






openTSNE: A Modular Python Library for t-SNE Dimensionality Reduction and Embedding

Pavlin G. Poličar 
University of Ljubljana

Martin Stražar 
Broad Institute

Blaž Zupan 
University of Ljubljana

Abstract

One of the most popular techniques for visualizing large, high-dimensional data sets is t -distributed stochastic neighbor embedding (t-SNE). Recently, several extensions have been proposed to address scalability issues and the quality of the resulting visualizations. We introduce **openTSNE**, a modular Python library that implements the core t-SNE algorithm and its many extensions. The library is faster than existing implementations and can compute projections of data sets containing millions of data points in minutes.

Keywords: t-SNE, embedding, visualization, dimensionality reduction, Python.

1. Introduction

The ever-growing volumes of high-dimensional data sets in machine learning call for efficient dimensionality reduction techniques and their implementations to produce informative data visualizations. Popular approaches include principal component analysis (PCA), multidimensional scaling, t -distributed stochastic neighbor embedding (t-SNE, [Van Der Maaten and Hinton 2008](#)), and uniform manifold approximation and projections (UMAP, [McInnes, Healy, and Melville 2018](#)). Among these, t-SNE has achieved widespread adoption as it can address high volumes of data and reveal the underlying data manifold. For instance, t-SNE is widely used in the bioinformatics community in areas such as single-cell transcriptomics ([Macosko *et al.* 2015](#); [Cao *et al.* 2019](#); [Tasic *et al.* 2018](#)), human genetics ([Hirata *et al.* 2019](#)), metagenomic assembly ([Beaulaurier *et al.* 2018](#)), spatial organization of microbial communities ([Sheth, Li, Jiang, Sims, Leong, and Wang 2019](#)), and metabolomics ([Tkachev *et al.* 2019](#)). To visually explore the data manifold, the objects of interest, such as diseased and healthy tissues or single cells, are profiled through thousands of features that include, for example, the expression of genes or the concentration of metabolites. The objective of t-SNE is to embed data points within a low-dimensional space, where the preservation of distances is rewarded in a non-

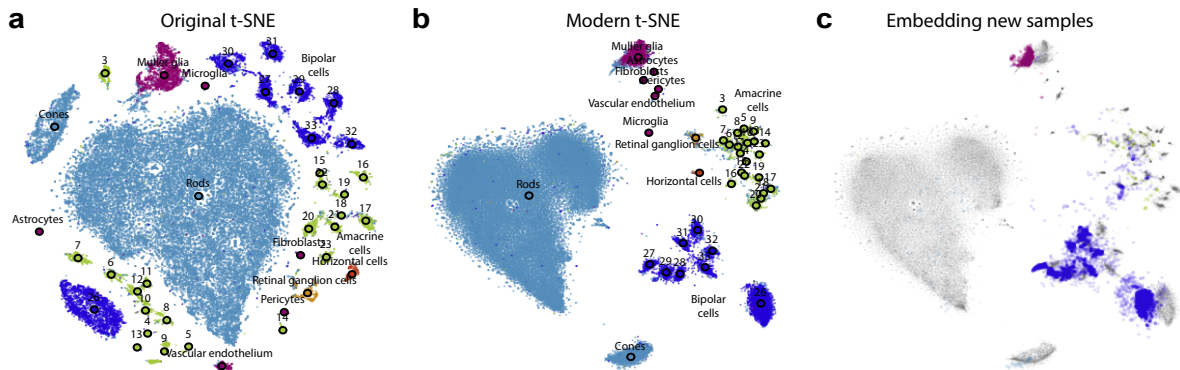


Figure 1: We use **openTSNE** to generate three t-SNE embeddings and demonstrate recent theoretical advances. The data in (a) and (b) include 44,808 mouse retinal cells that are described with high-dimensional gene-expression profiles from [Macosko *et al.* \(2015\)](#). The data in (c) additionally contains 27,499 expression profiles from mouse retinal cells from [Shekhar *et al.* \(2016\)](#). (a) We construct t-SNE embedding following the parameter choices from the original publication by [Van Der Maaten and Hinton \(2008\)](#). The visualization shows no preservation of the global organization of clusters, resulting from random initialization and an affinity model focused on preserving local neighborhoods. (b) A modern t-SNE embedding, utilizing the latest theoretical advances and practical recommendations constructed using a multi-scale affinity model, preserving both short-range and long-range interactions between data points and initialized so that the global layout is as meaningful as possible. Unlike in (a), the green and blue clusters representing different sub-types of amacrine and bipolar cells are now localized to the same regions of the space, indicating a higher level of similarity than to other cell types. The embedding in (c) shows how existing t-SNE reference atlases can be used to place new samples into existing embeddings. The positions of new data points correspond to cell types from the reference atlas.

linear fashion, thus resulting in a compact grouping of highly similar data points. Reports on single-cell gene expression data, where increasingly specific cell types are distinguished by a decreasing number of genes, often rely on t-SNE to embed high-dimensional expression profiles into a two-dimensional space and discover constituent cell types (Figure 1a).

Despite its utility, t-SNE has often been criticized for its limited scalability, lack of global organization – t-SNE identifies well-defined clusters that may be arbitrarily scattered throughout the embedding – and the absence of theoretically-founded methods to map new data into existing embeddings ([Ding, Condon, and Shah 2018](#); [Becht *et al.* 2019](#)). Most of these shortcomings have recently been addressed. [Linderman, Rachh, Hoskins, Steinerberger, and Kluger \(2019\)](#) developed FIt-SNE, an efficient approximation that massively improves the scalability of t-SNE, achieving linear time complexity in the number of samples. [Kobak and Berens \(2019\)](#) proposed several techniques to improve global cluster coherence (Figure 1b), including estimating similarities with a mixture of Gaussian kernels. In our previous work, we introduced a principled approach for embedding new samples into existing visualizations (Figure 1c and [Poličar, Stražar, and Zupan 2023](#)).

Easy access to efficient implementations almost universally correlates with the widespread adoption of novel computational techniques. Despite the many recent theoretical advance-

ments, popular t-SNE libraries have been slow to incorporate them into their implementations. In Python, the most commonly used implementation of the t-SNE algorithm comes from **scikit-learn** (Pedregosa *et al.* 2011), which is tightly integrated with the Python data science environment but suffers from poor scalability. Other implementations from the R (R Core Team 2024; Krijthe 2023) and Julia (Bezanson, Edelman, Karpinski, and Shah 2017; Jonsson 2021) programming languages also scale poorly (see Section 5.2). Another Python implementation, **MulticoreTSNE** (Ulyanov 2019), enjoys better scalability but is likewise limited when applied to larger data sets. Additionally, none of these implementations include recently proposed extensions that can result in more globally consistent embeddings. **FIt-SNE** (Linderman *et al.* 2019) alleviates these problems and provides bindings to both the Python and R programming languages. However, none of the existing t-SNE implementations support adding new samples into existing embeddings.

To alleviate all these problems, and to support newly proposed additions to the t-SNE algorithm, we here present **openTSNE**, an open-source Python implementation of the t-SNE algorithm. **openTSNE** is easy to install and includes precompiled binaries available through popular Python package managers. It provides a familiar API, making it suitable as a drop-in replacement for existing t-SNE implementations. **openTSNE** is distributed under the BSD-3-Clause License and its source code is publicly available at <https://github.com/pavlin-pollicar/openTSNE>. **openTSNE** can also be installed from PyPI and conda-forge. End-to-end usage examples, documentation, and API reference are all available at <https://opentsne.readthedocs.io/>.

2. Methods

We first introduce the relevant notation and briefly review the core t-SNE algorithm in Section 2.1. We then address the criticisms of scalability (Section 2.2), the ability to add new samples to existing embeddings (Section 2.3), and improvements that improve the global consistency of resulting embeddings (Section 2.4).

2.1. *t*-distributed stochastic neighbor embedding

t-distributed stochastic neighbor embedding (t-SNE) is a non-linear dimensionality reduction method that aims to find a low-dimensional embedding where neighborhoods are preserved. Given a multi-dimensional data set with N data points $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\} \in \mathbb{R}^D$, t-SNE aims to find a low dimensional embedding $\mathbf{Y} = \{\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_N\} \in \mathbb{R}^d$ where $d \ll D$, such that if points \mathbf{x}_i and \mathbf{x}_j are close in the high-dimensional space, their corresponding embeddings \mathbf{y}_i and \mathbf{y}_j are also close. Since t-SNE is primarily used as a visualization tool, d is typically set to two. The similarity between two data points in the high-dimensional space is defined using the Gaussian kernel

$$p_{j|i} = \frac{\exp\left(-\frac{1}{2}\mathcal{D}(\mathbf{x}_i, \mathbf{x}_j)/\sigma_i^2\right)}{\sum_{k \neq i} \exp\left(-\frac{1}{2}\mathcal{D}(\mathbf{x}_i, \mathbf{x}_k)/\sigma_i^2\right)}, \quad p_{i|i} = 0 \quad (1)$$

where \mathcal{D} is some distance measure. The values $p_{j|i}$ are then symmetrized to

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N}. \quad (2)$$

The bandwidth σ_i of each Gaussian kernel is selected such that the perplexity of the distribution matches a user-specified parameter value

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

where $H(P_i)$ is the Shannon entropy of P_i ,

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

This enables t-SNE to adapt to the varying density of the data in the multi-dimensional space. The perplexity can be interpreted as the continuous analogue to the number of nearest neighbors to which the distances will be preserved.

The similarity between points \mathbf{y}_i and \mathbf{y}_j in the embedding space is defined using the t -distribution with a single degree of freedom (Cauchy kernel)

$$q_{ij} = \frac{(1 + \|\mathbf{y}_i - \mathbf{y}_j\|^2)^{-1}}{\sum_{k \neq l} (1 + \|\mathbf{y}_k - \mathbf{y}_l\|^2)^{-1}}, \quad q_{ii} = 0. \quad (3)$$

We use the Kullback-Leibler (KL) divergence to measure the agreement between the two distributions \mathbf{P} and \mathbf{Q}

$$C = \text{KL}(\mathbf{P} \parallel \mathbf{Q}) = \sum_{ij} p_{ij} \log \frac{p_{ij}}{q_{ij}}. \quad (4)$$

The objective is to find embeddings \mathbf{Y} that minimize the KL divergence. The corresponding gradient takes the form

$$\frac{\partial C}{\partial \mathbf{y}_i} = 4 \sum_{j \neq i} (p_{ij} - q_{ij}) (\mathbf{y}_i - \mathbf{y}_j) w_{ij}, \quad (5)$$

where $w_{ij} = (1 + \|\mathbf{y}_i - \mathbf{y}_j\|^2)^{-1}$ and represents the unnormalized q_{ij} . We can rewrite this gradient as

$$\frac{\partial C}{\partial \mathbf{y}_i} = 4 \left[\underbrace{\sum_{j \neq i} p_{ij} q_{ij} Z (\mathbf{y}_i - \mathbf{y}_j)}_{\text{attractive forces}} - \underbrace{\sum_{j \neq i} q_{ij}^2 Z (\mathbf{y}_i - \mathbf{y}_j)}_{\text{repulsive forces}} \right], \quad (6)$$

where $Z = \sum_{k \neq l} (1 + \|\mathbf{y}_k - \mathbf{y}_l\|^2)^{-1}$. Under this formulation, we can cast t-SNE algorithm as a force-directed layout algorithm, where individual data points act as particles exerting both attractive and repulsive on each other until a state of equilibrium is reached.

Optimization is performed using batch gradient descent using the delta-bar-delta update rule (Jacobs 1988). The t-SNE optimization procedure consists of two phases: in the first *early exaggeration* phase, the attractive forces between data points are increased by some *exaggeration* factor ρ , typically set to 12, so that points in the embedding can more easily move throughout the space and settle near their respective neighbors. In the second phase of the optimization, the attractive forces are then reverted to their original values with $\rho = 1$.

Belkina, Ciccolella, Anno, Halpert, Spidlen, and Snyder-Cappione (2019) later found that convergence can be sped up by increasing the learning rate η from the standard $\eta = 200$ to $\eta = N/\rho$, where ρ here refers to the exaggeration factor used during the early exaggeration

phase. As a side-effect, embeddings converge faster, and the number of iterations can be lowered from the typical 1000 to 750, decreasing the overall runtime.

2.2. Efficient approximation schemes

A direct evaluation of the t-SNE gradients requires $\mathcal{O}(N^2)$ operations, which makes it impractical with large data sets and calls for efficient approximation schemes. The formulation in Equation 6 casts the t-SNE gradient as an N-body simulation where data points act as particles exerting attractive and repulsive forces onto one another. Both terms lend themselves to efficient approximations, enabling us to reduce the time complexity of the t-SNE algorithm to $\mathcal{O}(N)$. This allows modern t-SNE implementations to be leveraged for the visualization of data sets containing up to millions of data points.

Attractive forces

Van Der Maaten (2014) observed that evaluating the attractive forces between all pairs of data points is excessive. In practice, it suffices to approximate the attraction forces against a small number of k nearest neighbors. Using tree-based nearest-neighbor search methods, Van Der Maaten (2014) reduced the time complexity to $\mathcal{O}(N \log N)$. Linderman *et al.* (2019) noted that embeddings are visually indistinguishable when using merely *approximate* nearest neighbors, further reducing time complexity to $\mathcal{O}(N)$.

Repulsive forces

Similarly, the repulsive term can also be approximated, motivated by methods from particle simulations. Van Der Maaten (2014) proposed an approach based on N-body simulations and used a space-partitioning Barnes-Hut tree approach to approximate the interaction between data points. This reduces the time complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$. More recently, Linderman *et al.* (2019) proposed an alternative approach, FIt-SNE, based on non-uniform convolutions and interpolation, which further reduces the time complexity to $\mathcal{O}(N)$.

2.3. Embedding new samples

The t-SNE algorithm is non-parametric and does not define an explicit mapping from the high-dimensional space to the embedding space. Embeddings of new data points need to be found through optimization schemes (Poličar *et al.* 2023). When adding new data points to an existing, reference embedding, the reference data points are fixed in place while new data points are allowed to find their respective positions. The optimization remains the same as in standard t-SNE with only slight modifications to p_{ij} and q_{ij}

$$p_{j|i} = \frac{\exp\left(-\frac{1}{2}\mathcal{D}(\mathbf{x}_i, \mathbf{v}_j)/\sigma_i^2\right)}{\sum_i \exp\left(-\frac{1}{2}\mathcal{D}(\mathbf{x}_i, \mathbf{v}_j)/\sigma_i^2\right)}, \quad q_{j|i} = \frac{(1 + \|\mathbf{y}_i - \mathbf{w}_j\|^2)^{-1}}{\sum_i (1 + \|\mathbf{y}_i - \mathbf{w}_j\|^2)^{-1}},$$

where $\mathbf{V} = \{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_M\} \in \mathbb{R}^D$ where M is the number of samples in the new data set and $\mathbf{W} = \{\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_M\} \in \mathbb{R}^d$. Additionally, we omit the symmetrization step in Equation 2. Plugging these terms into Equation 4, we obtain the following gradient

$$\frac{\partial \mathcal{C}}{\partial \mathbf{w}_j} = 2 \sum_i \left(p_{j|i} - q_{j|i}\right) (\mathbf{y}_i - \mathbf{w}_j) \left(1 + \|\mathbf{y}_i - \mathbf{w}_j\|^2\right)^{-1}.$$

Similarly to standard t-SNE, a direct calculation of gradients takes $\mathcal{O}(N \cdot M)$ time, but it is straightforward to adapt the Barnes-Hut and FIT-SNE approximation schemes, reducing the time complexity to $\mathcal{O}(M \log N)$ and $\mathcal{O}(M)$, respectively. Special care must be taken to adjust the learning rate during optimization as the parameter values used during standard optimization may lead to unstable behaviour. We have observed that reducing the learning rate to $\eta \sim 0.1$ results in more stable optimization, as discussed in [Poličar *et al.* \(2023\)](#). Standard optimization positions data points in the embedding to achieve an equilibrium between the attractive and repulsive forces. However, when embedding new data points, the reference embedding remains fixed. Consequently, if a new data point needs to be positioned among existing reference data points, it may not be possible to reach such an equilibrium, as the reference data points cannot adjust to accommodate the new data point. To compensate for the increased repulsive forces, it is beneficial to increase the exaggeration factor to $\rho = 1.5$ during optimization.

2.4. Improving the global structure of embeddings

One of t-SNE’s long-standing criticisms is that it fails to preserve long-range distances but instead focuses on capturing the local structure of the data manifold ([Becht *et al.* 2019](#)). Recently, several approaches have been proposed to improve the global organization of the resulting embeddings.

[Kobak and Linderman \(2021\)](#) showed that the initialization of force-directed layout algorithms, such as t-SNE, UMAP ([McInnes *et al.* 2018](#)), and ForceAtlas2 ([Jacomy, Venturini, Heymann, and Bastian 2014](#)), largely dictates the global consistency of resulting embeddings. When initialized randomly, clusters are often arbitrarily scattered in the embedding space. However, when using initialization schemes based on PCA or Laplacian Eigenmaps ([Belkin and Niyogi 2002](#)), the clusters identified in the resulting embedding are typically grouped in a globally coherent manner.

In standard t-SNE, distances between data points are converted to similarities through the use of Gaussian kernels. The perplexity of these kernels governs the sizes of the local neighborhoods which are to be preserved. One easy way to improve the global consistency of resulting embeddings is to increase the sizes of these neighborhoods. This can be achieved by increasing the perplexity parameter. However, increasing perplexity often leads to loss in local structure and tends to obscure small clusters. [Kobak and Berens \(2019\)](#) instead propose to use mixtures of Gaussians to preserve better short-range and long-range distances and, thus, achieve a better trade-off between local and global structure.

Embeddings produced by standard t-SNE often use all available space and separate clusters by only thin boundaries. When working with large data sets, this often obscures the global relationships between clusters as all neighboring groups appear equidistant from one another. Other dimensionality reduction methods, such as UMAP and ForceAtlas2, produce embeddings where clusters appear more compact, and the white space separating the clusters may be interpreted as a loose measure of distance. Recently, [Böhm, Berens, and Kobak \(2022\)](#) showed that the exaggeration factor ρ could be used to produce layouts more similar to UMAP and ForceAtlas2. By incorporating exaggeration into later phases of the optimization, t-SNE introduces white space between clusters, which may better reflect the global relations between clusters.

Standard t-SNE reveals the clustering structure at a single level of resolution. While the perplexity parameter can be used to control the trade-off between local and global structure,

this can be time-consuming, and small, well-defined clusters can be missed. Alternatively, Kobak, Linderman, Steinerberger, Kluger, and Berens (2019) suggest that varying the degrees of freedom in the t -distribution can be used to explore the clustering structure at different levels of resolution. Modifying Equation 3 to $q_{ij} \propto (1 + \|\mathbf{y}_i - \mathbf{y}_j\|^2/\alpha)^{-\alpha}$ allows us to use heavier-tailed distributions to model distances in the embedding space, which acts to highlight small subgroups in the resulting visualization.

3. Implementation

We introduce **openTSNE**, a comprehensive Python library that implements the t-SNE algorithm and its numerous recently proposed extensions. **openTSNE** includes a simple API that allows users to quickly obtain informative t-SNE visualizations and a more flexible API that allows seasoned practitioners to modify all the different aspects of the t-SNE algorithm.

The core t-SNE algorithm comprises three main steps, mirrored in the structure of the **openTSNE** library:

1. The t-SNE algorithm requires the specification of an affinity model to describe the similarities between data points p_{ij} . Standard t-SNE uses perplexity-based Gaussian kernels with varying bandwidths as specified in Equation 1. However, other affinity models can also be used to highlight different aspects of the data or visualize other data modalities. For instance, Kobak and Berens (2019) showed that using mixtures of Gaussians often better reveals global cluster organization in the resulting visualization (albeit at some computational cost). Alternatively, a uniform affinity model can produce quantitatively similar embeddings to standard t-SNE at a third of the computational cost, making it particularly useful for larger data sets. Unlike the standard affinity model, which uses Gaussian kernels to determine the similarities p_{ij} , the uniform affinity model uses a uniform kernel to assign equal similarities to a pre-specified number of nearest neighbors. Visualizing different data modalities, such as graphs, can also be achieved by providing precomputed distance matrices that might consist of, for instance, the shortest paths between nodes. The `openTSNE.affinity` module provides a selection of commonly used affinity models but defaults to the standard perplexity-based Gaussian model, as proposed in the original publication. Users can also define custom affinity models by extending the `openTSNE.affinity.Affinities` class. The resulting class can then be used as a drop-in replacement for the built-in affinity models.

A key component of the affinity model construction process is finding a specified number of nearest neighbors for each data point. **openTSNE** provides several backends for finding the nearest neighbors, each with its own advantages and drawbacks. We use the implementation from **scikit-learn** (Pedregosa *et al.* 2011) for the exact nearest neighbor search. While asymptotically slower than approximate nearest-neighbor search methods, exact approaches introduce minimal computation overhead and tend to run faster for smaller data sets. The **annoy** (Bernhardsson 2023) library offers a fast and efficient implementation of random-projection trees and supports several commonly used metrics but lacks support for sparse matrices. The **pynndescent** (McInnes 2024) library supports sparse matrices and allows users to define and use custom metrics. Finally, we include the **hnsplib** (Malkov and Yashunin 2018) library as it provides a third alternative algorithm for approximate nearest neighbor search, which may perform better on specific

data sets. By default, **openTSNE** automatically selects the most appropriate algorithm for a given data set based on data set size, sparsity, and metric support. These can be overridden by the user and easily extended to support custom nearest-neighbor search solutions.

2. The optimization in t-SNE starts with initial positions for each data point in the embedding space. We can set the initial positions randomly or impose an initial global structure on the embedding. The `openTSNE.initialization` module provides several possible initialization schemes including random, PCA-based, and Laplacian Eigenmap-based initialization schemes. The user can also specify their own initialization by providing a `numpy.ndarray` object. By default, **openTSNE** opts for PCA-based initialization.
3. In optimizing the embedding, the t-SNE algorithm finds the positions of the points in the embedding space that reflect their corresponding similarities in the affinity model. **openTSNE** stores the initial positions and affinities in an `openTSNE.TSNEEmbedding` object that includes the `.optimize` method for embedding optimization.

We adopt the learning rate $\eta = N/\rho$ proposed by [Belkina *et al.* \(2019\)](#), who suggest using a global learning rate computed from the early exaggeration factor ρ . Empirically, we have found that adapting the learning rate to the exaggeration factor ρ used in the second phase of the optimization also further speeds up convergence.

Optimization can be performed using either the Barnes-Hut or FIt-SNE approximation scheme. While the FIt-SNE approximation scheme scales linearly with the number of samples, it introduces additional computational overhead, often unnecessary for smaller data sets, resulting in longer overall runtimes than the Barnes-Hut approximation scheme. By default, **openTSNE** uses an empirically-determined heuristic to choose the fastest algorithm for a given data set based on its number of samples. If the data set contains fewer than 10,000 samples, the Barnes-Hut approximation scheme is selected, otherwise, the FIt-SNE is used.

Once the embedding is constructed and optimized, it is ready to use for any downstream tasks. `openTSNE.TSNEEmbedding` objects inherit from `numpy.ndarray`, which enables easy manipulation and provides interoperability with the broader Python ecosystem.

Embedding new data points to an existing `openTSNE.TSNEEmbedding` follows the same three general steps as constructing a standard embedding:

1. Firstly, we must provide an affinity model describing the similarities between each new data point and the data points in the reference embedding. As for the construction of the initial embedding, **openTSNE** provides several possible affinity models in the `openTSNE.affinity` module. Most often, however, we want to utilize the affinity model from the construction of the reference embedding, as this leads to more spatially consistent embeddings. Using an affinity model from reference embedding for the embedding of the new data is the default behavior of **openTSNE**. The same rationale applies to the nearest neighbor search algorithm.
2. Secondly, we must specify the initial positions of the new data points in the embedding space. We can, again, initialize these positions randomly or according to their similarity to the data points in the reference embedding. As before, **openTSNE** provides several

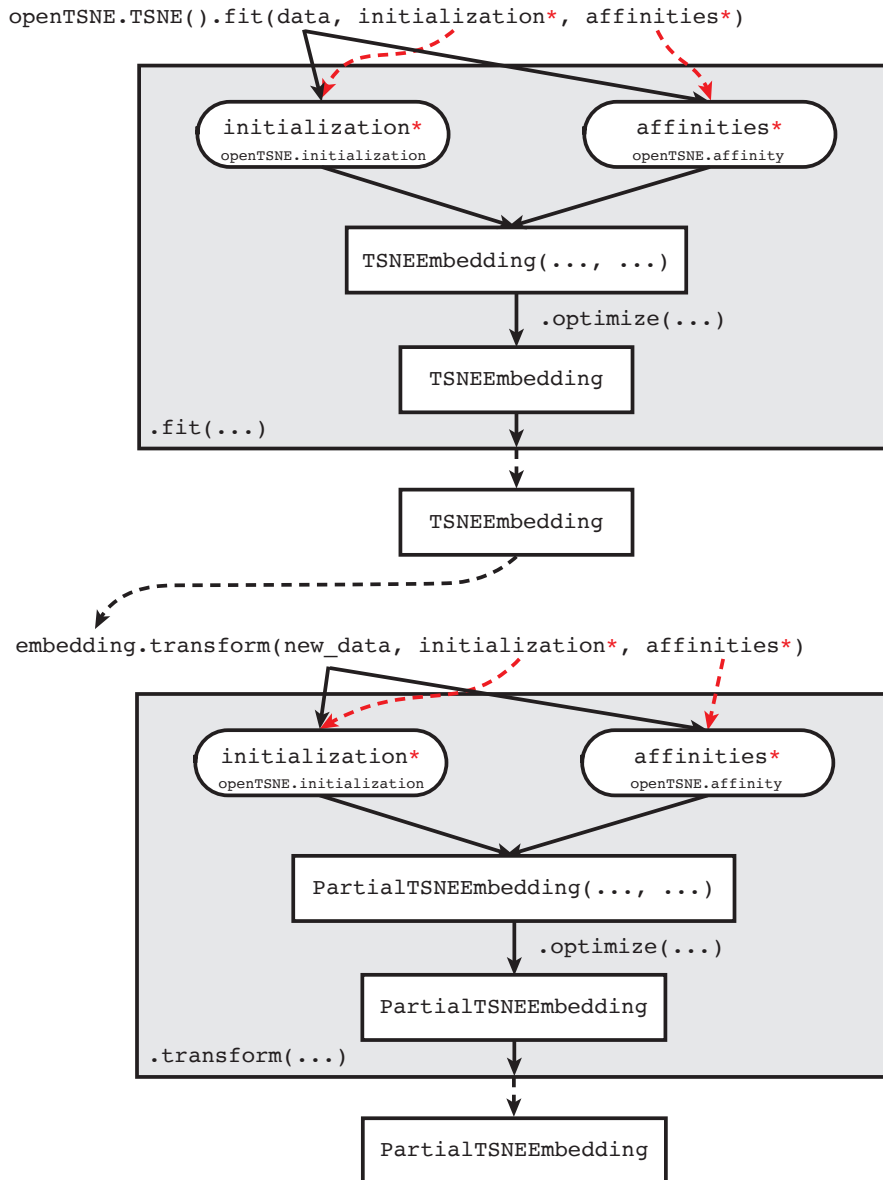


Figure 2: The high-level structure and information flow of the **openTSNE** library. Solid black lines show the standard parameter flow, while red dashed lines indicate the utility of optional parameters, allowing more refined control over the t-SNE algorithm.

initialization options in the `openTSNE.initialization` module, but the user is free to provide their specific initialization.

3. Finally, what remains is a construction of an embedding object, subject to optimization. Because this new embedding relies on an existing reference embedding, we construct a `openTSNE.PartialTSNEEmbedding` embedding object, which indicates its dependence on a reference `openTSNE.TSNEEmbedding` object. `openTSNE.PartialTSNEEmbedding` again inherits from `numpy.ndarray` and can therefore be easily manipulated. Optimization is, again, performed via the `.optimize` method.

Figure 2 shows the high-level structure and information flow of the **openTSNE** library. Through different parameters, the user is given complete control over every aspect of the t-SNE algorithm, from different ways to define affinities to the delicate tuning of optimization parameters. The parametrization of every part of t-SNE allows for rapid experimentation. It enables users to quickly and efficiently inspect their data from numerous angles and at varying levels of resolution.

Most end-users often only need a quick and easy way to create informative visualizations of their data sets. **openTSNE** provides a high-level API that closely follows the style of **scikit-learn** (Buitinck *et al.* 2013), a popular and widely used machine learning toolkit.

3.1. Example

Installation

openTSNE is designed to be easy to use and can be installed through either Python Package Index PyPI using the **pip** package manager,

```
pip install opentsne
```

or using the **conda** package manager from the **conda-forge** repository,

```
conda install -c conda-forge opentsne
```

openTSNE provides precompiled binaries for all major platforms (Windows, macOS, Linux) for all currently supported Python versions (3.7—3.11). Precompiled binaries are especially convenient for Windows users, as they obviate the need for a C/C++ compiler, which is typically not bundled with Windows operating system.

To verify that that **openTSNE** was installed correctly, users can run Python and try to import the module by:

```
>>> import openTSNE
>>> openTSNE.__version__
```

```
1.0.0
```

A simple workflow

We here outline the most straightforward usage of **openTSNE** on the Iris data set from Anderson (1936). This small data set profiles 150 iris flowers with four variables (sepal length, sepal width, petal length, petal width) and provides information about their species (*Iris Setosa*, *Iris Versicolor*, *Iris Virginica*). The Iris data set comes bundled with **scikit-learn**:

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> iris.data[:3]
```

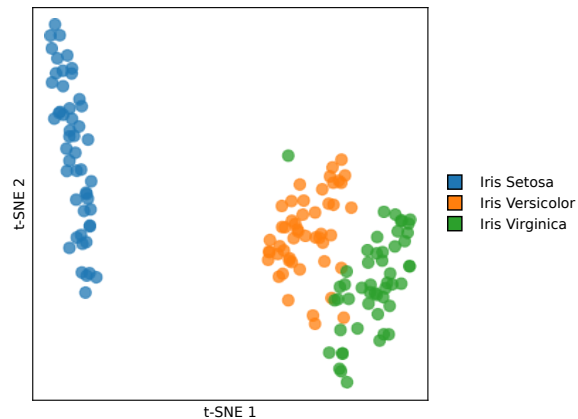


Figure 3: A visualization of the constructed t-SNE embedding on the Iris data set. The colors correspond to different species of Iris.

```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2]])
```

scikit-learn loads the data set into a **numpy** array which can directly be used by **openTSNE**:

```
>>> import openTSNE
>>>
>>> embedding = openTSNE.TSNE().fit(iris.data)
>>> embedding[:3]
```

```
TSNEEmbedding([[ -0.55581917, 17.56162344],
               [-1.76742536, 20.07763453],
               [-0.58444608, 20.00052495]])
```

Figure 3 displays the resulting t-SNE visualization. As expected, t-SNE can correctly group samples from the same species. The t-SNE embedding reveals that flowers from *Iris Setosa* are markedly different from the other two species. At the same time, flowers from *Iris Versicolor* and *Virginica* are more similar to one another.

We can now use the resulting embedding as a reference for new data points. In this simple example, we will duplicate some of the original samples and add them to the embedding.

```
>>> new_data = iris.data[:, :3]
>>> new_embedding = embedding.transform(new_data)
>>> new_embedding[:3]
```

```
PartialTSNEEmbedding([[ -0.02637324, 15.69956674],
                     [-2.84200792, 16.15478451],
                     [-2.4594136 , 15.37910876]])
```

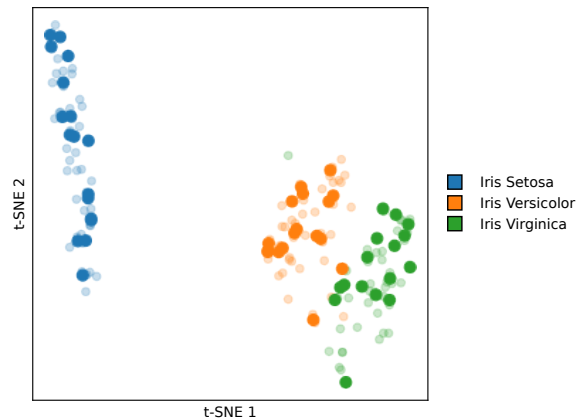


Figure 4: A visualization of the original, reference data set (transparent points) overlaid with the new data points (opaque points).

Figure 4 shows the resulting positions of the new data points, overlaid the original, reference embedding. As expected, t-SNE is correctly able to place the new data points to their corresponding groups.

Peeking under the hood

While the above example offers users a quick and easy way to generate informative visualizations, **openTSNE** allows us to control every step of the t-SNE embedding. To demonstrate this flexibility, we will repeat the same analysis of the Iris data set as before, this time leveraging the library’s more advanced features. We list other practical examples with more complex data sets in Section 4.

We, again, begin by loading the Iris data set using **scikit-learn**:

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> iris.data[:3]
```

```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2]])
```

Next, we import the **openTSNE** library:

```
>>> import openTSNE
```

We now mirror the three t-SNE steps from the previous section. First, we select an affinity model which describes the similarities between data points:

```
>>> affinities = openTSNE.affinity.PerplexityBasedNN(iris.data,
...     perplexity = 30)
```

Next, we determine the initial point positions in the embedding:

```
>>> initialization = openTSNE.initialization.pca(iris.data)
>>> initialization[:3]

array([[ -1.30971087e-04,  1.55848907e-05],
       [ -1.32435711e-04, -8.63672049e-06],
       [ -1.40967409e-04, -7.07276279e-06]])
```

Finally, using the affinity model and initialization, we can create an embedding object:

```
>>> embedding = openTSNE.TSNEEmbedding(initialization, affinities)
>>> embedding[:3]

TSNEEmbedding([[ -1.30971087e-04,  1.55848907e-05],
               [ -1.32435711e-04, -8.63672049e-06],
               [ -1.40967409e-04, -7.07276279e-06]])
```

Notice that the embedding object contains the same point coordinates as the initialization: although we have created an `openTSNE.TSNEEmbedding` object, we have not yet run any optimization. To optimize the embedding, we call the `embedding.optimize` method using our desired parameter settings. We use the default parameter settings here, including a shorter early-exaggeration phase, followed by a longer phase with no exaggeration.

```
>>> embedding.optimize(250, exaggeration = 12, inplace=True)
>>> embedding.optimize(500, inplace = True)
>>> embedding[:3]

TSNEEmbedding([[ -0.55581917, 17.56162344],
               [ -1.76742536, 20.07763453],
               [ -0.58444608, 20.00052495]])
```

We can validate that method produces the exact same embedding as before:

```
>>> np.linalg.norm(openTSNE.TSNE().fit(iris.data) - embedding)

0.0
```

Next, we can add new samples into the embedding,

```
>>> new_embedding = embedding.prepare_partial(iris.data[:, :3], perplexity=5)
>>> new_embedding.optimize(exaggeration = 1.5, n_iter = 250,
...   learning_rate = 0.1, max_grad_norm = 0.25, inplace = True)
>>> new_embedding[:3]

PartialTSNEEmbedding([[ -0.02637324, 15.69956674],
                      [ -2.84200792, 16.15478451],
                      [ -2.4594136 , 15.37910876]])
```

We here use the `embedding.prepare_partial` method to determine initial positions for the new data points, then call the `new_embedding.optimize` method to run the optimization process. Again, we can validate that this also produces the exact same embedding as before:

```
>>> np.linalg.norm(new_embedding - embedding.transform(iris.data[:, :3]))
0.0
```

4. Case studies

We provide three case studies demonstrating the usage of **openTSNE** in different settings using different combinations of hyperparameters to highlight different aspects of the given data sets. The first case study highlights how varying the perplexity and resolution parameters can reveal different aspects of the underlying topology of the studied data set. The second case study demonstrates how **openTSNE** may be used to create embeddings of massive data sets containing millions of data points and how its flexible API allows the quick and efficient construction of embeddings. The final case study illustrates how we can use existing t-SNE embeddings to add new samples into the embedding space. This enables us to reuse carefully annotated visualizations for the rapid characterization of new, unseen data points.

4.1. Uncovering structure in high-dimensional data

Dimensionality reduction techniques implicitly assume that high-dimensional data lies on a lower-dimensional manifold, which can accurately be captured by a small number of dimensions. However, there is no evidence that every data set can accurately be described using only two dimensions, and any such embedding will inevitably lead to a loss of information. Thus, it is beneficial to examine multiple embeddings, each providing a different perspective on the topology and other data characteristics.

We illustrate the benefits of creating different t-SNE plots of the same data set by generating four different embeddings of the data on single-cell gene expression in mouse brain (Tasic *et al.* 2018). We generate the embeddings using the following code snippet (shortened for clarity):

```
>>> embedding_a = openTSNE.TSNE(perplexity = 30).fit(x)
>>> embedding_b = openTSNE.TSNE(perplexity = 500).fit(x)
>>> embedding_c = openTSNE.TSNE(perplexity = [30, 500]).fit(x)
>>> embedding_d = openTSNE.TSNE(perplexity = 30, dof = 0.6).fit(x)
```

Figure 5a shows an embedding using default t-SNE parameters. While different clusters of excitatory and inhibitory neurons appear close to one another, all clusters appear equidistant from their neighbors, and the overall relations between groups are not obvious. The embedding in Figure 5b focuses on preserving larger neighborhoods of points, resulting in a more globally consistent layout where relations between clusters become more apparent. Here, it is evident from the increased white space between groups that there is one large class of excitatory neurons and two related classes of inhibitory neurons. Unfortunately, focusing on preserving large neighborhoods leads to the absorption of smaller clusters into larger ones. Alternatively, Figure 5c uses multi-scale similarity kernels that aim to preserve both

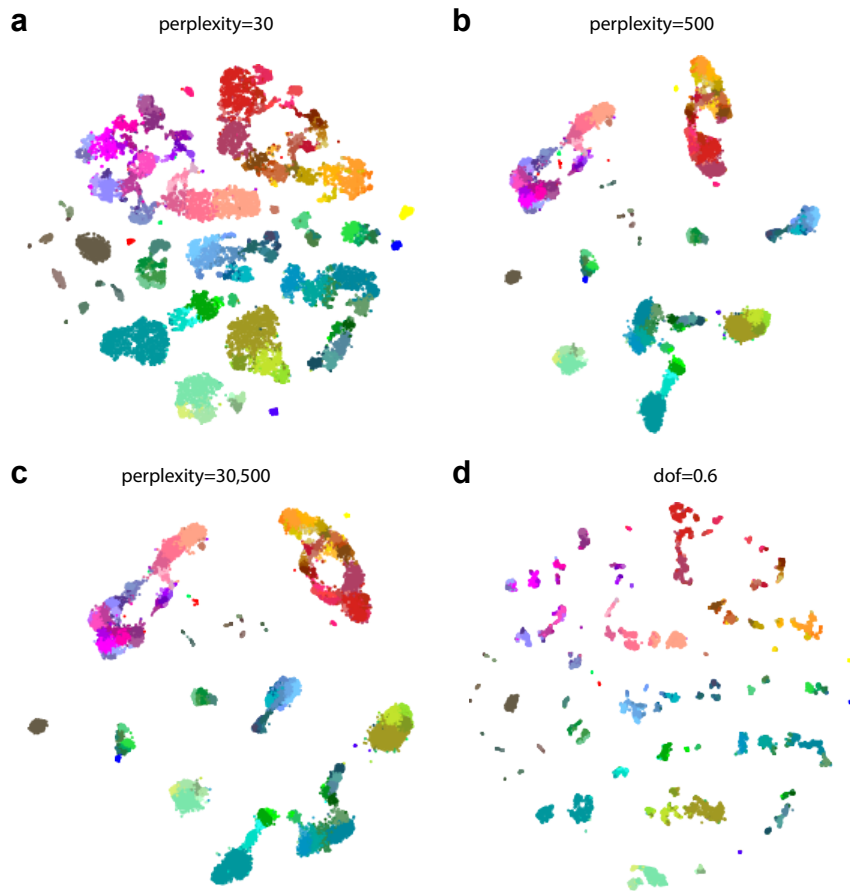


Figure 5: We use **openTSNE** to create four different visualizations of the [Tasic *et al.* \(2018\)](#) data set, each providing a different perspective into the topology of the data. The data set contains 21,874 single-cells originating from the mouse neocortex. Cluster annotations and colors are taken from the original publication. Warm colors correspond to excitatory neurons, cool colors correspond to inhibitory neurons, and gray/brown colors correspond to non-neuronal cells. Standard t-SNE (a) emphasizes local structure while increasing perplexity (b) results in a more meaningful layout of the clusters. We can also combine the two perplexities by using a multi-scale kernel affinity model (c) and obtain a trade-off between global and local structures. Alternatively, we can inspect more fine-grained structures and reveal smaller clusters by using a more heavy-tailed kernel (d).

the global organization of clusters and prevent smaller cluster absorption. We constructed the embedding from Figure 5d with the settings used for Figure 5a, but at a finer level of resolution. The figure demonstrates that some clusters are composed of numerous, smaller subgroups representing different cell subpopulations that are not visible under standard parameter settings.

4.2. Embedding massive data sets

When dealing with data sets containing millions of data points, standard t-SNE embeddings often become unwieldy – cluster boundaries are blurred, large clusters absorb smaller ones,

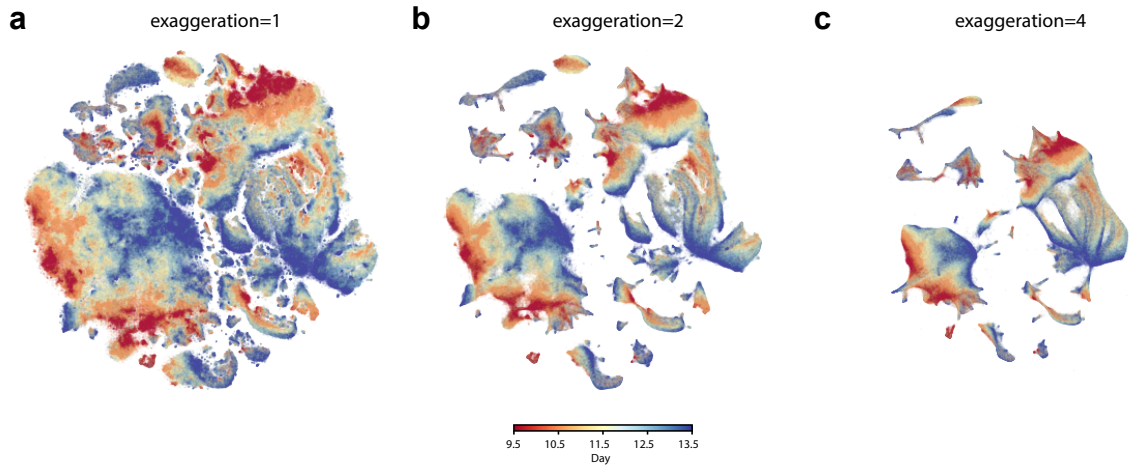


Figure 6: Increasing the exaggeration factor leads to compact clusters, highlighting the data’s global organization and emphasizing the continuous nature of cell state transitions. The data set from [Cao *et al.* \(2019\)](#) contains expression profiles from 2,058,652 single cells. The data were collected from mice embryos at different developmental stages daily after 9.5 to 13.5 days. (c) reveals that the data is comprised of two main components – the neural tube and mesenchymal cells – as well as several other smaller clusters. The colors indicate developmental progression, where red indicates the least-developed cells and blue indicates the most developed cells. The overall developmental trajectory is most apparent with higher exaggeration levels, showing red cells slowly transitioning into blue cells. Progressively easing the exaggeration factor uncovers finer clusters within the larger groups, as shown in (b) with exaggeration of two and subsequently in (a), where we show the standard t-SNE with no exaggeration. 32,011 putative doublets are excluded from the visualizations.

and relationships between groups become increasingly difficult to interpret. We constructed Figure 6a from the data containing expression profiles of over two million single cells captured at different time points in mouse development. The embedding reveals numerous clusters with transitions between time points marked by the color coding, which are difficult to interpret. [Kobak and Berens \(2019\)](#) suggest that increasing attractive forces between similar data points controlled via the *exaggeration* parameter leads to more compact clusters and, subsequently, more informative visualizations. For instance, Figure 6b doubles the default exaggeration, which uncovers some of the data’s overall structure. Further doubling the exaggeration in Figure 6c allows us to observe that the data is comprised of two main groups of cells and eight somewhat smaller clusters. The visualization also reveals several tiny clusters, possibly corresponding to rare cell types.

Exaggeration can highlight transitions between cell stages in developmental studies. Standard t-SNE often produces embeddings with clearly defined, discrete clusters. We can adjust the level of granularity and resolution of the clusters with several parameters, as shown in Figure 5. However, discrete clusters are often undesired in developmental studies where cells’ stage is assumed to follow a continuous transition path. To this end, researchers have used other embedding techniques such as UMAP and ForceAtlas2 to better capture the continuity between cell stages. Recently, [Böhm *et al.* \(2022\)](#) showed that embeddings produced by t-SNE with exaggeration values of 4 and ~ 30 construct embeddings that are markedly similar to

UMAP and ForceAtlas2, respectively. For example, in Figure 6a, the developmental trajectory between different time points is difficult to observe due to many sprawled out clusters. On the other hand, it is easier to trace the development when we increase the exaggeration factor from 1 to 2 to 4 in Figures 6b–c.

We have used the following code to create Figures 6a–c and to demonstrate how we can take advantage of the more advanced features of **openTSNE** to more quickly create meaningful visualizations of massive data sets.

First, we take small subset of data points and create a t-SNE embedding using a high perplexity value to emphasize the high level structure:

```
>>> indices = np.random.permutation(list(range(x.shape[0])))
>>> x_sample, x_rest = x[indices[:25000]], x[indices[25000:]]
>>> init_sample = openTSNE.TSNE(perplexity = 500).fit(x_sample)
```

Next, we will use this embedding to determine the starting positions for the remaining data points:

```
>>> init_rest = init_sample.prepare_partial(x_rest)
```

Lastly, we merge `init_sample` and `init_rest` and restore the original ordering to obtain the full initialization:

```
>>> init_full = np.vstack((init_sample, init_rest))[np.argsort(indices)]
```

Next, we precompute the affinity model. This can take quite a long time and, this way, we can reuse the same affinity model in various different embeddings, without having to repeat the same, costly computation every time:

```
>>> affinities = openTSNE.affinity.PerplexityBasedNN(x, perplexity = 30)
```

Using our precomputed initialization and affinity model, we are now ready to create our `TSNEEmbedding` object and optimize the embedding to our liking:

```
>>> embedding = openTSNE.TSNEEmbedding(init_full, affinities)
>>> embedding_ee = embedding.optimize(n_iter = 500, exaggeration = 12)
>>> embedding_c = embedding_ee.optimize(n_iter = 500, exaggeration = 4)
>>> embedding_b = embedding_c.optimize(n_iter = 500, exaggeration = 2)
>>> embedding_a = embedding_b.optimize(n_iter = 500, exaggeration = 1)
```

4.3. Embedding new samples

Unlike other popular dimensionality reduction techniques such as PCA or autoencoders, t-SNE is a non-parametric method and does not define an explicit mapping to the embedding space. Therefore, embeddings of new data points need to be found through optimization (Poličar *et al.* 2023). **openTSNE** is currently the only publicly available library allowing users to add new samples to existing embeddings in a principled manner.

Figures 1b and 1c demonstrate how we can use a previously labeled single-cell data set and embed cells from a separate experiment into the reference landscape. The reference data

from [Macosko *et al.* \(2015\)](#) contains gene expression profiles from mouse retinal cells. By embedding the samples from a similar experiment on bipolar retinal cells by [Shekhar *et al.* \(2016\)](#), we can correctly map the bipolar cell clusters onto the reference embedding.

The following snippet shows the code necessary to generate Figure 1.

We first construct Figure 1a with the most commonly used parameter values which aims to show that these can lead to poor cluster separation and an overall lack of global consistency:

```
>>> tsne_old = openTSNE.TSNE(initialization = "random", learning_rate = 200,
...   n_iter = 750).fit(x)
```

We generate the embedding from Figure 1b using parameter values emphasizing the global coherence of the resulting visualization. By default, **openTSNE** uses PCA-based initialization and automatically determines the optimal learning rate. This allows **openTSNE** to lower the default number of iterations to 500. To emphasize the global consistency of the resulting embedding, we can also specify multiple perplexity values, resulting in a multiscale affinity model:

```
>>> tsne_multiscale = openTSNE.TSNE(perplexity = [50, 500]).fit(x)
```

Finally, we can embed new samples into our obtained embedding space:

```
>>> new_embedding = tsne_standard.transform(x_new)
```

Embedding single cells into existing reference atlases can also be useful for cell-type classification in cases of unknown cell identities. For instance, in Figure 7, we construct a reference embedding (using the same code snippet as above) using labeled data from [Hochgerner *et al.* \(2018\)](#) containing gene-expression profiles of cells from the mouse brain. The authors assign a type to each cell. We can verify their classification accuracy by visualizing the expression of well-established gene markers for the major cell types. We then embed cells from [Harris *et al.* \(2018\)](#) into the constructed cell atlas. In [Harris *et al.* \(2018\)](#), labels are provided only for neuronal cells. We can quickly identify other non-neuronal cell types in the resulting mapping, including oligodendrocytes and astrocytes. We can further use marker genes to validate that the mapping in the reference landscape is correct.

The examples presented above demonstrate how to use **openTSNE** to quickly gain insight into newly-sequenced, single-cell data sets by utilizing existing cell atlases. The approach is general and not limited to single-cell gene expression. One can, in principle, apply it to any tabular data set regardless of the research field or origin of the data.

5. Discussion

5.1. Versatility

The ability to use and combine different optimization approaches to construct different embedding spaces is another of **openTSNE**'s core design principles. [Kobak and Berens \(2019\)](#) recently provided several recommendations and tricks to obtain better and more meaningful t-SNE visualizations. These include multi-scale similarity kernels, perplexity annealing,

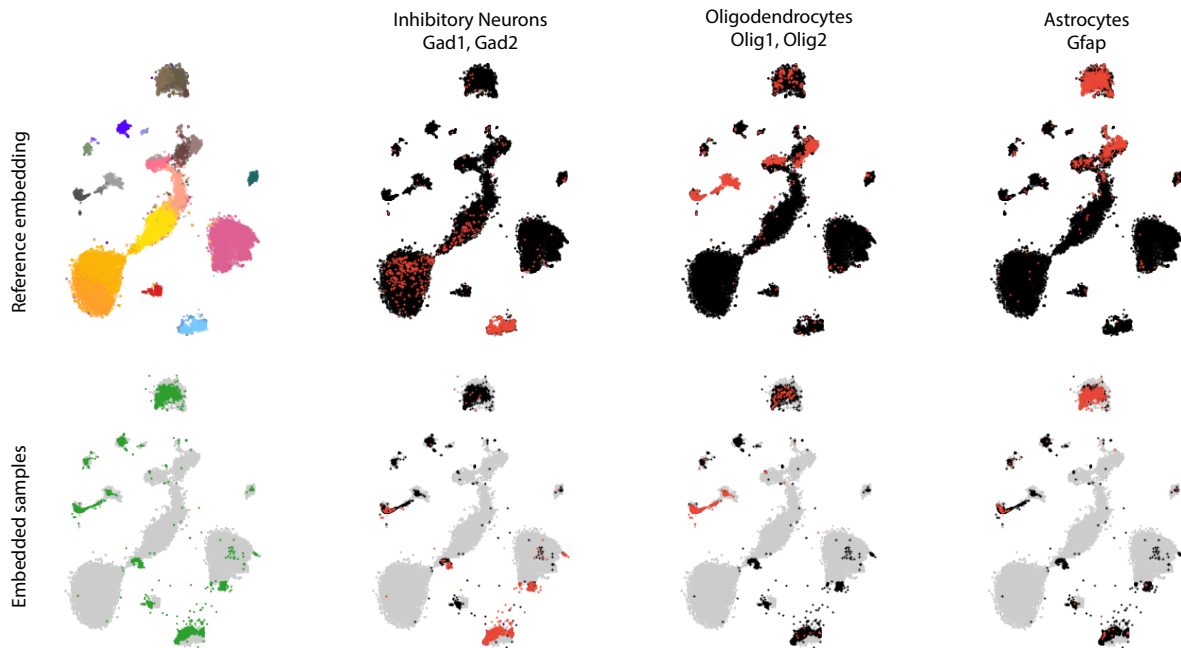


Figure 7: **openTSNE** supports embedding new samples into an existing reference t-SNE landscape. For the series of visualizations shown in this figure, we first construct a t-SNE embedding for the data from Hochgerner *et al.* (2018) containing 24,185 developing, single cells from the mouse hippocampus. The data contains gene expression in different neurons, supporting glia, and other vascular cells (upper left). Each data point corresponds to a single cell colored according to its inferred cell type as determined in the original publication. We use the same color scheme as in Figure 5 where warm colors correspond to excitatory neurons, cool colors correspond to inhibitory neurons, and gray/brown colors correspond to non-neuronal cells. We then embed new hippocampal cells collected in a study by Harris *et al.* (2018) using the embedding of Hochgerner *et al.* (2018) data as a reference. In their research, Harris *et al.* (2018) collected 6,971 single-cells and focused on identifying different types of inhibitory neurons. However, almost half of the collected cells are not neurons and were left uncharacterized. Inspecting the embeddings of these cells in the reference embedding (bottom left) reveals that in addition to inhibitory neurons, the data contains several supporting glial cells and a small population of endothelial cells. We can verify our approach’s accuracy by inspecting marker genes for the major cell types in the reference (top row) and embedded samples (bottom row).

and increasing exaggeration when working with massive data sets. **openTSNE** provides a flexible API to incorporate these improvements in just a few lines of code. Furthermore, **openTSNE** supports custom affinity models, enabling users to construct t-SNE embeddings on non-tabular relational data: the only requirement imposed by the affinity model is a notion of similarity between data points. Finally, **openTSNE**’s comprehensive callback system can be utilized to monitor and adapt different stages of the optimization phase and has been used to construct visually appealing animations of the t-SNE optimization process.

5.2. Speed

One of the t-SNE’s common criticisms is limited scalability to large data sets containing, for instance, millions of data points (Becht *et al.* 2019). Slow response times stem from the optimization procedures and their specific implementations in popular open-source libraries. Most implementations of t-SNE have been based on either a direct implementation of the t-SNE algorithm with asymptotic time complexity $\mathcal{O}(N^2)$ (Van Der Maaten and Hinton 2008), or the Barnes-Hut approximation with asymptotic time complexity $\mathcal{O}(N \log N)$ (Van Der Maaten 2014). Recently, however, Linderman *et al.* (2019) developed a novel approximation – FIt-SNE – which reduces the asymptotic time complexity to $\mathcal{O}(N)$, enabling scaling to large data sets.

Figure 8 benchmarks four popular Python (Van Rossum 1995) t-SNE implementations, including **scikit-learn** (v1.1.2)¹, **MulticoreTSNE** (v0.1), **FIt-SNE** (v1.1.0), and **openTSNE** (v1.0.0). Benchmarks were performed on an Intel Xeon CPU E5-1650 equipped with 128 GB of memory. Benchmarks were run for 1,000 iterations with the original t-SNE parameters, as some implementations do not allow for their modification. To ensure a realistic benchmarking scenario, we utilize the 10X Genomics 1.3 million mouse brain data set (Cao *et al.* 2019), containing 1.3 million single-cell gene-expression profiles. We preprocess the data using the standard single-cell analysis pipeline (Kiselev, Andrews, and Hemberg 2019) and extract the top 50 principal components. To generate benchmark data sets of different sample sizes, we subsample the data set five times at each sample size. In total, we test each implementation on 30 different data sets.

In Python, the most widely-used implementation of t-SNE comes from **scikit-learn**, which exhibits long runtimes when compared to other Python implementations. **scikit-learn**, like its C++ counterpart – **MulticoreTSNE** (Ulyanov 2019) , is based on the Barnes-Hut approximation scheme. However, as shown in Figure 8, **scikit-learn** exhibits somewhat longer runtimes than **MulticoreTSNE**. Newer implementations, such as **FIt-SNE** and **openTSNE**, implement the FIt-SNE approximation scheme, making them suitable for the visualization of millions of data points. While both libraries implement the same algorithm, **openTSNE** emphasizes ease of use and extensibility and is primarily written in Python. Despite this, both libraries exhibit similar runtimes, with **openTSNE** even being marginally faster when utilizing multi-threading, as shown in Figure 8.

Additionally, **openTSNE** provides a flexible API, allowing the user to split the embedding-construction process into several parts. This modularity enables us to cache and reuse intermediate results, allowing users to experiment with different parameter settings and iterate on their final visualizations. Such functionality is not supported in other t-SNE implementations, which require the user to start from scratch for every embedding, resulting in large amounts of redundant computation.

In other programming languages, the selection of fast, open-source implementations of t-SNE is more limited. In R (R Core Team 2024), the most widely used implementation of t-SNE – **Rtsne** (Krijthe 2023) – is based on the Barnes-Hut approximation, and is closely comparable to **MulticoreTSNE**. In Julia (Bezanson *et al.* 2017), the most popular implementation – **TSne.jl** (Jonsson 2021) – is based on a naive $\mathcal{O}(N^2)$ implementation of the algorithm, which allows it to be used only on smaller data sets containing up to thousands of data points. We

¹We also compared the differences in performance between **scikit-learn** v1.1.2 and more the more recent releases v1.2.2 and v1.3.0, but found no discernible differences in the overall runtime.

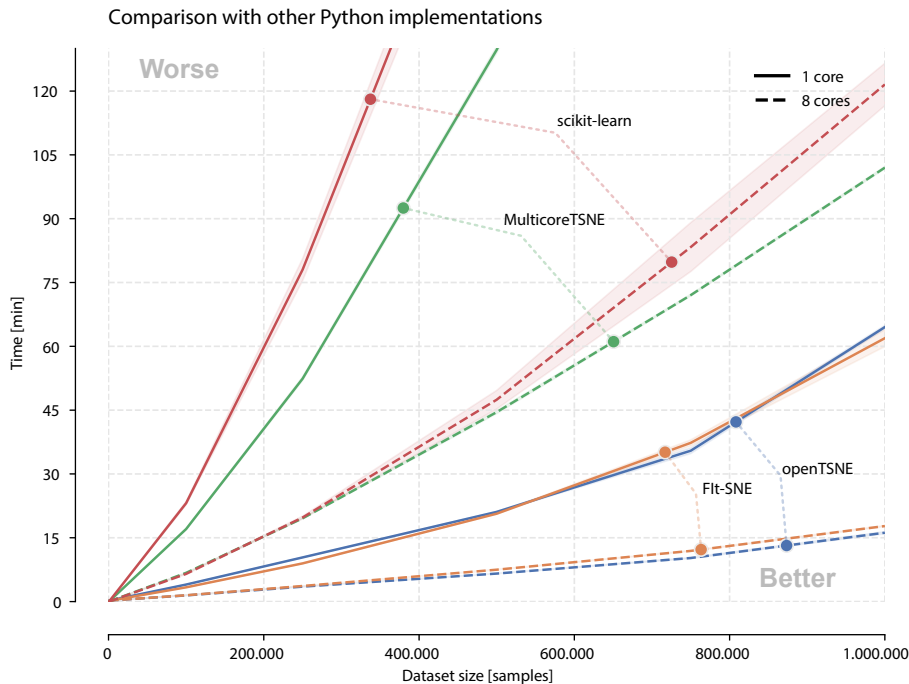


Figure 8: We benchmark **openTSNE** (v1.0.0) against three popular open-source implementations from **scikit-learn** (v1.1.2, Pedregosa *et al.* 2011), **MulticoreTSNE** (v0.1, Ulyanov 2019), and **Fit-SNE** (v1.1.0, Linderman *et al.* 2019). Experiments were run on an Intel Xeon CPU E5-1650 equipped with 128 GB of memory. Notice that **openTSNE** scales similarly to **Fit-SNE**, as they both use the same interpolation-based approximation scheme, while **scikit-learn** and **MulticoreTSNE** utilize the Barnes-Hut approximation.

benchmark **Rtsne** (v0.15) and **TSne.jl** (v1.3.0) and compare it to **openTSNE** (v1.0.0). Figure 9 illustrates the scaling of each approximation scheme. Of these implementations, only **openTSNE** can scale to data sets containing millions of data points in a reasonable amount of time.

We note here that the benchmarks in Figures 8 and 9 were run for 1,000 iterations to ensure a fair comparison between different implementations. However, it has recently been shown that, using an appropriate learning rate, we can safely reduce the number of iterations to 750 (Belkina *et al.* 2019). This means that the actual runtime will be faster than reported in these benchmarks in everyday usage of **openTSNE**.

Comparison to UMAP

We compare the performance of the **openTSNE** (v1.0.0) and **umap-learn** (0.5.3) libraries. **umap-learn** is the official Python implementation of the UMAP algorithm (McInnes *et al.* 2018), an alternative dimensionality reduction technique used for the visualization of high-dimensional data. A throughout comparison of the two algorithms is outside the scope of this manuscript, and we here instead highlight only their similarities and differences as they relate to the runtime of the implementations.

Both t-SNE and UMAP are force-directed layout algorithms in which each data point acts as

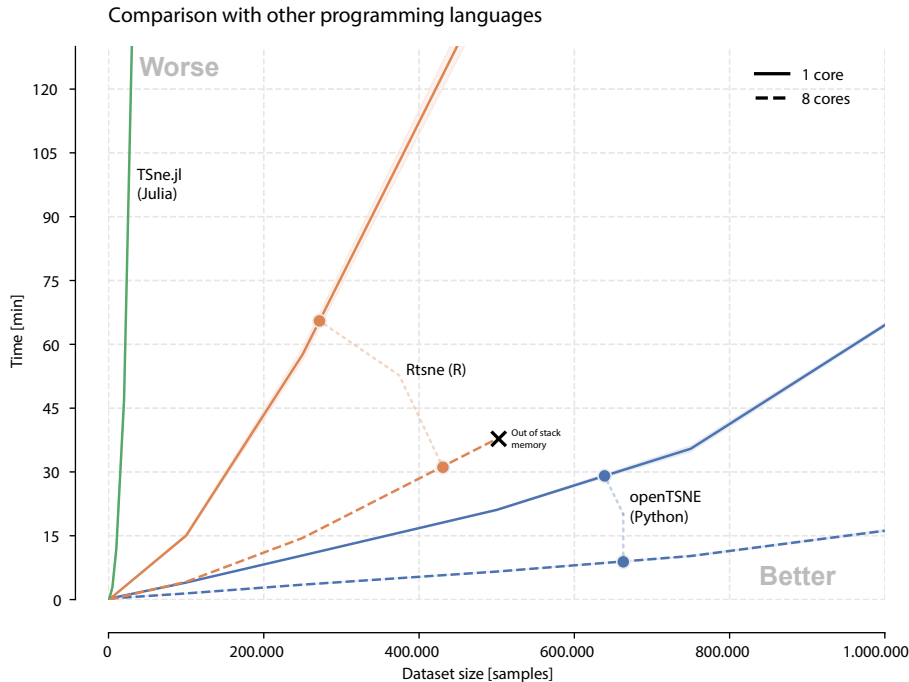


Figure 9: We select the fastest available, open-source implementations of the t-SNE algorithm from the Python, R, and Julia programming languages. We choose **openTSNE** (v1.0.0) for Python, **Rtsne** (v0.15, [Krijthe 2023](#)) for R, and **TSne.jl** (v1.3.0, [Jonsson 2021](#)) for Julia. The benchmarks reflect the time complexity of the implemented approximation schemes. For instance, **TSne.jl** implements a naive $\mathcal{O}(N^2)$ algorithm, making it prohibitively expensive for all but the smallest of data sets. **Rtsne** scales similarly to **MulticoreTSNE** in Python, as it implements the $\mathcal{O}(N \log N)$ Barnes-Hut approximation scheme. Interestingly, the multi-threaded implementation version of **Rtsne** crashes on benchmark data sets containing over 500,000 data points due to a lack of stack memory. We found these crashes to be inconsistent and somewhat random. **openTSNE** is the only library implementing the more recent FI-t-SNE approximation scheme, which makes it suitable for larger data sets.

a particle, attracting and repelling other particles. In the first step, both algorithms use the high-dimensional data set to construct a k -nearest neighbors graph (KNNG). Then, the low-dimensional embedding is optimized such that data points connected in the KNNG attract one another, and all data points exert some small, repulsive force. The two methods differ in the KNNG construction process, the attractive and repulsive forces specification, and their optimization approaches.

The primary factor dictating the runtime of both methods is the number of nearest neighbors used when constructing the KNNG. Apart from the runtime required to find the k -nearest neighbors, the number of neighbors directly influences the runtime of each optimization step. In each step, every data point exerts an attractive force on each neighbor in the low-dimensional embedding. Therefore, the larger the number of neighbors considered, the longer the runtime. Modern implementations of the t-SNE algorithm construct a KNNG with $3 \cdot u$ nearest neighbors where u is the desired perplexity. By default, **openTSNE** (v1.0.0) uses a perplexity value of 30, meaning that 90 nearest neighbors must be found and considered for

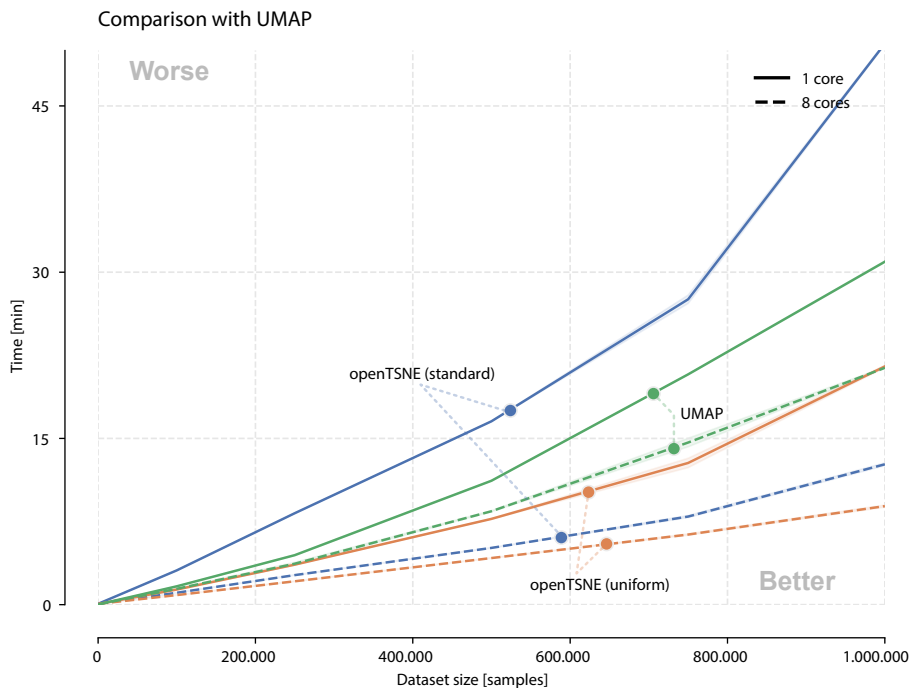


Figure 10: We benchmark **openTSNE** (v1.0.0) against **umap-learn** (v0.5.3), the official Python implementation of the UMAP algorithm. Notice that the standard single-threaded perplexity-based variant of t-SNE runs slower than UMAP but outperforms UMAP in the multi-threaded case. We also benchmark the uniform affinity model variant of t-SNE, which outperforms both standard t-SNE and UMAP.

each data point. In contrast, the **umap-learn** package (v0.5.3) constructs a KNNG using only 15 neighbors by default, substantially reducing the time spent on nearest neighbor search and point interactions compared to t-SNE.

Alternatively, empirical evidence indicates that replacing the standard perplexity-based Gaussian affinity model in t-SNE with a uniform affinity model produces markedly similar embeddings to that of standard t-SNE (Böhm *et al.* 2022). For instance, running standard t-SNE with a perplexity of $u = 30$ constructs a KNNG with $3 \cdot u = 90$ neighbors. However, when using the equivalent uniform affinity model, only u neighbors are needed, making the uniform affinity model attractive for visualizing large data sets.

In Figure 10, we include benchmarks of the standard perplexity-based t-SNE algorithm, its uniform affinity model variant, and UMAP. Differently from the previous benchmarks, which were run using older parameter settings to accommodate different t-SNE implementations, we now run both libraries using their default parameter settings to measure the runtime of typical, real-world use cases.

Given that, by default, UMAP considers six times fewer neighbors than standard t-SNE, it is unsurprising that the single-threaded variant of UMAP outperforms the single-threaded implementation of standard t-SNE. Surprisingly, however, the single-threaded uniform affinity variant of t-SNE outperforms both the single-threaded variant of UMAP and its multi-threaded variant, despite dealing with twice as many neighbors. This result likely stems from

the fact that UMAP is written in pure Python and uses **numba** (Lam, Pitrou, and Seibert 2015) to accelerate computation. On the other hand, we implemented critical elements of the **openTSNE** library in **Cython** (Behnel, Bradshaw, Citro, Dalcin, Seljebotn, and Smith 2011), which is subsequently transpiled into C++, which may be able to be more aggressively optimized by the C++ compiler. Another contributing factor may be the efficiency of the approximation schemes. While the UMAP and FIt-SNE algorithms are asymptotically linear in the number of samples, their real-world runtimes may differ significantly.

The multi-threaded benchmarks show that **openTSNE** outperforms **umap-learn** in standard and uniform variants. We can attribute this to different optimization procedures used by the two algorithms. At each optimization step, **openTSNE** computes all pairwise interactions between data points in the embedding. While directly calculating these interactions would result in quadratic time complexity, **openTSNE** implements efficient approximation schemes that reduce this to linear time complexity (see Section 2.2). Importantly, these approximation schemes are very amenable to parallelization. The runtime differences between the single-core and multi-core performance of **openTSNE** clearly demonstrate this. On the other hand, UMAP bypasses the need to compute all pairwise interactions between data points using negative sampling, allowing it to scale linearly in the number of samples without additional approximation schemes. However, negative sampling is less amenable to parallelization and does not benefit from the availability of additional cores as much as t-SNE.

Lastly, we focus on an often overlooked runtime metric – the runtime per optimization step. Both t-SNE and UMAP comprise two steps. First, both methods construct a KNNG, which runtime is dominated by k -nearest neighbor search. In the second step, a low-dimensional embedding is found via optimization. By default, **openTSNE** runs optimization for 750 iterations. UMAP, on the other hand, uses a heuristic based on the data set size to determine the number of iterations. For data sets containing fewer than 10,000 data points, UMAP opts for 500 iterations. For larger data sets, however, this is reduced to 200 iterations. Figure 10 indicates that even in the single-threaded scenario, the perplexity-based variant of **openTSNE** requires significantly less time per iteration compared to **umap-learn**, as each **openTSNE** benchmark runs 3.75 times more iterations than **umap-learn** despite considering six times more neighbors at each optimization step.

5.3. Ease of use

Intuitive access and simple installation procedures typically correlate with the widespread adoption of novel computational techniques. While the t-SNE implementation from **scikit-learn** fits this requirement, its implementation is prohibitively slow for even moderately-sized data sets that span tens of thousands of data records. Other C++ implementations such as **MulticoreTSNE** and **FIt-SNE** exhibit better scaling in more massive data sets, but the packages do not provide precompiled binaries and require users to compile the software themselves. This problem is critical, for instance, for users of the Windows operating system, where the C++ compiler does not come bundled with the system, making the correct configuration of current t-SNE implementations cumbersome.

We designed **openTSNE** to be accessible to a broader audience. We provide precompiled binaries for all major Python versions on all major platforms, making the installation process as seamless as possible. One can install **openTSNE** through the Python Package Index (PyPI) or **conda** from the **conda-forge** channel, the two most widely adopted Python package

	<i>scikit-learn</i>	<i>MulticoreTSNE</i>	<i>FIt-SNE</i>	<i>openTSNE</i>	<i>Rtsne</i>	<i>Tsne.jl</i>
<i>Ease of installation</i>						
PyPI package	✓	✓	✓	✓		
conda package	✓		✓	✓		
<i>Approximation schemes</i>						
None ($\mathcal{O}(N^2)$)						✓
Barnes-Hut ($\mathcal{O}(N \log N)$)	✓	✓		✓	✓	
FIt-SNE ($\mathcal{O}(N)$)			✓	✓		
<i>Advanced features and extensions</i>						
Extensible affinity models				✓		
Variable degrees of freedom			✓	✓		
Variable exaggeration			✓	✓		
Globally-consistent initialization			✓	✓		
Automatic learning rate			✓	✓		
Adding samples to embeddings				✓		
Callback system				✓		
Interactive optimization				✓		
<i>Project quality</i>						
Actively maintained	✓			✓	✓	✓
Continuous integration	✓	✓		✓	✓	✓
User documentation	✓			✓	✓	
End-to-end usage examples	✓		✓	✓	✓	✓
API reference	✓			✓	✓	

Table 1: Comparison of the features of **openTSNE** (v1.0.0) to **scikit-learn** (v1.1.2), **BHT-SNE** (master), **MulticoreTSNE** (v0.1), and **FIt-SNE** (v1.1.0), as well as the major implementations in R and Julia, **Rtsne** (v0.15), **Tsne.jl** (v1.3.0). Active maintenance indicates whether the project has had any meaningful development in the past year from the time of writing.

managers. **openTSNE**'s high-level interface is inspired by **scikit-learn**, which is well established in the Python data science ecosystem. **openTSNE** implements multi-threaded versions of the Barnes-Hut and FIt-SNE approximation schemes, enabling it to be applied to data sets containing millions of data points. Finally, **openTSNE** is extensible. Its modular design enables researchers to experiment quickly with different parameter settings and easily incorporate custom components into the software. We provide an extended list of features and comparison to other popular t-SNE implementations in Table 1.

6. Conclusion

We introduce **openTSNE**, the most feature-complete, open-source, Python implementation of the widely popular t-SNE visualization algorithm. The library incorporates the latest devel-

opments to the t-SNE algorithm and makes them easily accessible through a familiar API. **openTSNE** implements efficient approximation schemes, making it suitable for visualizing large data sets containing up to millions of data points. Finally, **openTSNE** is easy to install and comes with precompiled binaries available for all major platforms.

Reproducibility

The results in this paper were obtained using Python 3.10.2 with the **openTSNE** v1.0.0 package. All results and figures can be reproduced using the scripts in the supplementary materials, which are also provided in the GitHub repository <https://github.com/pavlin-polycar/opentsne-paper>. Detailed instructions are provided showing how to reproduce all materials and results shown in this manuscript.

All case studies can easily be reproduced on a consumer-grade laptop computer in minutes using the accompanying scripts. Note, however, that the second case study features a large data set and may require up to several hours to complete.

Scripts are included to reproduce the benchmarks of the different implementations of the t-SNE algorithm. However, please be aware that the full benchmark suite can take several days to complete due to the poor performance of other libraries. A smaller, illustrative benchmark suite is provided, more suitable for consumer-grade laptop computers, demonstrating the runtimes of the different t-SNE implementations. Please note, however, that the scaling advantages of **openTSNE** are diminished when running on smaller data sets and will not appear as significant as they would otherwise.

Acknowledgments

We want to thank Dmitry Kobak for helpful discussions regarding t-SNE and his contributions to the source code. We would also like to thank George Linderman for his help with the FIt-SNE algorithm. This work was supported by Slovenian Research Agency (P2-0209, L2-3170).

References

- Anderson E (1936). “The Species Problem in Iris.” *Annals of the Missouri Botanical Garden*, **23**(3), 457–509. doi:10.2307/2394164.
- Beaulaurier J, Zhu S, Deikus G, Mogno I, Zhang XS, Davis-Richardson A, Canepa R, Triplett EW, Faith JJ, Sebra R, Schadt EE, Fang G (2018). “Metagenomic Binning and Association of Plasmids with Bacterial Host Genomes Using DNA Methylation.” *Nature Biotechnology*, **36**(1), 61. doi:10.1038/nbt.4037.
- Becht E, McInnes L, Healy J, Dutertre CA, Kwok IW, Ng LG, Ginhoux F, Newell EW (2019). “Dimensionality Reduction for Visualizing Single-Cell Data Using UMAP.” *Nature Biotechnology*, **37**(1), 38. doi:10.1038/nbt.4314.
- Behnel S, Bradshaw R, Citro C, Dalcin L, Seljebotn DS, Smith K (2011). “**Cython**: The Best of Both Worlds.” *Computing in Science & Engineering*, **13**(2), 31–39. doi:10.1109/mcse.2010.118.

- Belkin M, Niyogi P (2002). “Laplacian Eigenmaps and Spectral Techniques for Embedding and Clustering.” In TG Dietterich, S Becker, Z Ghahramani (eds.), *Advances in Neural Information Processing Systems 14: Proceedings of the 2001 Conference*. doi:10.7551/mitpress/1120.003.0080.
- Belkina AC, Ciccolella CO, Anno R, Halpert R, Spidlen J, Snyder-Cappione JE (2019). “Automated Optimized Parameters for t-Distributed Stochastic Neighbor Embedding Improve Visualization and Analysis of Large Datasets.” *Nature Communications*, **10**(1), 1–12. doi:10.1101/451690.
- Bernhardsson E (2023). *Annoy: Approximate Nearest Neighbors in C++/Python*. Python package version 1.17.3, URL <https://pypi.org/project/annoy/>.
- Bezanson J, Edelman A, Karpinski S, Shah VB (2017). “Julia: A Fresh Approach to Numerical Computing.” *SIAM Review*, **59**(1), 65–98. doi:10.1137/141000671.
- Böhm JN, Berens P, Kobak D (2022). “Attraction-Repulsion Spectrum in Neighbor Embeddings.” *Journal of Machine Learning Research*, **23**(95), 1–32. doi:10.1101/2024.03.26.586728.
- Buitinck L, Louppe G, Blondel M, Pedregosa F, Mueller A, Grisel O, Niculae V, Prettenhofer P, Gramfort A, Grobler J, Layton R, VanderPlas J, Joly A, Holt B, Varoquaux G (2013). “API Design for Machine Learning Software: Experiences From the **scikit-Learn** Project.” In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pp. 108–122.
- Cao J, Spielmann M, Qiu X, Huang X, Ibrahim DM, Hill AJ, Zhang F, Mundlos S, Christiansen L, Steemers FJ, Trapnell C, Shendure J (2019). “The Single-Cell Transcriptional Landscape of Mammalian Organogenesis.” *Nature*, **566**(7745), 496–502. doi:10.1038/s41586-019-0969-x.
- Ding J, Condon A, Shah SP (2018). “Interpretable Dimensionality Reduction of Single-Cell Transcriptome Data with Deep Generative Models.” *Nature Communications*, **9**(1), 2002. doi:10.1101/178624.
- Harris KD, Hochgerner H, Skene NG, Magno L, Katona L, Gonzales CB, Somogyi P, Kessaris N, Linnarsson S, Hjerling-Leffler J (2018). “Classes and Continua of Hippocampal CA1 Inhibitory Neurons Revealed by Single-Cell Transcriptomics.” *PLoS Biology*, **16**(6), e2006387. doi:10.1371/journal.pbio.2006387.
- Hirata J, Hosomichi K, Sakaue S, Kanai M, Nakaoka H, Ishigaki K, Suzuki K, Akiyama M, Kishikawa T, Ogawa K, Masuda T, Yamamoto K, Hirata M, Matsuda K, Momozawa Y, Inoue I, Kubo M, Kamatani Y, Okada Y (2019). “Genetic and Phenotypic Landscape of the Major Histocompatibility Complex Region in the Japanese Population.” *Nature Genetics*, **51**(3), 470–480. doi:10.1038/s41588-018-0336-0.
- Hochgerner H, Zeisel A, Lönnerberg P, Linnarsson S (2018). “Conserved Properties of Dentate Gyrus Neurogenesis Across Postnatal Development Revealed by Single-Cell RNA Sequencing.” *Nature Neuroscience*, **21**(2), 290–299. doi:10.1038/s41593-017-0056-2.

- Jacobs RA (1988). “Increased Rates of Convergence Through Learning Rate Adaptation.” *Neural Networks*, **1**(4), 295–307. doi:10.1016/0893-6080(88)90003-2.
- Jacomy M, Venturini T, Heymann S, Bastian M (2014). “ForceAtlas2, A Continuous Graph Layout Algorithm for Handy Network Visualization Designed for the **Gephi** Software.” *PLOS One*, **9**(6), e98679. doi:10.1371/journal.pone.0098679.
- Jonsson L (2021). **TSne.jl**. Julia package version 1.3.0, URL <https://github.com/lejon/TSne.jl>.
- Kiselev VY, Andrews TS, Hemberg M (2019). “Challenges in Unsupervised Clustering of Single-Cell RNA-Seq Data.” *Nature Reviews Genetics*, **20**(5), 273–282. doi:10.1038/s41576-018-0088-9.
- Kobak D, Berens P (2019). “The Art of Using t-SNE for Single-Cell Transcriptomics.” *Nature Communications*, **10**(1), 1–14. doi:10.1038/s41467-019-13056-x.
- Kobak D, Linderman G (2021). “Initialization Is Critical for Preserving Global Data Structure in Both t-SNE and UMAP.” *Nature Biotechnology*, **39**(2), 156–157. doi:10.1038/s41587-020-00809-z.
- Kobak D, Linderman G, Steinerberger S, Kluger Y, Berens P (2019). “Heavy-Tailed Kernels Reveal a Finer Cluster Structure in t-SNE Visualisations.” In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 124–139. Springer-Verlag.
- Krijthe JH (2023). **Rtsne: t-Distributed Stochastic Neighbor Embedding Using a Barnes-Hut Implementation**. R package version 0.17, URL <https://CRAN.R-project.org/package=Rtsne>.
- Lam SK, Pitrou A, Seibert S (2015). “**Numba**: A LLVM-Based Python JIT Compiler.” In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pp. 1–6.
- Linderman G, Rachh M, Hoskins JG, Steinerberger S, Kluger Y (2019). “Fast Interpolation-Based t-SNE for Improved Visualization of Single-Cell RNA-Seq Data.” *Nature Methods*, **16**(3), 243–245. doi:10.1038/s41592-018-0308-4.
- Macosko EZ, Basu A, Satija R, Nemesh J, Shekhar K, Goldman M, Tirosh I, Bialas AR, Kamitaki N, Martersteck EM, Trombetta JJ (2015). “Highly Parallel Genome-Wide Expression Profiling of Individual Cells Using Nanoliter Droplets.” *Cell*, **161**(5), 1202–1214. doi:10.1016/j.cell.2015.05.002.
- Malkov YA, Yashunin DA (2018). “Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **42**(4), 824–836. doi:10.1109/tpami.2018.2889473.
- McInnes L (2024). **PyNNDescent: A Python Nearest Neighbor Descent for Approximate Nearest Neighbors**. Python package version 0.5.12, URL <https://pypi.org/project/pynndescent/>.

- McInnes L, Healy J, Melville J (2018). “UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction.” *arXiv 1802.03426*, arXiv.org E-Print Archive. doi: [10.48550/arXiv.1802.03426](https://doi.org/10.48550/arXiv.1802.03426).
- Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay É (2011). “**scikit-learn**: Machine Learning in Python.” *Journal of Machine Learning Research*, **12**, 2825–2830.
- Poličar PG, Stražar M, Zupan B (2023). “Embedding to Reference t-SNE Space Addresses Batch Effects in Single-Cell Classification.” *Machine Learning*, **112**, 721–740. doi: [10.1007/s10994-021-06043-1](https://doi.org/10.1007/s10994-021-06043-1).
- R Core Team (2024). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL <https://www.R-project.org/>.
- Shekhar K, Lapan SW, Whitney IE, Tran NM, Macosko EZ, Kowalczyk M, Adiconis X, Levin JZ, Nemesh J, Goldman M, McCarroll SA, Cepko CL (2016). “Comprehensive Classification of Retinal Bipolar Neurons by Single-Cell Transcriptomics.” *Cell*, **166**(5), 1308–1323. doi: [10.1016/j.cell.2016.07.054](https://doi.org/10.1016/j.cell.2016.07.054).
- Sheth RU, Li M, Jiang W, Sims PA, Leong KW, Wang HH (2019). “Spatial Metagenomic Characterization of Microbial Biogeography in the Gut.” *Nature Biotechnology*, **37**(8), 877–883. doi: [10.1038/s41587-019-0183-2](https://doi.org/10.1038/s41587-019-0183-2).
- Tasic B, Yao Z, Graybuck LT, Smith KA, Nguyen TN, Bertagnolli D, Goldy J, Garren E, Economo MN, Viswanathan S, Penn O, Bakken T, Menon V, Miller J, Fong O, Hirokawa KE, Lathia K, Rimorin C, Tieu M, Larsen R, Casper T, Barkan E, Kroll M, Parry S, Shapovalova NV, Hirschstein D, Pendergraft J, Sullivan HA, Kim TK, Szafer A, Dee N, Groblewski P, Wickersham I, Cetin A, Harris JA, Levi BP, Sunkin SM, Madisen L, Daigle TL, Looger L, Bernard A, Phillips J, Lein E, Hawrylycz M, Svoboda K, Jones AR, Koch C, Zeng H (2018). “Shared and Distinct Transcriptomic Cell Types Across Neocortical Areas.” *Nature*, **563**(7729), 72–78. doi: [10.1038/s41586-018-0654-5](https://doi.org/10.1038/s41586-018-0654-5).
- Tkachev A, Stepanova V, Zhang L, Khrameeva E, Zubkov D, Giavalisco P, Khaitovich P (2019). “Differences in Lipidome and Metabolome Organization of Prefrontal Cortex Among Human Populations.” *Scientific Reports*, **9**(1), 1–10. doi: [10.1038/s41598-019-53762-6](https://doi.org/10.1038/s41598-019-53762-6).
- Ulyanov D (2019). **MulticoreTSNE**. Python package version 0.1, URL <https://pypi.org/project/MulticoreTSNE/>.
- Van Der Maaten L (2014). “Accelerating t-SNE Using Tree-Based Algorithms.” *Journal of Machine Learning Research*, **15**(1), 3221–3245.
- Van Der Maaten L, Hinton G (2008). “Visualizing Data Using t-SNE.” *Journal of Machine Learning Research*, **9**, 2579–2605.
- Van Rossum G (1995). “Python Reference Manual.” *R 9525*, Department of Computer Science.

A. Single-cell RNA-seq data preprocessing

We here describe the preprocessing steps applied to the single-cell RNA-seq data sets used in Figure 1 (Macosko *et al.* 2015), Figure 5 (Tasic *et al.* 2018), Figure 6 (Cao *et al.* 2019), and Figure 7 (Hochgerner *et al.* 2018). The standard single-cell preprocessing pipeline that have used consists of gene filtering, gene selection, data normalization, and dimensionality reduction.

First, we discard genes detected fewer than ten times, then select the 3,000 most highly-variable genes following the gene-selection procedure from Kobak and Berens (2019). We then perform counts-per-median library-size normalization followed by log normalization and standardization without zero-centering. Finally, we extract the top 50 principal components, which are then used to construct subsequent t-SNE embeddings. This pipeline is applied to all the aforementioned data sets with the following exceptions, which serve to better illustrate our points:

- In Macosko *et al.* (2015), we perform library-size normalization by computing counts-per-million and apply centering during standardization.
- In Hochgerner *et al.* (2018), we select the 1,000 most variable genes.

When embedding additional data points into the existing embedding in Figures 1c and 7, we follow the procedure from Poličar *et al.* (2023). Therefore, no preprocessing is applied to the data sets from Shekhar *et al.* (2016) and Harris *et al.* (2018), and the similarities between the reference data set and the new data set are computed on the raw counts.

Affiliation:

Pavlin G. Poličar
Faculty of Computer and Information Science
University of Ljubljana
Večna pot 113, Ljubljana, Slovenia
E-mail: pavlin.policar@fri.uni-lj.si