
```

A.4.7 Clustering with HDBSCAN

Code used to cluster using HDBSCAN. To improve efficiency of computing the baseline we cluster on the first 784 principal components that capture 100% of the variance. For the other clusterings, we use the methods in Section A.4.4's embeddings.

[ ]:
```
#Prepare data to cluster
embeddings_to_cluster = [X]
for Z in embed:
    embeddings_to_cluster.append(Z)

clustering_labels = []
clustered_labels = []
rand_score_clustered = []
mi_score_clustered = []
rand_score = []
mi_score = []
total_capture = []

#Cluster based on all embeddings including the raw data
for i in range(len(embeddings_to_cluster)):
    if i == 0:
        clustering_labels.append(hdbscan.HDBSCAN(min_samples=10,
            min_cluster_size=500).fit_predict(X784))
    else:
        clustering_labels.append(hdbscan.HDBSCAN(min_samples=10,
            min_cluster_size=500).fit_predict(embeddings_to_cluster[i]))
    clustered = (clustering_labels[i] >= 0)
    clustered_labels.append(clustered)
    #Total Score
    rand_score.append(adjusted_rand_score(y, clustering_labels[i]))
    mi_score.append(adjusted_mutual_info_score(y, clustering_labels[i]))
    #Scores based on what was actually clustered
    rand_score_clustered.append(adjusted_rand_score(y[clustered],
                                    clustering_labels[i][clustered]))
    mi_score_clustered.append(adjusted_mutual_info_score(y[clustered],
                                    clustering_labels[i][clustered]))
    #Total capture
    total_capture.append(np.sum(clustered) / X.shape[0])
```

```
#Add best KNN/KNC/CPD trade off embedding as benchmark for HDBSCAN
embed_for_cluster = [embed[1]]
for Z in embed:
    embed_for_cluster.append(Z)
```

A.4.8 Plotting the Clusters

Code used to plot the clusters in Figure 3.7.

[ ]:
```
titles = ["HDBSCAN", "PCA", "(Fi)t-SNE with PCA initialization\n(Learning␣
 ↪rate $\eta=200$ Perplexity 30)",
          "(Fi)t-SNE with PCA initialization\n(Multiscale Similarities 30,␣
 ↪n/100)",
          "(Fi)t-SNE with PCA initialization\n(Learning rate $\eta=n/
 ↪12$\nMultiscale Similarities 30, n/100)", "UMAP with Random␣
 ↪Initialization",
          "UMAP with Laplacian Eigenmaps"]

letters = 'abcdefgh'

plt.figure(figsize=(7.2, 4.5))

#Plot clustering on top of the embedding
for i,Z in enumerate(embed_for_cluster):
    plt.subplot(2,4,1+i)
    plt.gca().set_aspect('equal', adjustable='datalim')
    #PLot clustering on top of the non-clustered data
    plt.scatter(Z[~clustered_labels[i],0], Z[~clustered_labels[i],1],
                s=1, color=(.5,.5,.5), alpha=0.5,
                rasterized=True, edgecolor='none') #not clustered
    plt.scatter(Z[clustered_labels[i],0], Z[clustered_labels[i],1], s=1,
                c=clustering_labels[i][clustered_labels[i]],␣
 ↪cmap='rainbow',
                rasterized=True, edgecolor='none') #clustered
    plt.title(titles[i], va='center')
    #Add clustering scores
    plt.text(0.69,.02,'MIC:\nARIC:\nMI:\nARI:', transform=plt.gca().
 ↪transAxes, fontsize=6)
    plt.text(0.90,.02,'{:.2f}\n{:.2f}\n{:.2f}\n{:.2f}'.format(
        mi_score_clustered[i], rand_score_clustered[i], mi_score[i],␣
 ↪rand_score[i]),
        transform=plt.gca().transAxes, fontsize=6)
    #Add capture %
    plt.text(0.1,0.9, 'C%:', transform = plt.gca().transAxes, fontsize=8,␣
 ↪fontweight='bold')
    plt.text(0.28,0.9, '{:.2f}'.format(total_capture[i]*100), transform =␣
 ↪plt.gca().transAxes, fontsize=8, fontweight='bold')
```

```
    plt.text(0, 0, letters[i], transform = plt.gca().transAxes,␣
 ↪fontsize=8, fontweight='bold')
    plt.xticks([])
    plt.yticks([])

sns.despine(left=True, bottom=True)
plt.tight_layout()
plt.savefig('clustering/report/cluster_fash.png', dpi=600)
```

A.5 Speed

A.5.1 Package Import

This code section details the functions and packages imported to run the speed experiment in Section 3.3.

[ ]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import tensorflow as tf

from sklearn.datasets import load_digits
from sklearn.datasets import fetch_openml

from sklearn.decomposition import PCA
from sklearn.utils import resample
from tensorflow.keras.datasets.fashion_mnist import load_data
from umap import UMAP

#Import all the algorithms
from umap import UMAP
from openTSNE import TSNE as OpenTSNE
from tqdm import tqdm

import time
%matplotlib inline
```

A.5.2 Speed Function

This code section details the functions and packages imported to run the speed experiment in Figure 3.8 and 3.9.

[ ]:
```
def speed_experiment(algorithm, algorithm_name, data, sizes=[100, 1000,␣
↪2500, 10000, 25000, 70000, 100000], n_runs=5):
    result = []
    for k in tqdm(range(len(sizes))):
        size = sizes[k]
        for run in range(n_runs):
            subsample = resample(data, n_samples=size)
            start_time = time.time()
            if 'UMAP' in algorithm_name:
                algorithm.fit_transform(subsample)
            else:
                algorithm.fit(subsample)
            elapsed_time = time.time() - start_time
            del subsample
            result.append((size, elapsed_time))
    return pd.DataFrame(result, columns=('dataset size', 'runtime (s)'))
```

A.5.3 MNIST Digits & Fashion Import

This code section details how we import and initialize the MNIST Digits & Fashion data with PCA.

```
[ ]: digits = fetch_openml('mnist_784')
X_d = digits.data
fashion = fetch_openml('Fashion-MNIST', version=1)
X_f = fashion.data
```

```
[ ]: x_mean_rem_d = X_d - X_d.mean(axis=0)
U, lambd, V = np.linalg.svd(x_mean_rem_d, full_matrices=False)
X784_d = np.dot(U, np.diag(lambd))[:,:784]
x_mean_rem_f = X_f - X_f.mean(axis=0)
U, lambd, V = np.linalg.svd(x_mean_rem_f, full_matrices=False)
X784_f = np.dot(U, np.diag(lambd))[:,:784]
```

A.5.4 Methods

This code section details how we initialised the four t-SNE algorithms: Barnes-Hut t-SNE with and without PCA initialisation and Fit-SNE with and without PCA initialisation. It also details how we initialised the two UMAP algorithms: UMAP with random initialisation and with Laplacian Eigenmaps. See Figure 3.8 and 3.9.

```
[ ]: methods = [UMAP(init='random',random_state=42),
             UMAP(random_state=42),
             OpenTSNE(n_jobs=-1, initialization='random',␣
 ↪random_state=42,negative_gradient_method='fft'),
             OpenTSNE(n_jobs=-1,␣
 ↪negative_gradient_method='fft',random_state=42),
             OpenTSNE(n_jobs=-1, initialization='random',␣
 ↪negative_gradient_method='bh',random_state=42),
             OpenTSNE(n_jobs=-1,␣
 ↪negative_gradient_method='bh',random_state=42)]
```

A.5.5 Speed Experiment Run

This code section details how we ran the speed experiments and collected the results, see Figure 3.8 and 3.9.

```
[ ]: performance_data_d = {}
performance_data_f = {}

for i in range(len(methods_n)):
    algo = methods_n[i]
    if i == 0:
        alg_name = 'UMAP_random_init'
    elif i == 1:
        alg_name = 'UMAP_le_init'
    elif i == 2:
```

```
        alg_name = 'OpenTSNE_random_init_fft'
    elif i == 3:
        alg_name = 'OpenTSNE_pca_init_fft'
    elif i == 4:
        alg_name = 'OpenTSNE_random_init_bh'
    else:
        alg_name = 'OpenTSNE_pca_init_bh'
    performance_data_d[alg_name] = speed_experiment(algo, alg_name, X784_d)
    performance_data_f[alg_name] = speed_experiment(algo, alg_name, X784_f)

    print(f"[{time.asctime(time.localtime())}] Completed {alg_name}")
```

A.5.6 Graph Plotting

This code section details how we plotted the performance data on the MNIST digits dataset, see Figure 3.8.

[ ]:
```
for alg_name, perf_data in performance_data_d.items():
    algo = methods[i]
    sns.regplot('dataset size', 'runtime (s)', perf_data, order=2,␣
 ↪label=alg_name)
plt.legend()
plt.xlim(0, 110000)
```

This code section details how we plotted the performance data on the MNIST fashion dataset, see Figure 3.9.

[ ]:
```
for alg_name, perf_data in performance_data_f.items():
    algo = methods[i]
    sns.regplot('dataset size', 'runtime (s)', perf_data, order=2,␣
 ↪label=alg_name)
plt.legend()
plt.xlim(0, 110000)
```