`miniumap`: an efficient Python implementation of UMAP algorithm for general-purpose dimensionality reduction

Team 9

Abstract

One of the important steps of single-cell RNA sequencing data analysis is dimensionality reduction which allows researchers to visualize and interpret the data. Recently, a new dimensionality reduction algorithm named UMAP was developed which performs well compared to other algorithms like t-SNE. Here, we developed `miniumap` which is an efficient Python implementation of UMAP algorithm for general-purpose dimensionality reduction. To test `miniumap`, we selected a hand-written digits (mnist) dataset and a single-cell RNA sequencing read count matrix of mouse cell atlas and compared the performance of `miniumap` to that of the well-developed UMAP module. Assessed by visual inspection as well as the quantitative measurement using k-nearest neighbors algorithm, `miniumap` achieved high accuracy on both datasets. The running time and memory usage of `miniumap` are slightly higher than that of UMAP module. The advantages and disadvantages of `miniumap` are also discussed in this report.

Introduction

Over the past decades, single-cell RNA sequencing technologies have been dramatically advanced with more and more commercial solutions being developed and more widely used by researchers. Because of the high throughput and resolution, single-cell RNA sequencing can dissect cellular heterogeneity, reveal regulatory relationships between genes, and track the cell lineage trajectories during the developmental process (Hwang *et al.*, 2018). However, single-cell RNA sequencing data has thousands of dimensions. The tools for dimensionality reduction are required to visualize and interpret the data. Here, we implemented uniform manifold approximation and projection (UMAP) to perform dimensionality reduction of high dimensional data (McInnes *et al.*, 2018). UMAP can better preserve the global structure as well as the continuity of the cell subsets and run faster compared with other dimensionality reduction methods like t-SNE (Becht *et al.*, 2019). To assess the performance of `miniumap`, we used datasets including hand-written digits (mnist) dataset (sklearn.datasets.load_digits — scikit-learn 1.0.1 documentation) and single-cell RNA sequencing data of mouse cell atlas (MCA) (Han *et al.*, 2018). To measure the accuracy, we compared the results from `miniumap` and UMAP module by visual inspection and calculating the accuracy using k-nearest neighbor algorithm. We showed that `miniumap` was reliable. but there are still some limitations, e.g. We also compared the maximal memory usage and running time.

Implementation

The input of `miniumap` consists of the dataset, the path for output and optional parameters. Note that each row of the input dataset is one data point, while each column is one dimension. The output is the x and y coordinates of each data point and a 2D scatter plot (Figure 1).

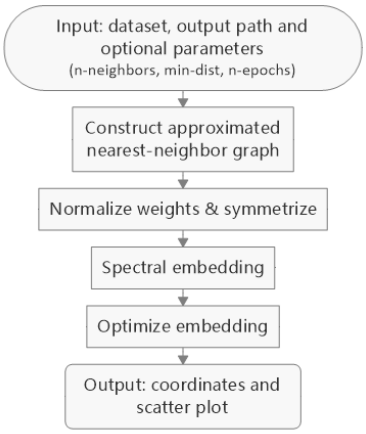

Figure 1. The flow chart of `miniumap`.

Our implementation of UMAP follows the algorithm pseudocode in the original UMAP paper, applying Cython in critical functions for acceleration. In this section, we recap the original UMAP algorithm and then introduce our pseudocode.

First, an **approximated nearest-neighbor graph** with respect to any distance metric is computed with the NN-Descent algorithm [citation needed]. Then, the graph is normalized for each neighborhood to assign a (asymmetrical) weight to each neighbor (called the **membership strength** in the **local fuzzy simplicial set**) such that all neighborhoods have the same weight sum and at least one "sure" neighbor (with membership strength = 1). Formally, for each data point x_i and its neighboring point x_{i_j}, let

$$\rho_i = \min_j d\left(x_i, x_{i_j}\right) \tag{1}$$

be the distance to the *closest neighbor*. A *smoothed normalization factor*, σ_i, is selected such that

$$\sum_j \exp\left(\frac{-\max\left(0, d\left(x_i, x_{i_j}\right) - \rho_i\right)}{\sigma_i}\right) = \log_2 k \tag{2}$$

where k is the number of neighbors.

Thus, the asymmetrical weight assigned to the directed edge $\left(x_i, x_{i_j}\right)$ is

$$w\left(\left(x_i, x_{i_j}\right)\right) = \exp\left(\frac{-\max\left(0, d\left(x_i, x_{i_j}\right) - \rho_i\right)}{\sigma_i}\right) \tag{3}$$

Computationally, σ_i is obtained with **binary search**, thanks to the left-hand side of (2) monotonically increasing.

Algorithm 1. Compute local graph topology

```
procedure BuildLocalFuzzySimplicialSet(X[n, D], A[n, d], x_index, k)
    knn_indices, knn_dists = NNDescent(X, x_index, k)
    rho = knn_dists[1]
    sigma = SmoothKNNDists(knn_dists, k, rho)

    for y_index, y_dist in knn, knn_dists
        A[x_index, y_index] = given by (3)
    end
```

```
    return A[x_index, :]
end
```

Algorithm 2. Compute normalizing factor σ

```
procedure SmoothKNNDists(knn_dists[n], k, rho; PRECISION)
    let f(sigma; rho=rho) be given by (2)
    sigma_lo, sigma_hi = 0.0, 1.0
    while f(sigma_hi) <= log2(k)
        sigma_hi *= 2
    end

    while (sigma_hi - sigma_lo) < PRECISION
        sigma_mid = (sigma_hi + sigma_lo)/2
        if f(sigma_mid) <= log2(k)
            sigma_lo = sigma_mid
        else
            sigma_hi = sigma_mid
        end
    end

    return sigma_mid
end
```

After computing the weighted directed graph (a.k.a. local fuzzy simplicial sets), we need to symmetrize the graph (a.k.a. **union all local fuzzy simplicial sets**). Because the mathematical rationale assigns a meaningful value (here membership strength) to each weight, it is natural to use the **probabilistic t-conorm** to combine the weights. Formally, let p be the probability that x_1 belongs to the neighborhood of x_2, and q conversely. The probability that *there is at least some connection between* x_1 *and* x_2 is then

$$p + q - pq$$

Computationally, let A be the adjacency matrix of the weighted directed graph, we symmetrize the graph to be

$$B = A + A^T - A * A^T \tag{4}$$

where $*$ is pointwise multiplication.

With the topological representation of the dataset at hand, UMAP **minimizes the total cross entropy between the embedding and original layout**. The initial layout could in principle be randomly generated, but **spectral layout** can be used on the topology for faster convergence and stability. Formally, with B given by (4), let D be the degree matrix of B, the normalized Laplacian matrix is

$$L = I - D^{-\frac{1}{2}} B D^{-\frac{1}{2}} \tag{5}$$

The spectral embedding is thus the eigenvectors with 2nd-$(k+1)$th smallest eigenvalues (in

magnitude).

Algorithm 3. Initial layout by spectral embedding

```
procedure SpectralEmbedding(B[n, n], k)
    let L be given by (5)
    eigvals, eigvecs = eigh(L, which='SM', v0=ones(n=n))
    return eigvecs[1:k+1]
end
```

The total fuzzy set cross entropy is

$$\begin{aligned} C\big((G,\mu),(G,\nu)\big) &= \sum_{e\in G} \mu(e)\log\frac{\mu(e)}{\nu(e)} + \big(1-\mu(e)\big)\log\frac{1-\mu(e)}{1-\nu(e)} \\ &= \sum_{e\in G} \mu(e)\log\mu(e) + \big(1-\mu(e)\big)\log\big(1-\mu(e)\big) \\ &\quad - \sum_{e\in G} \mu(e)\log\nu(e) + \big(1-\mu(e)\big)\log\big(1-\nu(e)\big) \end{aligned} \tag{6}$$

We only need to minimize

$$-\sum_{e\in G} \mu(e)\log\nu(e) + \big(1-\mu(e)\big)\log\big(1-\nu(e)\big)$$

The first term in the summand is handled with **sampling**. We sample **1-simplices** (edges) e with probability $\mu(e)$ (weight) and maximize $\log\nu(e)$. The second term is handled with **negative sampling**. That is, instead of maximizing $\big(1-\mu(e)\big)\log\big(1-\nu(e)\big)$ over all $e\in G$, we randomly sample a small number of $e'\in G$ whenever we have sampled a 1-simplex for the first term, assuming them to have membership strength 0. It therefore only remains to find a smooth approximation for the **embedding membership function** to use in stochastic gradient descent:

$$\nu\big((x,y)\big) = \Psi_{mindist}(x,y) = \begin{cases} 1, & \|x-y\|_2 \le mindist \\ \exp\big(-(\|x-y\|_2 - mindist)\big), & otherwise \end{cases} \tag{7}$$

This is done by fitting Ψ against the curve

$$\Phi_{a,b}(x,y) = \frac{1}{1+a\|x-y\|_2^{2b}} \tag{8}$$

The gradients of interest are (ε is a small positive number to prevent division by zero)

$$\nabla(\log\nu(x,y)) \approx \nabla\big(\log\Phi_{a,b}(x,y)\big) = \frac{-2ab\|x-y\|_2^{2(b-1)}}{1+a\|x-y\|_2^{2b}}(x-y) \tag{9}$$

$$\nabla\big(\log\big(1-\nu(x,y)\big)\big) \approx \nabla\left(\log\left(1-\Phi_{a,b}(x,y)\right)\right) = \frac{2b}{(\varepsilon+\|x-y\|_2^2)\big(1+a\|x-y\|_2^{2b}\big)}(x-y) \tag{10}$$

Algorithm 4. Optimize final embedding

```
procedure OptimizeEmbedding(topology[n, n], Y[n, d], min_dist, n_epochs,
                            n_neg_samples)
    alpha = 1.0
    a, b = fit (8) against (7)
    for e in 1:n_epochs+1
        for i, j, p in nonzero(topology)
            if random() < p
                Y[i, :] += alpha * (9)
```

```
            end
            for _ in 0:n_neg_samples
                c = randint(0, n-1)
                Y[i, :] += alpha * (10)
            end
        end
        alpha = 1.0 - e/n_epochs
    end
    return Y
end
```

Finally, we have the main algorithm:

Algorithm 5. Main algorithm

```
procedure UMAP(X[n, D], k, d, min_dist, n_epochs, n_neg_samples)
    let A[n, n] be a new sparse LIL matrix
    for x_index in 0:n
        BuildLocalFuzzySimplicialSet(X, A, x_index, k)
    end
    let B be given by (4)
    Y = SpectralEmbedding(B, k)
    return OptimizeEmbedding(B, Y, min_dist, n_epochs, n_neg_samples)
end
```

A screenshot of our code for the main algorithm is as follows

Code. Main algorithm

```
def umap(X, k=15, d=2, min_dist=0.1, spread=1.0, n_epochs=300, random_seed=None, n_neg_samples=5, metric='euclidean'):
    n_datapoints = X.shape[0]

    # Adjacency matrix of directed weighted kNN graph
    #     A[i, j] = Pr{ j belongs to the neighborhood of i }
    asymm_weights = scipy.sparse.lil_matrix((n_datapoints, n_datapoints), dtype=np.double)
    # Note that argknn[0] == arg(x, X) and knn_dists[0] == 0
    print("Computing nearest neighbor graph...")
    all_argknn, all_knn_dists = approx_nearest_neighbors(X, k, metric=metric, random_seed=random_seed)
    print("Finished.")

    print("Computing weights...")
    # For each row (datapoint) i in X, updates weights for neighbors of i (W[i, i[...]])
    for i in tqdm.trange(0, len(all_knn_dists)):
        rho = all_knn_dists[i][1]
        sigma = smooth_knn_dist(all_knn_dists[i], k, rho)
        for argxn, xn_dist in zip(all_argknn[i], all_knn_dists[i]):
            # Thus the maximum weight is 1 (along the diagonal and closest neighbors)
            asymm_weights[i, argxn] = np.exp(-max(0, xn_dist - rho) / sigma)
    del all_argknn, all_knn_dists
    print("Finished.")

    print("Computing spectral embedding...")
    # Symmetrize into undirected weighted kNN graph using probabilistic t-conorm
    asymm_weights = asymm_weights.tocsc()
    awt = asymm_weights.transpose()

    # A side note: the behavior of operator * in NumPy depends on the data type, so we
    # always refer to the explicit methods (multiply, dot and matmul)
    symm_weights = asymm_weights + awt - asymm_weights.multiply(awt)
    del asymm_weights, awt

    # Generate an initial low-dimensional representation
    init_repr = spectral_embedding(symm_weights, d)
    print("Finished.")

    random.seed(random_seed)
    print("Optimizing final representation...")
    # Optimize the final representation
    if d == 2:
        final_repr = optimize_embedding2(symm_weights, init_repr, min_dist, spread, n_epochs, n_neg_samples)
    else:
        final_repr = optimize_embedding(symm_weights, init_repr, min_dist, spread, n_epochs, n_neg_samples)
    print("Finished.")
```

Results

Two datasets were used to test the algorithm. The first one is the digits dataset obtained from sklearn.datasets module (sklearn.datasets.load_digits — scikit-learn 1.0.1 documentation). Each data point contains the feature of an 8x8 pixel image of a hand-written digit. The digits contain 10 types (“0” to “9”). The dataset contains 1797 data points. Each data point has 64 dimensions. We tested `miniumap` using same parameters as default (n-neighbors = 15, min-dist = 0.1, n-epochs = 200). By comparing the results from `miniumap` and UMAP module using visual inspection, `miniumap` produced similar good results as UMAP module (Figure 2). Different types of digits were clearly separated from each other. To further measure the accuracy of `miniumap`, I used the k-nearest neighbors algorithm (k=15) to calculate the probable label of every data point after dimensionality reduction. Then, if the calculated label is different from the original label, we considered it as a mistake. The accuracy was calculated by dividing the number of correctly labelled data points by the total number of data points. Both `miniumap` and UMAP module achieved similar high accuracies, which are around 98.66%. The confusion matrixes also show similar classification results (Figure 3). The wrong classification could be caused by poor or ambiguous handwriting.

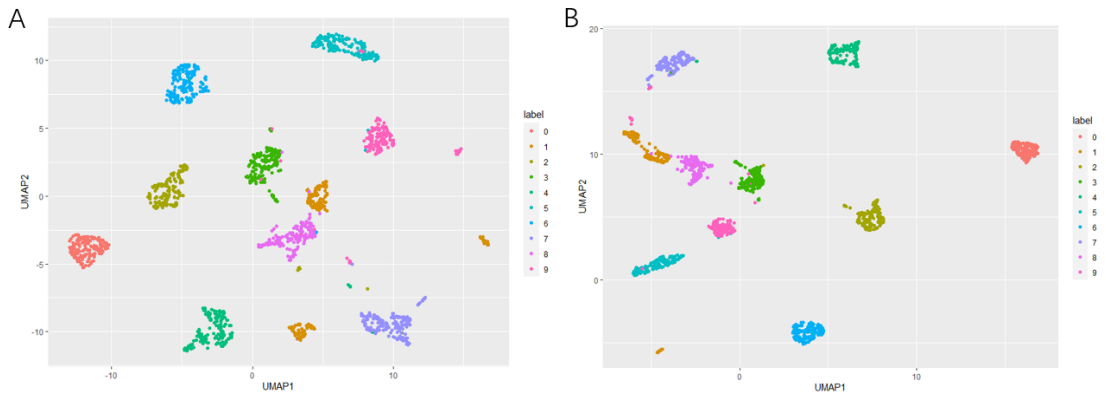

Figure 2. The dimensionality reduction of mnist dataset obtained from `miniumap` and UMAP module. **A**, `miniumap`. **B**, UMAP module.

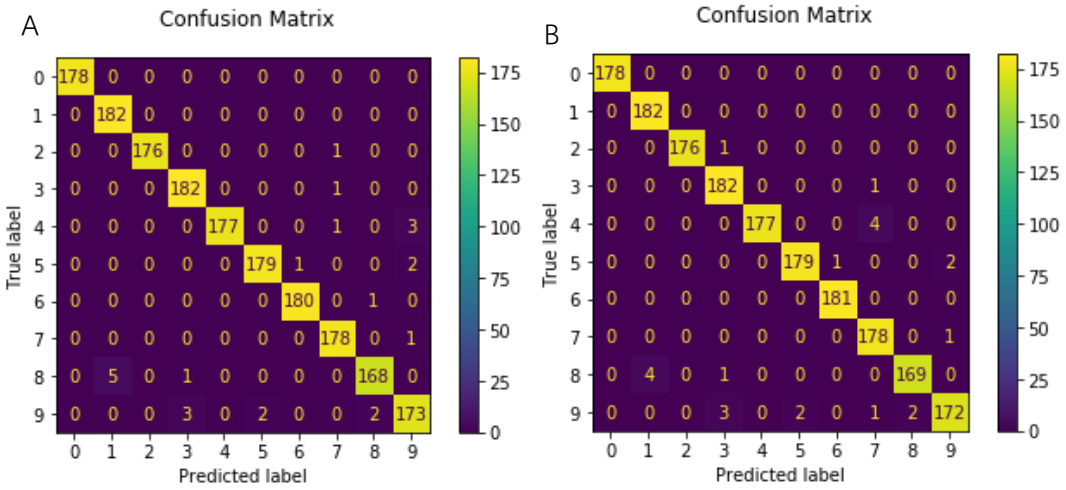

Figure 3. The confusion matrix of k-nearest neighbor classification. **A**, `miniumap`. **B**, UMAP module.

The second dataset is the single-cell RNA sequencing read count matrix of mouse cell atlas (Han *et al.*, 2018). In Han et al's study, the mRNA of 405191 cells from different mouse tissues and developmental stages were sequenced to build the transcriptional landscape of mouse cells. We subset 20000 cells from all 405191 cells to reduce the workload of the computer. The dimensions were first reduced to 100 using PCA to exclude genes that do not contribute much to the overall variance, which therefore helped achieve better results later (Becht *et al.*, 2019). We performed dimensionality reduction using both `miniumap` and UMAP module (parameters: n-neighbors: 30, min-dist: 0.2, e-pochs: 200). The results looked more chaotic than mnist dataset, but we can still see most cells were clustered quite well (Figure 4). We also calculated the accuracy using knn algorithm. miniumap achieved an accuracy of 80.95%, which was slightly better than the accuracy of UMAP module (79.32%).

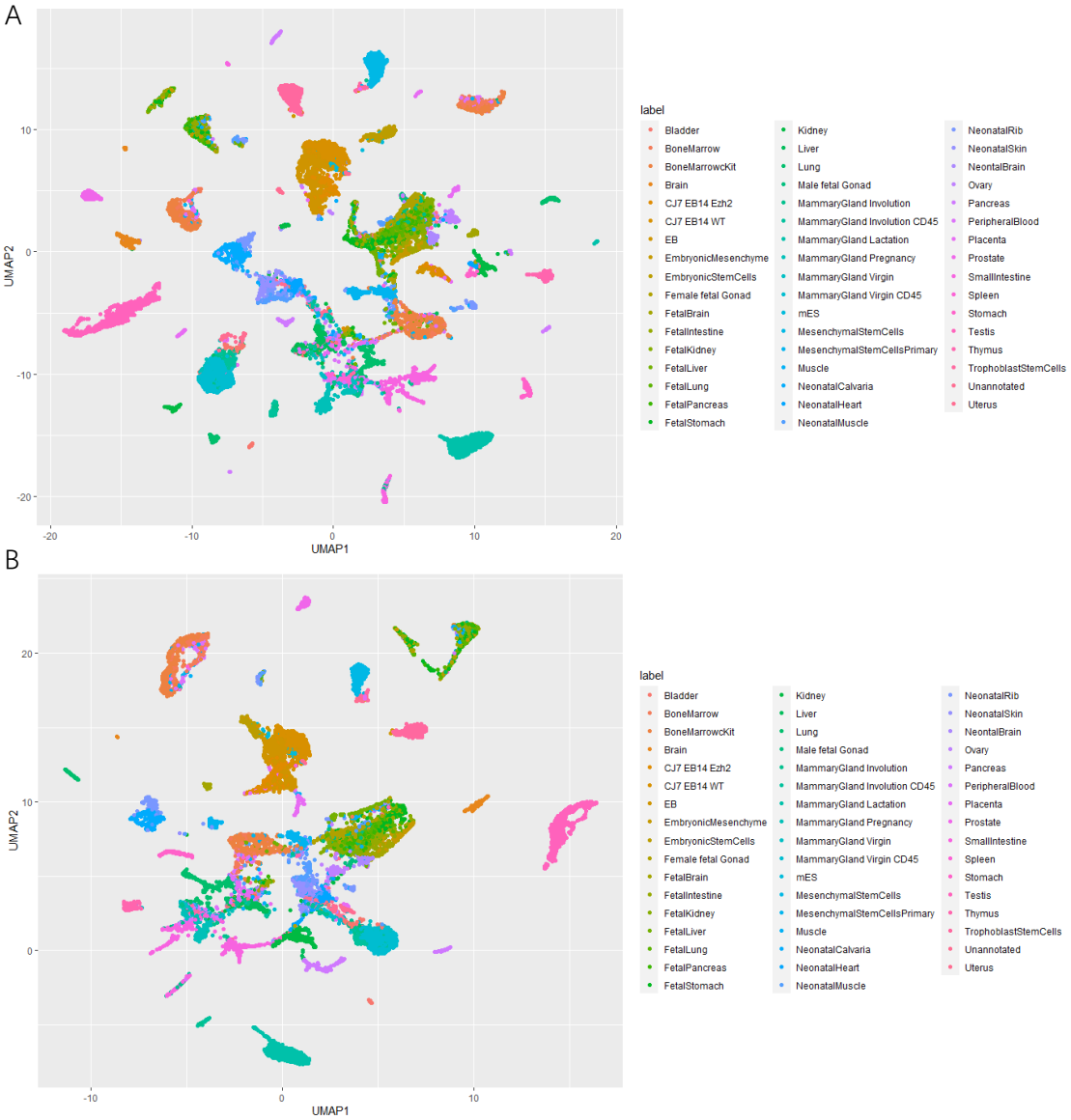

Figure 4. The dimensionality reduction of MCA dataset obtained from miniumap and UMAP module. **A**, `miniumap`. **B**, UMAP module.

We compared the running time and memory usage. The running time of `miniumap` is slightly higher than UMAP module. If the dataset has 20000 data points and 100 dimensions, the running time is around 70 seconds for `miniumap`, while 20-40 seconds for UMAP module. The memory usage of `miniumap` is approximately 200 MB, which is slightly higher than UMAP module (160 MB). But there should be no problem for `miniumap` to be run on local computers.

Discussion

In `miniumap`, we developed a Python implementation of the UMAP algorithm used for dimensionality reduction. It has many advantages which justify it as a practical tool. `miniumap` is implemented efficiently using sparse matrices and Cython acceleration, and is scalable for large data size and dimension. `miniumap` can make use of a given random seed for reproducible output. `miniumap` can calculate data topology using a wide variety of metric functions (as supported by

`pynndescent`), making it a versatile tool for all types of datasets. `miniumap` (optionally) automatically visualizes embedding after each run for the user to inspect conveniently. `miniumap` is accessible from both Python code and command line, making it easy to use.

Despite the various advantages of our implementation, we note that several existing software packages also implement UMAP, and have some advantages over `miniumap`. The most notable is the official implementation in Python module umap. Although umap cannot be directly accessed from command line, it is optimized for wider use cases, has advanced visualization facilities, can be directly integrated with scikit-learn, and is notably faster than our implementation. In the following discussion, we will be focused on the speed of our implementation and note several improvements. We observed that the first steps of our implementation, namely computing approximate nearest neighbors, weights and initial embedding, are not critical, taking around 30 seconds for a dataset of size 30000, k=30 and 200 training epochs. By comparison, the final optimization step alone takes around 33 seconds. Because this step is already highly optimized with Cython (all critical code has been converted to efficient C/C++ code), we believe that further performance improvement hinges on parallelized stochastic gradient descent and more efficient sampling. For parallelization note that the for loop over all 1-simplices writes to a unique 1-simplex and can be trivially parallelized. For efficient sampling, we note that in our current implementation each 1-simplex consumes 1~6 random numbers regardless of its sampling probability. For a total of 200 training e-pochs a 1-simplex with a probability of 0.01 should roughly get sampled only 2 times. Thus, computing which data to sample beforehand would tremendously reduce the overhead of random numbers. umap simplifies this idea even further and distributes each sample evenly along the epochs (that is, the above-mentioned 1-simplex will be sampled exactly at epochs 100 and 200). However, negative sampling cannot be trivially distributed in this way, because each positive sampling needs to be paired with corresponding negative sampling. For this reason, negative sampling is unchanged. In order to test our idea, we implemented a rough $1/p$-epoch evenly distributed sampling approach, and the running time is comparable to the official umap package in single-threaded mode (27s vs 23s), confirming our hypothesis. However, we decided not to release this feature for the sake of robustness with a wider range of datasets, and also because the improvement is not dramatic.

References

BECHT, E., MCINNES, L., HEALY, J., DUTERTRE, C.-A., KWOK, I. W. H., NG, L. G., GINHOUX, F. AND NEWELL, E. W. (2019) 'Dimensionality reduction for visualizing single-cell data using UMAP', *Nature Biotechnology*, 37(1), pp. 38–44. doi: 10.1038/nbt.4314.

HAN, X. *ET AL.* (2018) 'Mapping the Mouse Cell Atlas by Microwell-Seq', *Cell*, 172(5), pp. 1091-1107.e17. doi: https://doi.org/10.1016/j.cell.2018.02.001.

HWANG, B., LEE, J. H. AND BANG, D. (2018) 'Single-cell RNA sequencing technologies and bioinformatics pipelines.', *Experimental & molecular medicine*, 50(8), pp. 1–14. doi: 10.1038/s12276-018-0071-8.

MCINNES, L., HEALY, J. AND MELVILLE, J. (2018) 'Umap: Uniform manifold approximation and projection for dimension reduction', *arXiv preprint arXiv:1802.03426*.

Code and data availability

All source code of `miniumap`, code for analyzing the performance, two datasets and their corresponding results were submitted in the code dropbox. Please note that the MCA dataset that we submitted is the subseted one. The original read count matrix can be found on https://figshare.com/s/9c3a0136f12b97f1dadd (Han_400k/rdata/pca.RData) (Becht *et al.*, 2019).