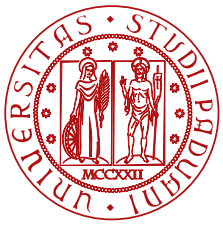


UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN COMPUTER ENGINEERING

Fast and accurate manifold learning using approximate nearest neighbor search

Relatore
Prof. Ceccarello Matteo

Laureando
Foti Giovanni

ANNO ACCADEMICO 2023-2024

Data di laurea 05/12/2024

Abstract

Dataset embedding and visualization are crucial tasks in machine learning as they allow to gather information about the data that can be leveraged to design downstream systems. UMAP is a commonly used dimensionality reduction algorithm because of its ability to accurately represent both local and global structures of data. Its computational cost is composed by the cost for the construction of the k nearest neighbors graph and the cost for the layout optimization phase using Stochastic Gradient Descent. This work explores how using approximate k nearest neighbor search indexes can speed up the algorithm while preserving the quality of the embedding. We were able to achieve up to a 9x speedup in UMAP embeddings using approximate k-nearest neighbor search algorithms like HNSW, while preserving the quality of the embeddings for both visualization and downstream machine learning tasks.

Contents

Chapter 1

Introduction

In machine learning, dataset embedding and visualization are essential for understanding high-dimensional data and designing downstream systems. Dimensionality reduction techniques like Principal Component Analysis (PCA)[11], t-SNE[14], and UMAP[16] play a pivotal role by projecting high-dimensional data onto lower-dimensional spaces, making patterns, clusters, and relationships more interpretable. Among these methods, UMAP has emerged as a popular choice due to its speed and its ability to capture both local and global data structures.

Despite its strengths, UMAP faces computational challenges as the size and dimensionality of datasets increase. Its computational cost is dominated by two steps: the construction of the k-nearest neighbors (kNN) graph and the subsequent optimization of the low-dimensional embedding using Stochastic Gradient Descent (SGD). While highly accurate, these steps can become a bottleneck for large-scale datasets, limiting UMAP's applicability in real-world scenarios. This work investigates how leveraging approximate nearest neighbor (ANN) search methods can accelerate the kNN graph construction phase. ANN methods trade exactness for efficiency, offering the potential to significantly reduce computation time while preserving the quality of UMAP embeddings.

This thesis explores how approximate nearest neighbors search techniques can enhance UMAP's performance. Specifically, we provide an extension of UMAP that leverages the Faiss[7] nearest neighbors search library to speed up the construction of the kNN graph. We evaluate the trade-offs between computational efficiency and embedding quality across different Approximate Nearest Neighbor search indexes, parameter configurations and real world datasets.

The remainder of this thesis is structured as follows. Chapter 2 introduces the theoretical backgound, including graph theory, manifold learning, nearest neighbor search algorithms and the UMAP algorithm. Chapter 3 describes the changes performed to the original implementation of UMAP. Chapter 4 describes the preliminary tests conducted on UMAP, the experi-

mental setup, the experiment structure and the collected performance metrics used to evaluate the performance of the proposed implementation. Chapter 5 presents and discusses the results, highlighting the impact of the proposed method on computational cost and embedding quality, using both quantitative and qualitative measures. Finally, Chapter 6 summarizes the findings and proposes directions for future research.

Chapter 2

Background

2.1 Graph theory

Spectral Graph Theory An undirected Graph is an object $G(V,E)$ where $V = 1,...,n$ is the set of **nodes** and $E = \{ \{i,j\} : i,j \in V \}$ is the set of **edges**. A directed graph is an object $G(V,A)$ where $V = 1,...,n$ is the set of **nodes** and $A = \{(i,j) : i,j \in V\}$ is the set of **arcs**. Graphs are structures used to represent connections between nodes. The connections can be used to represent similarity between nodes. It is possible to assign to each edge/arc a weight w_{ij}. In the similarity interpretation, the weight can describe how strong the similarity between two nodes is. For each node in a graph, it is possible to define its neighborhood, which is the set of nodes adjacent (i.e. connected though an edge) to it.

The **adjacency matrix** of a graph is a common data structure used to represent the graph. The matrix $A \in \Re^{n \times n}$ is a square matrix where for each element $a_{ij} \in A$, $a_{ij} = w_{ij}$. Alternatively, each element $a_{ij} \in A$ is different from 0 if and only if the edge/arc $\langle i,j \rangle$ exists in the graph, that is if the nodes i,j are adjacent.

The **degree matrix** $D \in \Re^{n \times n}$ of a graph is a square matrix where each element $d_{ij} = deg(v_i) \iff i = j$, and zero otherwise. $deg(v_i)$ is the degree of node i which is the cardinality of its neighborhood.

The **Laplacian matrix** $L = D - A$ of the graph is obtained by subtracting the adjacency matrix and the degree matrix of a graph. As such, the Laplacian is a $n \times n$ real-valued symmetric matrix, all of its eigenvalues are real numbers and its n distinct eigenvectors are orthogonal to eachother. It can also be shown that the eigenvalues are all non-negative. The eigenvectors associated to the eigenvalues can be interpreted as follows: the eigenvalue 0 is strictly related to the connected components of the graph; in particular, the multiplicity of the 0 eigenvalue of the Laplacian corresponds to the number of connected components in the graph. Non-zero eigenvalue, are related to the structure of the neighborhood of each node. Larger eigenval-

ues/eigenvectors will correspond to a space where the distance between neighbors is maximized while smaller eigenvalues/eigenvectors will refer to a space where neighbors are mapped closer.

The 2-dimensional **Spectral Layout** of a graph is obtained by embedding the graph using the two smallest non-zero eigenvectors of the Laplacian matrix. This layout is often used because it encapsulates the neighborhood information in the representation. Let v_2, v_3 be the second and third smallest, non-zero eigenvector of the Laplacian matrix. We denote with $v_{j,i}$ the i-th component of vector v_j. Then, given a node $i \in V = \{1,...,n\}$, its two dimensional spectral embedding coordinates are $(v_{2,i},v_{3,i})$.

2.2 Dimensionality reduction

Dimensionality reduction is the task of learning a low dimensional representation for high dimensional data while trying to preserve its original structure. This task is useful both for representing data and to reduce complexity of datasets meant to be used for other tasks, such as machine learning algorithms. For the task of data visualization, the data is transformed from a high dimensionality space to a 2 or 3 dimensional space which is suitable for plotting. For machine learning tasks, dimensionality reduction techniques are applied to the input data with the hope of extracting the most important features/dimensions from the high dimensional space and thus obtaining a low dimensional representation that behaves well with the downstream machine learning algorithms, i.e. computations are performed faster, more numerical stability.

Linear dimensionality reduction techniques such as Principal Components Analysis aim to keep the low dimensional representation of dissimilar points farther apart. Among non-linear dimensionality reduction, there exist techniques such as t-SNE and UMAP aim to learn the low dimensional manifold the data lies on and thus prioritize maintaining intact the local structure of data, i.e. preserve the similarity between points which are close together in the high dimensional representation; in this class of dimensionality reduction techniques, there are also algorithms that preserve global distance metric among the manifold such as Isomap[23] while using non-linear methods.

Methods for dimensionality reduction can be divided as described in [24]: a first distinction can be made between methods that optimize a convex vs a non-convex function. Then, among those that optimize a convex function, it is possible to distinguish those that approach the problem solving the spectral eigenvalue decomposition on a full matrix vs a partial matrix. An example of non-convex full spectral method is Principal Components Analysis. Manifold learning methods such as UMAP and t-SNE, which are also employed for dimensionality reduction, are instead examples of non-convex sparse-spectral nonlinear techniques. This distinction can be seen also by how each method performs the spectral analysis of the pairwise similarity

matrix of the data: methods like PCA perform the eigenvalue decomposition of the full matrix, while methods that focus on keeping intact the local structure of the data often solve the sparse or generalized eigenvalue decomposition problem.

2.3 Manifold learning

Manifolds Manifolds are mathematical objects that expand the notions of surfaces and curves to an arbitrary number of dimensions. In particular, an n-dimensional **manifold** M is a topological space which locally resembles the n-dimensional Euclidean space. In practice, most interesting usages of manifolds regard smooth manifolds.

To define smooth manifolds we need the notion of a chart. A **chart** is a pair (U,ϕ) where U is an open subset of M, the manifold, and ϕ is a function (homeomorphism) from U to $\Re^n$. Given two charts (U,ϕ), (V,ψ), it is possible to transition from one to another if $U \cap V \neq \emptyset$ using the composite map $\psi \circ \phi^{-1} : \phi(U \cup V) \to \psi(U \cup V)$, which is called the transition map from (U,ϕ) to (V,ψ). If this map is smooth (infinitely differentiable, C^∞), the two charts are said smoothly compatible. An **atlas** A is a collection of charts whose domain covers M. If, all the combination of pairs of charts (U,ϕ), (V,ψ) in A are smoothly compatible, A is said to be a smooth atlas. If A is maximal, that is, any other possible chart smoothly compatible with all charts in A is already in A, the pair (M,A) is said to be a **Smooth manifold**. Smooth manifolds that also admit a family of inner-products (metric tensor) on the tangent spaces of the points are called **Riemannian manifolds**. The family of inner products is called the Riemann metric and are usually smooth maps. We can say a Riemannian manifold is a generalized n-dimensional surfaces which is somewhat regular, is locally euclidean and has a notion of distance between points (the Riemannian metric).

Manifold learning refers to non-linear dimensionality reduction techniques that assume that the source data has been sampled from a smooth n-dimensional latent manifold where it lies on or is close to. Sometimes manifold learning refers to any dimensionality reduction technique that learns the geometric and topological properties of a manifold.

Manifold learning techniques differ from common (linear) dimensionality reduction algorithm as they generalize the problem to any manifold that present structures more complex than flat euclidean surfaces. To solve this task, more complex, non-linear transformations are used such as non-linear kernels, neural networks or using probability distributions. A crucial aspect in manifold learning is choosing to preserve either the local structure of the data, or the global structure. To conserve local structure, algorithms have to preserve similarities between points close together in the high dimensional space, for instance by solving a sparse spectral decomposition problem over the chosen similarity metric. An example of this type of algorithm is t-SNE[14]

which converts the distance between points into probability distributions and tries to minimize the difference between the high dimensional and low dimensional representation. In turn, to preserve global structure, i.e. keep low dimensional representation of dissimilar points apart, algorithms need to preserve the similarity metric across the whole manifold as does Isomap, which preserves the geodesic distances of all the points in the dataset.

2.4 Nearest Neighbors Search

Given a dataset V of d-dimensional points or vectors, the k Nearest Neighbors Graph (k-NN Graph) is the graph $G = (V,A)$ formed for each node v by k edges that connect v to the k most similar other nodes of V under a given similarity metric.

Common metrics include Euclidean or L2 distance $d(p,q) = ||p - q||_2$, where $||v||_2 = \sum_{i=1}^{n} v_i^2$, cosine similarity $s_{cos}(p,q) = cos\theta = \frac{p \cdot q}{||p||_2 \cdot ||q||_2}$, inner product similarity $s_{IP}(p,q) = \langle p,q \rangle$ among others. If the vectors are L2-normalized, the inner product and the cosine distance become equivalent. Furthermore, since the euclidean distance of L2-normalized vectors becomes $d_{L2}(p,q) = 2 - 2 \cdot s_{cos}(p,q)$, minimizing euclidean distance corresponds to maximizing cosine similarity.

The simple naive algorithm for building k-nearest neighbors graph requires the calculation of all n^2 distances in the dataset, which needs $O(d \cdot n^2)$ time factoring in the search of the k smallest distances for each point. Tree based algorithms such as K-D Tree[1] or Ball Tree[22], can reduce the complexity to $O(d \cdot n \log n)$ (under certain assumptions on the underlying structure of the data) which improve the performance in most practical cases but are still potentially very costly for high-dimensional datasets with a large number of samples.

As dataset size increases, both in the number of samples and in the number of dimensions, resorting to Approximate Nearest Neighbor(ANN) search algorithms yields significant improvements for time and space utilization. This class of techniques sacrifices accuracy with the goal of obtaining a more efficient use of resources. Common approaches include Locality Sensitive Hashing [10][5], tree-based data structures as used in [19], inverted file non-exhaustive search[7], graph based approaches like Hierarchical Navigable Small Worlds(HNSW)[15], Navigable Spreading-out Graphs(NSG)[9] or NN-Descent[6].

2.4.1 Faiss

Faiss[7] is an open-source library for efficient similarity search algorithms developed by Meta's Fundamental AI Research group. It is written is C++ and provides Python bindings for most of its functionalities. It provides a comprehensive suite of algorithms and data structures tailored for large-scale datasets, supporting both exact and approximate (non-exhaustive) searches. It

fully supports euclidean distance and inner product similarity for dense vectors. Faiss is built around the Index object which encapsulates the database, the search algorithm and data structures leveraged.

For large datasets, when non-exhaustive search is to be preferred, Faiss implements Inverted File Indexes and graph based indexes. These two approaches focus on a subset of the dataset to speed up the search, while making different trade offs with regards to memory usage and speed.

Inverted File Indexes Inverted File (in short IVF) indexing is a technique that clusters the dataset vectors at indexing time. The clustering uses a vector quantizer that outputs `nlist` different indexes. Each index stores its cluster's vectors, eventually compressed, in an inverted list, thus forming the global inverted file (IVF) structure. An higher value of `nlist` allows for finer partitioning of the dataset, lowering the search time for each list since it will have fewer elements but increasing the index size. At search time only a subset of the `nlist` indexes are searched. This value is set in Faiss using the `nprobe` parameter which dictates the number of indexes closest to the cluster of the queried point to be checked. Increasing this parameter leads to longer search times but improves the recall of the index.

Inverted File Product Quantization (IVF-PQ) indexes are inverted file indexes which use a Product Quantizer [12] to encode the vectors. A quantizer is a compression method that takes in input a d-dimensional vector and outputs an $nbits$ number(i.e. a natural number from 1 to 2^{nbits}). In particular, a Product Quantizer is a quantizer which takes in input a vector, it splits it into M_{pq} sub-vectors and encodes each sub-vector separately into an $nbits$ number. For instance, a Faiss `PQ32x4` quantizer, compresses a d-dimensional vector to a 32-dimensional vector of 4-bits numbers. Product quantization allows to reduce the memory usage of the index and to reduce query time.

Faiss also includes an optimized version of Product Quantization which uses SIMD CPU instruction to allow fast accumulation of the codes at indexing and search times A fast scan product quantizer is identified by a string such as `PQ32x4fs`. This method is limited by the fact that the value M_{pq} for the number of sub-vectors has to be a factor of the dimensionality d of the vectors.

Hierarchical Navigable Small Worlds Among graph based indexes Faiss contains an implementation of the Hierarchical Navigable Small World [15] (HNSW) index. HNSW is a graph-based data structure designed to facilitate efficient approximate nearest neighbor searches in high-dimensional spaces. It constructs a multi-layered graph where each node represents a data point, and edges connect nodes based on proximity. The hierarchical organization allows for efficient navigation during search queries, balancing search speed, accuracy, and memory consumption. Key parameters for HNSW are:

- **M**: the number of birectional links for each new element during construction. Higher values of M increase accuracy but also memory usage since the graph structure to store becomes more complex.

- **efConstruction**: The size of the dynamic list for the nearest neighbors during the graph's construction phase. Larger values lead to more accurate graphs but require more computational resources during the initialization of the index.

- **efSearch**: The size of the dynamic list for the nearest neighbors during the search phase. Increasing efSearch improves recall at the cost of higher query time.

In the following we will refer to HNSW indexes like HNSW32 to indicate an HNSW faiss index with M = 32.

2.4.2 NNDescent

NNDescent[6] is a graph-based algorithm for constructing approximate k-nearest neighbor graphs efficiently. It operates on the principle that a neighbor of a neighbor is likely to be a neighbor, iteratively refining the graph by exploring neighbors of current neighbors. It is implemented in the PyNNDescent Python library.

2.5 UMAP

UMAP (Uniform Manifold Approximation and Projection for Dimension Reduction) is an algorithm for dimensionality reduction designed to be fast and efficient. It allows many different tasks, such as clustering, visualization, dimensionality reduction and is usable as an intermediate step for data analysis tasks.

UMAP is a non-linear dimensionality reduction algorithm, in particular a manifold learning technique. The UMAP algorithm is composed of two phases: construction of a fuzzy topological representation and optimization of the embedding. Computationally, this corresponds to the construction of an undirected graph, the computation of an initial layout, and an iterative optimization of this layout.

2.5.1 Theoretical Background: Simplicial Complexes and Fuzzy Topology

To model the topological structure of the high-dimensional data, UMAP relies on simplicial complexes and fuzzy topological theory. Simplicial complexes are structures used to construct topological spaces using components called simplices. A k-dimensional simplex is the simplest

k-dimensional polyhedron, which is the convex hull of its $k + 1$ vertices (see Figure 2.1: 0-simplex: point, 1-simplex: line, 2-simplex: triangle, 3-simplex: tetrahedron).

Simplicial complexes are obtained from a set of simplices by joining together their faces. Simplices and simplicial complexes can be used in fuzzy topology as building blocks for topologies and topological objects. Fuzzy topology is an extension of classical topology which uses fuzzy logic to allow degrees of membership of objects to sets. A fuzzy simplicial complex is a simplicial complex where the strength of the bond that keeps the simplices together is expressed by their membership strengths. In practice, this means that it's possible to describe mathematically a topological structure by means of simple objects (points, lines, triangles) glued together with a certain strength which can express any characteristic of interest of the space we are studying.

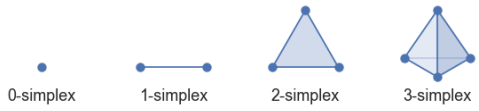


Figure 2.1: Low dimensional simplices

2.5.2 UMAP Assumptions

UMAP is designed around some theoretical assumptions. It assumes that the data is uniformly sampled from a Riemannian manifold, which is locally approximately constant (i.e., within small neighborhoods, the manifold is approximately uniform) and locally connected (i.e., any two points in the neighborhood can be connected by a path). UMAP also assumes the manifold to have a latent dimensionality smaller than the high dimensionality of the source dataset. Thus, learning an n-dimensional representation of the manifold corresponds to learning an n-dimensional representation of the source data.

2.5.3 UMAP Algorithm Overview

UMAP begins by constructing a graphical representation of the data using fuzzy simplicial complexes. Points sufficiently close together become part of the same simplicial complex, formed by the points and the segments that connect them. This type of simplicial complex is known as a Vietoris-Rips complex [25] and is a known tool in topological data analysis [3][4].

The UMAP algorithm then proceeds to optimize the low-dimensional embedding by minimizing an objective function that measures the difference between the high-dimensional and

low-dimensional representations. The optimization is performed using Stochastic Gradient Descent (SGD).

2.5.4 Algorithm Details

Fuzzy Topological Representation Construction

UMAP starts building the initial graph representation from the neighborhoods of the points. Points are considered close together by using two parameters for the algorithm: the number of neighbors and the minimum distance. The number of neighbors defines the cardinality of the set of points closest to each point. Over this set is defined the local metric for the neighborhood of each point, as a unit ball centered at the point that spans to the farthest neighbor. The minimum distance is a global parameter that defines the local connectivity threshold, that is, the cut-off distance after which two points are not connected. Choosing a higher number of neighbors considers local sets of points more numerous, thus increasing the definition of the local representation. Choosing a higher minimum distance sets a higher threshold for local connectivity, lowering the likelihood that points or neighborhoods are isolated from one another.

UMAP relies on fuzzy topological theory and fuzzy simplicial complexes to approximate the manifold in the following way: (fuzzy) simplices of up to order 2 (e.g., points, lines, and triangles) are considered for the neighborhood of each point. The membership strength of the fuzzy simplicial complexes built from the simplices is obtained by evaluating the distances between the points they consider, which is relatively cheap since a limited number of neighbors is considered. In practice, UMAP uses the 1**-skeleton** of the fuzzy simplices. The 1**-skeleton** of a topological space X (i.e., the manifold) represented using simplicial complexes is the union of the topological simplices of up to order 1 (i.e., points and lines). This procedure creates a topological graph with edges weighted according to the membership strength of simplices.

The fuzzy topological representation is built starting from the creation of a local metric space for each point. Using fuzzy simplicial set theory, each point belongs to a local set (a neighborhood) with a probability calculated through a membership strength function:

$$\mu(i,j) = \exp\left(-\frac{d(i,j) - \rho_i}{\sigma_i}\right)$$

where:

- $d(i,j)$ is the distance between sample i and nearest neighbor j, typically the Euclidean distance.
- ρ_i is the local connectivity adjustment for sample i, ensuring connectivity.

- σ_i is the normalization factor derived from the metric tensor approximation for sample i.

For each sample i, the local connectivity adjustment involves defining a distance ρ_i as:

$$\rho_i = \min\{d(x_i, x_{ij}) \mid 1 \leq j \leq k, d(x_i, x_{ij}) > 0\}$$

This ensures that each sample i is connected to at least one other sample with an edge weight of 1.

Optimization of the Embedding

The layout obtained in the previous step is initialized using the spectral layout of the graph (see Section 2.1). This encapsulates the neighborhood information from the high-dimensional representation and is further optimized in the low-dimensional space.

The optimization step uses Stochastic Gradient Descent (SGD) to minimize the following cross-entropy objective function:

$$\mathcal{L} = \sum_{e \in \mathcal{E}} w_h(e) \log\left(\frac{w_h(e)}{w_l(e)}\right) + (1 - w_h(e)) \log\left(\frac{1 - w_h(e)}{1 - w_l(e)}\right)$$

where $w_h(e)$ and $w_l(e)$ are the weights of the edges in the high-dimensional and low-dimensional representations, respectively. These weights are derived from the membership strength $\mu(i, j)$.

The layout combines spectral initialization with a pseudo force-directed layout, where attractive and repulsive forces are determined by the distances in the high-dimensional space. These forces are approximated smoothly to allow optimization via SGD.

2.5.5 The ‘umap-learn‘ implementation

UMAP is implemented in the `umap-learn` Python library, written with numba, numpy and scipy. It follows the Scikit-learn estimator specification. The standard use case is as follows:

1. Load the dataset(s), as numpy array or similar

2. Construct a UMAP object providing the desired parameters such as number of neighbors, min distance, number of components (i.e. dimension of the embedding), distance metric. UMAP is an object compatible with scikit-learn models.

3. From the UMAP-model object, call the desired method among for example `fit`, `transform`, `fit_transform`, passing the dataset as argument.

Construction of the UMAP object When the UMAP object is instantiated, the arguments and parameters of the algorithm are checked and assigned. The initialization accepts many parameters: some like the number of nearest neighbors or the minimum distance control the output of the algorithm, others like the parameter low_memory or the number of jobs control the computational behaviour or how the execution will be carried out in practice.

Dataset fitting and transformation methods Transform methods always check the data then articulate the calculation based on the arguments passed to the constructor of the UMAP object.

Simplicial set embedding implementation This step is implemented in the `simplicial_set_embedding` function. It receives in input the source data to be embedded and the 1-skeleton of the high dimensional fuzzy simplicial set as represented by a graph's adjacency matrix, a sparse matrix. Other parameters include the number of components which is the dimensionality of the euclidean space into which to embed the data, initial parameters for SGD and for the functor approximation, the number of epochs, the initialization method of the low dimensional embedding and the metric used to measure distance in high dimensional space.

A pre initialization step is performed: weights in the graph matrix are set to 0 (i.e. edges are pruned) by removing entries where the weight is lower than the ration between the max weight in the matrix and the number of epochs.

The initialization step is performed as follows: if an initialization method is provided, then the embedding is generated based on this method (e.g. spectral layout, PCA, TruncatedSVD for sparse data) and some noise is added to it; if an initial embedding is provided, it becomes the initial embedding.

Before starting the optimization procedure, the "epochs per sample" are calculated. This is then used in the layout optimization function.

After evaluating the input data form (e.g. if densmap or output dens) and normalizing the embedding data, the layout optimization function is called.

Layout optimization function This method uses stochastic gradient descent to optimize the objective function. The implementation uses numba to parallelize the execution of each epoch. The algorithm defines the number of epochs based on both the input parameter for UMAP and on other parameters such as if the output has to be a densmap. Then SGD uses the distance metric to compute the objective function and the gradient and iterates for the specified number of epochs. In this step negative sampling is also employed.

Possible speed-ups for UMAP Cornerstone of the UMAP algorithm is the kNN index that is queried to get the nearest neighbors for each point. By default UMAP uses the NNDescent[6] algorithm for the calculation of this index, implemented in the PyNNDescent library. The time required for this calculation is significant and grows rapidly with dataset size as can be observed in Section 4.1, thus employing a different algorithm for the calculation of the kNN index is a possible way of reducing the computational cost of UMAP.

The algorithm uses the tree structure built by NNDescent to reduce the time to obtain the kNN of each point. Candidate algorithms should have a similar data structure which minimizes both the time required to build the index and the time to query the points.

Cost Analysis of the Phases The cost of the first phase of UMAP lies with the calculation of the kNN index. The cost of the second phase, the optimization of the layout, is divided between the cost for the initialization of the layout and the cost for the optimization with stochastic gradient descent. The duration of the second part is mostly tied to the number of epochs used for the optimization, which can be either set manually in the initialization of UMAP or is automatically decided by the algorithm based on the other parameters. In contrast, the duration of the initialization of the layout depends on the implementation of the layout algorithm. For the spectral layout, which is the default choice, the scipy library is used to solve the sparse eigenvalue decomposition problem and initialize the graph layout.

Details about execution times of UMAP and its phases are provided in section 4.1.

Chapter 3

Methodology

In this chapter we will provide the details of the changes performed to the `umap-learn` Python package. This work extends the original implementation of UMAP[1], stemming from the release 0.5.6.

The goal of these changes is to allow the usage of approximate nearest neighbor search algorithms different from PyNNDescent. In particular, this implementation uses the Faiss[7] Python library to access optimized implementations of many Nearest Neighbor search algorithms and indexes.

A repository containing the source code of the implementation is available on GitHub[2].

3.1 The GeneralizedIndex class

The proposed implementation introduces a new class internal to UMAP, `GeneralizedIndex`, defined in the `indexes.py` file in umap. It provides an adaptable nearest neighbor search index interface which allows the usage of both the Faiss and PyNNDescent libraries depending on the configuration provided. The class is composed by the constructor, which builds the nearest neighbors graph, and methods to query or update the index.

It is used in UMAP to compute the k nearest neighbors graph.

3.1.1 Constructor

The constructor (`__init__` method) of `GeneralizedIndex` accepts as parameters the dataset to index, the number of neighbors to search for and the configuration for the index to employ. In particular, the constructor checks the `use_pynndescent` parameter: if it equals `True` the behaviour of `GeneralizedIndex` is the same as a PyNNDescent object; if it equals `False` the

[1] `www.github.com/lmcinnes/umap`
[2] `www.github.com/gioffo/umap-faiss`

constructor builds a Faiss index based on the `faiss_index_factory_str` parameter, or a Faiss HNSW Flat index if this parameter is not provided.

Faiss Indexes The Faiss branch of the constructor allows the initialization of any Faiss index accepted by the `faiss.index_factory()` method, using the `faiss-cpu` implementation. This method parses a format string to define an index and allows for the creation of indexes of varying complexity. It is also possible to set additional search or construction parameters for the search index, such as `nprobe` for IVF indexes or `efSearch` and `efConstruction` for HNSW indexes, using the `faiss_kwds` parameter of `GeneralizedIndex`. The constructor supports the configuration of the metric to use for the indexing through the `metric` parameter. Faiss indexes primarily supports Euclidean (L2) distance and inner product, which is also adaptable to cosine similarity.

Time Profiling The `GeneralizedIndex` class incorporates basic profiling tools to measure and record the time spent in various stages of index creation and nearest neighbor graph construction. It has a `_times` dictionary that stores the time taken to initialize the index (`index_creation`), the time taken to build the nearest neighbors graph (`knn_construction`) and the total time (`total`). Since PyNNDescent handles both the index creation and the k nearest neighbors graph construction at once, only the total time is tracked in this case. For the Faiss indexes, the `index_creation` time includes the training phase of the index and is usually longer for indexes with more complex structures such as pre-normalization of the vectors, refinement stages, nested indexes. The `knn_construction` time consists in the time taken by the index to calculate the k nearest neighbors indices and distances for each one of the vectors in the dataset.

3.1.2 Methods

The `GeneralizedIndex` class includes implementations for the PyNNDescent methods used in UMAP in order to preserve all the functionalities provided by the library.

Query method The query method allows to retrieve distances and indices for the k nearest neighbors of vectors `X` in matrix form. If `GeneralizedIndex` is using NNDescent, the search parameter `epsilon` is set in accordance to the `angular_trees` parameter.

Update method The update method allows to update vectors in the index. This functionality is used by the analogous `update` UMAP method which changes the dataset and updates the embedding.

Prepare method This method is needed to guarantee compatibility with PyNNDescent. It is used to pre-compile `numba` functions and speed-up consequent calls. It has no effect for Faiss indexes.

3.2 UMAP changes

The UMAP code has been modified to make use of the `GeneralizedIndex` class for the nearest neighbors calculation. Specifically, in the `nearest_neighbors` function called by UMAP in `fit` and `fit_transform` processes, NNDescent has been replaced by a `GeneralizedIndex` object.

The parameters `faiss_index_factory_string` and `faiss_kwds` have been introduced in the UMAP class in order to pass them to the underlying classes. The former is the string used by the Faiss index factory to build the index. The latter is a dictionary that is used to set parameters for the index. The `use_pynndescent` parameter, already present in UMAP, has been used to implement the switching logic between NNDescent and the Faiss indexes.

Time profiling has been added to UMAP to log the time taken by the various phases, and to provide an access point to the execution time of the Faiss index. The `times` attribute of UMAP is a Python dictionary which contains the start and end timestamps of the nearest neighbors, fuzzy simplicial set computation and embedding optimization phases of UMAP.

Chapter 4

Experiments

4.1 UMAP Preliminary Tests

This section will focus on a preliminary analysis of the `fit_transform` method of UMAP, which, given a dataset, initializes UMAP and returns the embedding. As described in the previous section, this method fits the input dataset to the desired embedding space or dimensionality and returns the embedding. In an effort to study how each phase of this procedure scales with regard to parameters, dataset size and dimensionality, we ran the UMAP algorithm over a grid of parameters and datasets To capture the running times of the algorithm, the source code of UMAP has been modified by adding logging of execution times in an internal data structure, which is then available at the end of the execution.

4.1.1 Datasets

For the initial evaluation of UMAP, the following datasets have been used: the Real World Dataset collection[1] of the Scikit-Learn python package, used mainly for a preliminary evaluation of the impact of the parameters on the embedding output; the Fashion-MNIST dataset fetched using the `fetch_openml` method of scikit-learn and synthetic datasets `swiss_curve`, `u_shape`.

The tests for the evaluation of running times were conducted on the Fashion MNIST[26] dataset. This dataset contains 70'000 28x28 images of clothing items belonging to 10 classes. Since classes have not been used for this analysis and the images are composed of 28 by 28 pixels (integer values from 0 to 255), the dataset is represented as points of 784 dimensions. The dataset has been subsampled to different samples sizes and the tests have been run using the parameters as in Table 4.1.

[1] https://scikit-learn.org/stable/datasets/real_world.html

Table 4.1: Test runs parameters

Samples	Initialization	n_neighbors	min_dist
5000	'pca', 'random', 'spectral', 'tswspectral'	15, 25, 50, 75	0.1, 0.2, 0.3
6750	'pca', 'random', 'spectral', 'tswspectral'	15, 25, 50, 75	0.1, 0.2, 0.3
7500	'pca', 'random', 'spectral', 'tswspectral'	15, 25, 50, 75	0.1, 0.2, 0.3
8750	'random', 'spectral', 'tswspectral'	25, 75	0.1, 0.25
10000	'random', 'spectral', 'tswspectral'	25, 75	0.1, 0.25
11250	'random', 'spectral', 'tswspectral'	25, 75	0.1, 0.25
12500	'random', 'spectral', 'tswspectral'	25, 75	0.1, 0.25
13570	'random', 'spectral', 'tswspectral'	25, 75	0.1, 0.25
15000	'random', 'spectral', 'tswspectral'	25, 75	0.1, 0.25

4.1.2 Running times

During the tests, total and partial running times of UMAP have been gathered. In particular the following time measurements have been collected: duration of the creation of the Nearest Neighbors index; duration of the creation of the initial fuzzy simplicial set; duration of the optimization of the embedding. The end of the embedding optimization step coincides with the termination of the algorithm so it has also been used to capture the total running time of the algorithm.

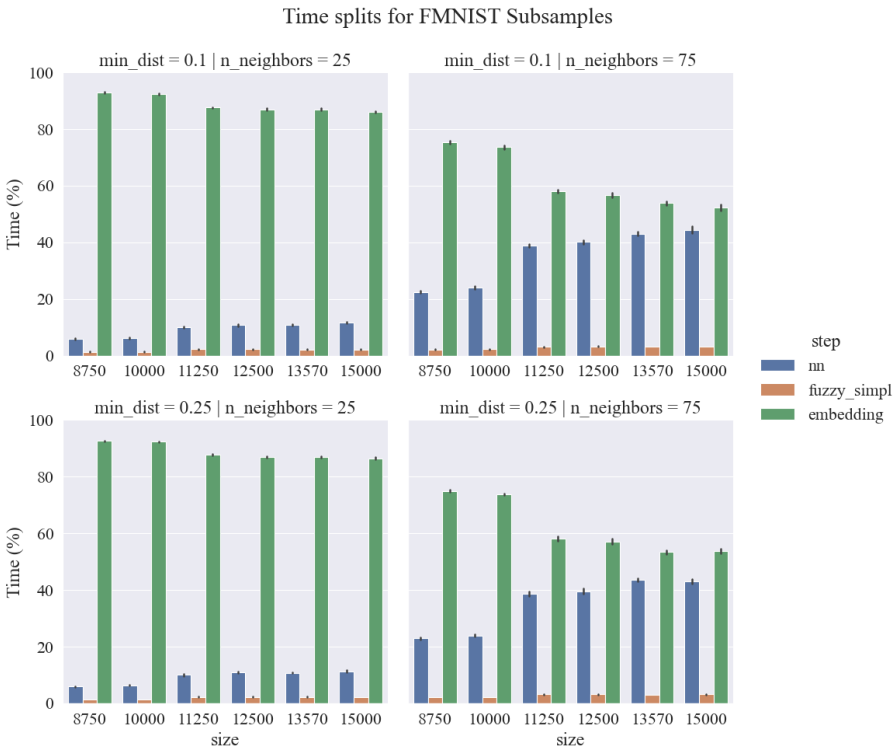
4.1.3 Observations

Figure 4.1 shows the relation between the phases duration, dataset size, number of neighbors and the minimum distances parameters. The x-axis represents the size of the dataset. The y-axis represents the fraction of total time, in percentage, taken by each step. We can observe that increasing the number of samples the time taken by the algorithm for the calculation of the nearest neighbors index gets longer with respect to the embedding calculation phase. A similar behaviour, altough to a more limited extent, can be seen for the simplicial set calculation phase, too. Also, increasing the number of nearest neighbors employed by the algorithm has a significant impact on the duration of the nearest neighbors calculation phase, reflecting on a steeper increase with larger datasets.

The trend for the total duration of the runs can be observed in Figure 4.2. The figure highlights how total execution time increases with the increasing dimension of the sample. Also it can be observed how the number of nearest neighbors directly impacts the execution time as expected.

In Figure 4.3 the effect of the initialization algorithm can be seen. The following initializa-

Figure 4.1: Run times split for Fashion-MNIST dataset samples.



tion methods have been tested: Principal Component Analysis (PCA) uses the first n components from PCA applied to the input data, where n is the target dimensionality; spectral uses a spectral embedding of the fuzzy 1-skeleton, which is the default initialization method; random uses random initial position; `tswspectral` uses a truncated singular value decomposition to try to speed-up the eigensolver for the spectral embedding of the 1-skeleton, which can take a long time in certain cases. PCA was the slower method in this test, followed by the `tswspectral` method, while the random method was the fastest. It can be observed that, despite starting with a random embedding, the random method converged still faster to the optimal embedding than the spectral initialization.

4.2 Datasets

In this chapter we examine the performance of the proposed implementation on some real world datasets. We study the recall of the nearest neighbor search index used by UMAP, the execution time of the phases, the Procrustes disalignment of the embedding with respect to a full-recall UMAP execution and the loss of a k-nearest neighbors classifier run on the resulting embedding. We compare these measures between faiss indexes and the NNDescent standard nearest neighbor search index, using the proposed implementation. The following datasets were studied:

Figure 4.2: Total run times for Fashion-MNIST dataset samples.

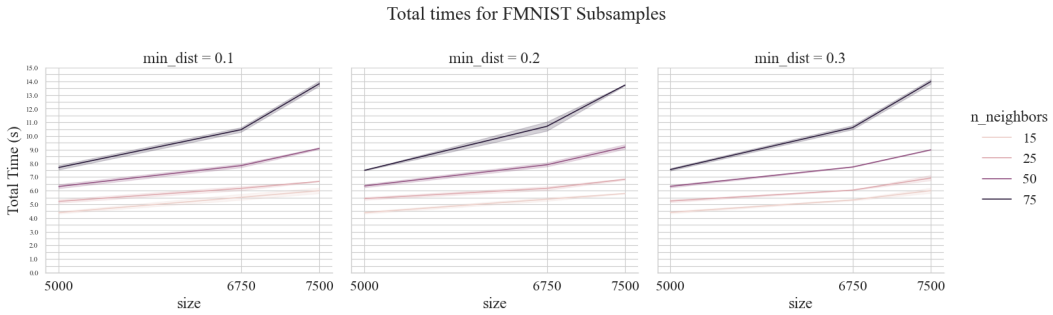
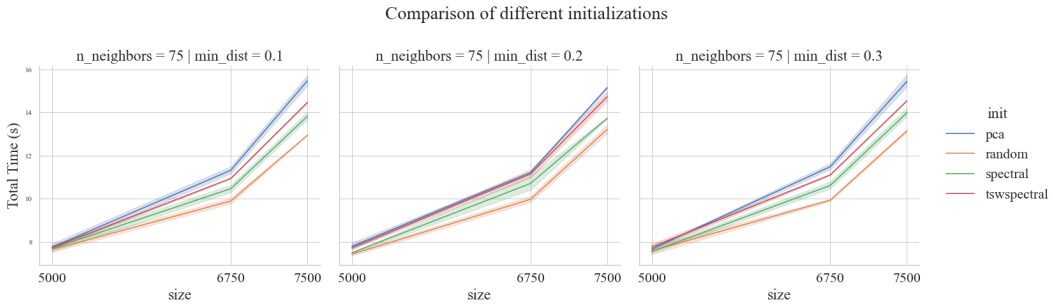


Figure 4.3: Comparison of total run times on FMNIST for different initializations, using 75 as number of neighbors.



- **COIL-100**[20] is a collection of 7200 color images of 100 objects recorded at 72 different angles. Every image consists of a 128×128 intensity matrix for each of the 3 color channels. The resulting datapoint considered is a 45152 dimensional vector.

- **Mouse scRNA**[2] is a collection of profiled gene expression of mouse hypothalamic cells. It consists of 20921 vectors with 26774 components.

- **Statlog (shuttle)**[8] is a set of 58000×9 dimensional vectors that represent the status of the NASA space shuttle components along with their classes.

- **MNIST**[13] is a collection of 28×28 greyscale images of handwritten digits. The dataset consists of 70000 images which are used as 784 dimensional vectors.

- **FMNIST**[26] (Fashion MNIST) is a collection of 28×28 greyscale images of fashion items. As for MNIST, it contains images of items belonging to 10 different classes and 70000 total images treated as 784 dimensional vectors.

- **GoogleNews word vectors**[18] is a dataset of 3 million phrases obtained from Google News articles and embedded into a 300 dimensional space using *word2vec*[17]. We use a sample of 500'000 vectors from this collection.

For all the datasets minus Google News we used the Euclidean distance to compute the embedding. For Google News we used the cosine distance.

4.3 Experiment Structure and Measures

We tested the proposed implementation by instantiating the UMAP object from scratch and computing the 2-dimensional embedding using the `fit_transform` method of UMAP. This procedure was repeated 5 times for each combination of parameters. From the UMAP object we collected the embedding, the execution times data and the k nearest neighbors indices. We compare the k nearest neighbors indices with the indices computed by a Faiss Flat index and calculate average recall values and distribution.

Index Recall As the measure of accuracy of the nearest neighbor search algorithm we consider the k-recall@k metric. This corresponds to the fraction of the first k nearest neighbors retrieved by the index which are also true nearest neighbors, that is those who are also part of the set of the first k true nearest neighbors according to a perfect recall ranking of neighbors.

For this measure we study both the average value across all the points and the distribution of recall values.

Procrustes Distance Given two embeddings X and Y of size n, such that x_i corresponds to y_i, we define Y' as the optimal rotation, translation and rescaling of Y with respect to X such that the squared error $\sum_{i=1}^{n}(x_i - y_i')^2$ is minimized. This transformation is called Procrustes superimposition and the Procrustes distance is defined as the square root of the error.

An example of Procrustes superimposition can be observed in Figure 4.4, which shows two embedding of the COIL-20[21] dataset before and after the Procrustes superimposition. The first embedding is scaled to the unary ball, the second embedding is transformed to minimize the distance from the corresponding points in the first embedding. We can observe how both local and global structures of the second embed are not altered after the transformation, and how the embed has been rotated to move the loops closer to their position in the first embedding.

Figure 4.5 shows how the value of the Procrustes distance for a given dataset has by itself limited meaning. This example shows four UMAP embeddings, with the same parameters, of the COIL-20 dataset, superimposed to the reference embedding in the top left corner. All the embeddings exhibit the characteristic loops but the arrangement is different. This is reflected in the value of the Procrustes distance which punishes embeddings that move loops farther away from the reference embedding. This behaviour motivated us to look at this measure from a dataset oriented perspective and study the distribution of the captured values against some reference

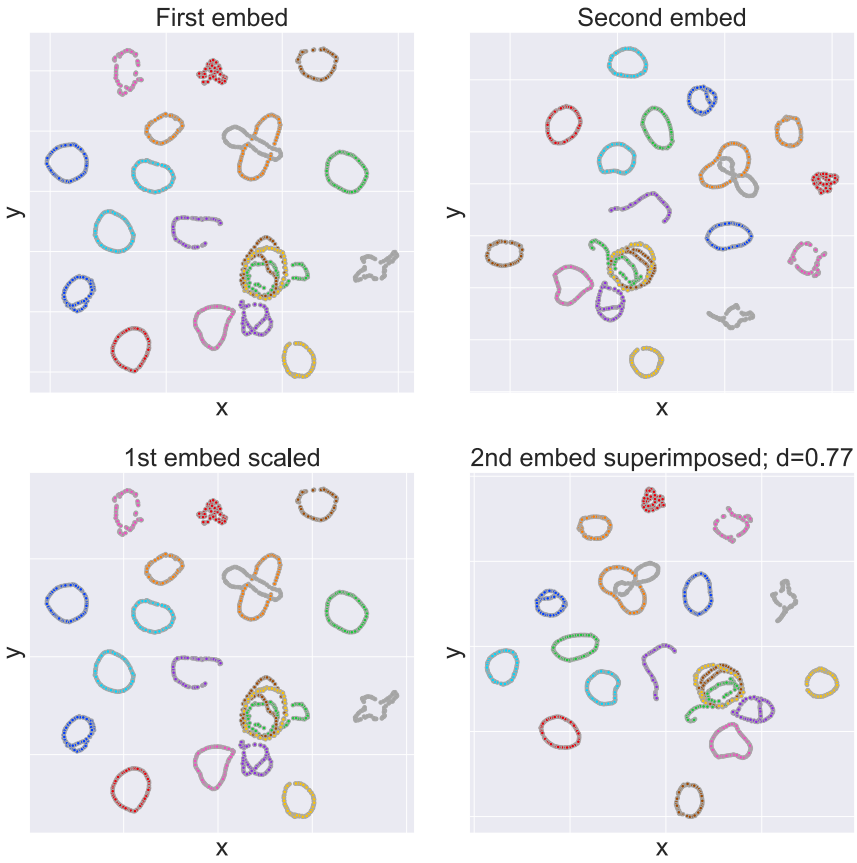


Figure 4.4: UMAP embeddings of COIL-20[21] before and after Procrustes superimposition

embeddings, such as those generated using NNDescent or those generated using a full-recall index.

Taking the embedding generated under a given index configuration as Y and a "ground-truth" embedding generated by UMAP using an index with 100% K-recall@K as X, we measured the Procrustes distance between the two embeddings and compared it to the distribution of the runs with the same UMAP parameters on each dataset.

We used the Procrustes distance implementation found in `scipy.spatial` Python package, which uses Singular Value Decomposition to solve the superimposition problem.

4.4 Parameter Grid

A total of 11520 embeddings under different parameters were computed. For each of the 2305 setups we repeated the experiment 5 times. Up to 394 combinations of parameters were tested for each dataset, split between parameters for the UMAP embedding, like the number of neighbors and the minimum distance, and parameters for the inner nearest neighbors search index. A

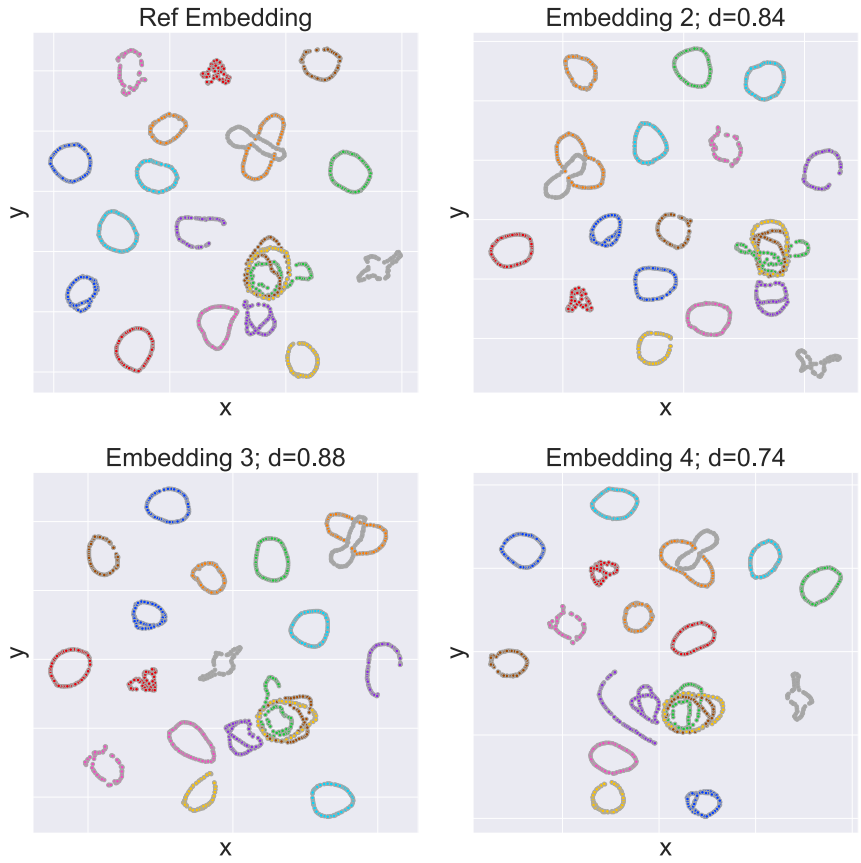


Figure 4.5: UMAP embeddings of COIL-20[21] that show how the magnitude of the Procrustes distance is dependent on the dataset. In this case we could be lead to accept embeddings with Procrustes distance between 0.7 and 0.9 as sufficiently similar.

comprehensive list of parameters tested is available in Table 4.2.

The values chosen for the `n_neighbors` UMAP parameter cover low and high neighbors case, in order to explore how the performance of the studied indexes changes. Since the parameter `min_dist` didn't show a significant impact on the embedding time in our preliminary analysis, we decided to perform the experiment using the default value of 0.1.

For the HNSW class of indexes we produced a grid for the parameters `M`, `efSearch`, `efConstruction`.

The grid for the IVF-PQ indexes included the construction parameter `nlist`, the search parameter `nprobe` and the Product Quantization parameter `m_pq`. The parameter `m_pq` needs to be a factor of the number of dimensions `m` of the dataset. We choose the values for each dataset by first extracting all the divisors of `m` greater or equal than 2. If less than 10 values are found, we choose the last 3. If more are found, a logarithmic sampling strategy is applied, using powers of 2 to partition the set of values and choosing the factors at the 3 largest positions. This heuristic guarantees an evenly spaced grid of values even for datasets with many dimensions

Table 4.2: Parameters grid

Parameter	Scope	Values
n_neighbors	UMAP	{25, 75}
min_dist	UMAP	{0.1}
M	faiss-hnsw	{4, 16, 24, 32, 64}
efConstruction	faiss-hnsw	{10, 20, 40, 80}
efSearch	faiss-hnsw	{1, 4, 8, 16, 32, 64, 128, 256}
nlist	faiss-ivfpq	$\{\sqrt{n}, 2 \cdot \sqrt{n}, 4 \cdot \sqrt{n}\}$
nprobe	faiss-ivfpq	{1, 16, 32, 100}
m_pq	faiss-ivfpq	1 to 3 log-spaced values

such as COIL-100.

4.5 Experimental setup

The experiments were run on a single node of the Lovelace cluster with two AMD EPYC 7282 16-cores processors and 494 GB of RAM. The execution environment was an Apptainer container running Ubuntu 22.04.4 LTS. The Python environment was created using mamba and ran Python 3.11.9, faiss-cpu 1.8.0 and the modified version of UMAP 0.5.6.

Chapter 5

Results

In this chapter we will discuss the results obtained in the experiments of the previous section. For each configuration we will consider the average values of recall, execution time(s) and Procrustes distance across 5 runs. We will analyze the relationship between index parameters, UMAP execution time and index recall, the quantitative performance, the qualitative performance and the stability of the embedding.

5.1 Execution Time and Recall Analysis

In this section we will discuss how the proposed implementation performed on each dataset. We will study the performance of the computation of an UMAP embedding using a Faiss Index such as HNSW or IVF-PQ as k nearest neighbors search index and compare it against the NNDescent baseline. We will analyze the execution time of the algorithm and its relation with the Index recall and the parameters configuration.

5.1.1 Dataset specific

This section will provide an overview of the results obtained on each dataset. First, we will discuss the distribution of the runs to understand possible dataset specific behaviors and limits and provide a first comparison with the NNDescent baseline. Then, we will study the best performing runs according to a recall lower bound and a speedup value for the execution time. For a given k-recall@k threshold R, we will highlight the configuration that achieves average recall greater than R with the lowest total UMAP execution time, and calculate the speedup as the ratio between the execution time of NNDescent and using the index. The goal is twofold: to examine how speedup varies under bounded recall and to identify the index classes that offer the best trade-off between speed and accuracy.

As underlined in the Section 4.1, higher number of nearest neighbors leads to a longer embedding time, due to the fact that the complexity of the nearest neighbors calculation phase increases. For this reason we split up the analysis according to the number of nearest neighbors used by the algorithm.

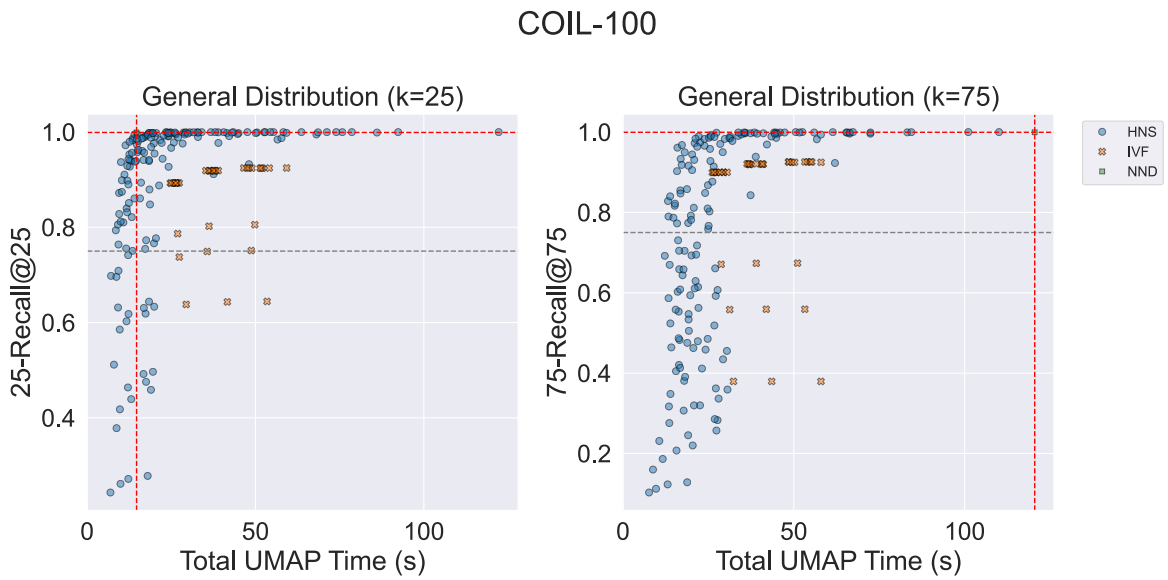


Figure 5.1: Coil-100 runs using min_dist=0.1, n_neighbors={25,75} for the different index classes tested. On the vertical axis figures the average k-recall@k (higher is better), on the horizontal axis the total UMAP execution time in seconds(lower is better). Red dotted lines signal the average recall and runtime of UMAP using NNDescent.

COIL-100 Figure 5.1 shows the distribution of runs under the different index classes tested for the COIL-100 dataset using min_dist=0.1 and n_neighbors={25,75}. Tables 5.1 and 5.2 present a closer analysis of the best performing runs in the grid by varying the minimum recall threshold. The coordinates of the NNDescent baseline run are located by the intersection of the red dotted lines.

When the number of nearest neighbors considered by the embedding is low (n_neighbors=25), we can observe that NNDescent obtains near perfect average recall and allows UMAP to compute the embedding in less time than most of the HNSW configurations with comparable recall, although some configurations are capable of obtaining similar performance with a lower time cost. In particular, as can be seen in Table 5.1, indexes such as HNSW with M = 16, efConstruction = 40, and efSearch = 32 are able to match NNDescent performance. Lighter indexes configurations such as HNSW with M = 4 become competitive at lower recall thresholds, allowing UMAP to complete 70% faster with around 80% 25-recall@25. While the HNSW runs are concentrated around the NNDescent result, we

Recall Th.	UMAP Time	Speedup	Avg. Recall	Index Key
-	14.68	1.00	0.999	NNDescent
0.99	14.05	1.04	0.994	HNSW16_C40_S32
0.95	11.41	1.28	0.972	HNSW24_C10_S32
0.90	10.04	1.46	0.936	HNSW24_C10_S16
0.85	9.500	1.54	0.872	HNSW24_C10_S8
0.80	9.153	1.60	0.805	HNSW4_C20_S16
0.75	8.523	1.72	0.793	HNSW4_C10_S32
0.70	8.523	1.72	0.793	HNSW4_C10_S32

Table 5.1: Coil-100 runs with min_dist = 0.1, n_neighbors = 25. The first row shows UMAP total time and index recall when using NNDescent. Other rows show UMAP execution time, speedup ratio and average recall of the fastest runs that obtain average recall above the threshold. The speedup is calculated as the ratio between the time using NNDescent over the time for the other index.

can observe that the runs for the IVF class are scattered in the suboptimal zone lower and to the right of the baseline and, as such, are not the best suited candidates for a more efficient nearest neighbor implementation in this case.

Increasing the number of nearest neighbors used for the embedding to 75 we can observe that most of the runs that use Faiss indexes are faster than the NNDescent baseline. HNSW indexes obtain faster execution times and recall performance comparable to the baseline. In this context, IVF indices are capable of reaching recall values greater than 75% while also computing the embedding faster than NNDescent. Table 5.2 shows the speedup obtained by the fastest configurations that obtained an average recall above the threshold. At the highest recall threshold (75-recall@75 > 0.99), HNSW with M = 32 achieves a speedup of 4.68x with a recall of 0.992, showing near baseline recall performance while being significantly faster. As the recall threshold decreases, the speedup reaches 9.14x with average 75-recall@75 = 0.828, achieved by HNSW with M=32 and more conservative search and construction parameter configurations with respect to the highest recall threshold run.

Mouse Figure 5.2 shows the general distribution of runs on the scRNA dataset, reduced to 1000 dimensions using PCA. We can observe that, under both values of n_neighbors, the HSNW class produces runs that obtain comparable recall and reasonable speedup. The IVF class in this dataset does not produce noteworthy results, suffering from low average recall and UMAP execution time larger than NNDescent in all cases, regardless of the choice of parameters.

Table 5.3 shows the best performing indexes and the produced speedup when n_neighbors = 25. HNSW with M=16 achieves the fastest embedding using efConstruction = 20 and efSearch =

Recall Th.	UMAP Time	Speedup	Avg. Recall	Index Key
-	120.51	1.00	0.999	NNDescent
0.99	25.74	4.68	0.992	HNSW32_C40_S64
0.95	15.94	7.55	0.961	HNSW32_C10_S64
0.90	15.82	7.61	0.902	HNSW4_C20_S64
0.85	15.82	7.61	0.902	HNSW4_C20_S64
0.80	13.17	9.14	0.828	HNSW32_C10_S32
0.75	13.17	9.14	0.828	HNSW32_C10_S32
0.70	13.17	9.14	0.828	HNSW32_C10_S32

Table 5.2: Coil-100 runs with min_dist = 0.1, n_neighbors = 75. The first row shows UMAP total time and index recall when using NNDescent. Other rows show UMAP execution time, speedup ratio and average recall of the fastest runs that obtain average recall above the threshold. The speedup is calculated as the ratio between the time using NNDescent over the time for the other index.

64, reaching 0.991 25-recall@25 and a speedup of 1.33x with respect to the baseline. Lowering the recall threshold we can observe that HNSW with M=32 is able to consistently achieve a good trade-off and reach moderately high recall (R=0.904) and reasonable speedup (1.52x). Further lowering the recall threshold does not yield higher speedups, which are limited at 1.53x reached at the 70% recall threshold by HNSW32 index with efConstruction = 10 and efSearch = 4. In Table 5.4, when we increase to 75 the number of nearest neighbors used for the embedding, HNSW with M=32 provides the optimal trade-off at multiple recall thresholds, reaching higher speedups than in the lower number of neighbors case thanks to the longer nearest neighbor phase implied in this case. With efConstruction = 20 and efSearch = 128, HNSW32 is able to provide almost perfect 75-recall@75 (R = 0.997) with a 1.90x speedup over NNDescent. With lower recall threshold the speedup increases up to 2.25x achieved by HNSW4 with efConstruction = 10 and efSearch = 256, while still reaching 0.783 recall on average.

Shuttle Figure 5.3 shows the general distribution of runs on the shuttle dataset, under different values of n_neighbors. When number of neighbors used by the embedding is 25, we can observe that the HNSW index runs are clustered around the baseline and are able to achieve sufficient performance in most cases, both in terms of execution time of UMAP and of average recall, sometimes also surpassing the baseline. On the other hand, IVF runs achieve poor all-around performance under all parameters tested.

Tables 5.5 and 5.6 show the performance of the best runs using 25 and 75 nearest neighbors for the embedding. In the first case, HNSW with M=64 achieves a recall of 0.990, which is higher than NNDescent, while obtaining a 1.29x speedup, providing a more efficient and more

Recall Th.	UMAP Time	Speedup	Avg. Recall	Index Key
-	5.82	1.00	0.999	NNDescent
0.99	4.39	1.33	0.991	HNSW16_C20_S64
0.95	3.89	1.50	0.952	HNSW32_C10_S32
0.90	3.84	1.52	0.904	HNSW32_C10_S16
0.85	3.84	1.52	0.904	HNSW32_C10_S16
0.80	3.84	1.52	0.904	HNSW32_C10_S16
0.75	3.83	1.52	0.751	HNSW64_C10_S4
0.70	3.82	1.53	0.734	HNSW32_C10_S4

Table 5.3: Mouse scRNA dataset runs with min_dist = 0.1, n_neighbors = 25. The first row shows UMAP total time and index recall when using NNDescent. Other rows show UMAP execution time, speedup ratio and average recall of the fastest runs that obtain average recall above the threshold. The speedup is calculated as the ratio between the time using NNDescent over the time for the other index.

Recall Th.	UMAP Time	Speedup	Avg. Recall	Index Key
-	12.48	1.00	1.000	NNDescent
0.99	6.56	1.90	0.997	HNSW32_C20_S128
0.95	6.12	2.04	0.967	HNSW32_C80_S64
0.90	6.12	2.04	0.967	HNSW32_C80_S64
0.85	5.67	2.20	0.889	HNSW32_C10_S32
0.80	5.67	2.20	0.889	HNSW32_C10_S32
0.75	5.54	2.25	0.783	HNSW4_C10_S256
0.70	5.54	2.25	0.783	HNSW4_C10_S256

Table 5.4: Mouse scRNA dataset runs with min_dist = 0.1, n_neighbors = 75. The first row shows UMAP total time and index recall when using NNDescent. Other rows show UMAP execution time, speedup ratio and average recall of the fastest runs that obtain average recall above the threshold. The speedup is calculated as the ratio between the time using NNDescent over the time for the other index.

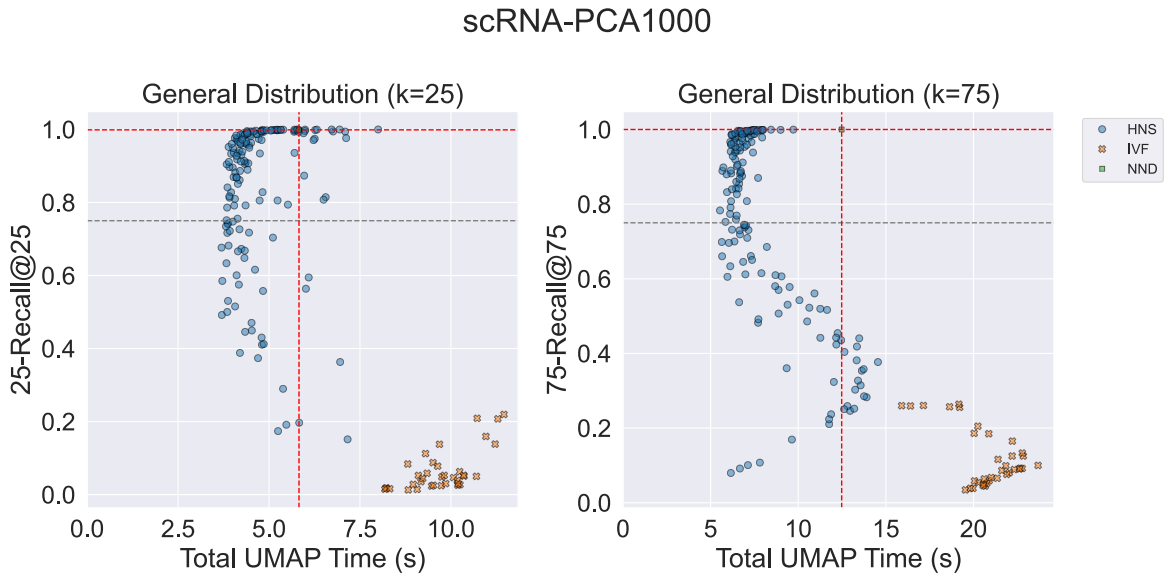


Figure 5.2: Mouse dataset runs using min_dist=0.1, n_neighbors={25,75} for the different index classes tested. On the vertical axis figures the average k-recall@k (higher is better), on the horizontal axis the total UMAP execution time in seconds(lower is better). Red dotted lines signal the average recall and runtime of UMAP using NNDescent.

accurate solution than the baseline. Lowering the recall threshold allows HNSW to reach a speedup of 2.28x when using a smaller M = 16, with efConstruction = 80 and efSearch = 4, and an average recall of 0.745. In the second case, when n_neighbors = 75, HNSW indexes are able to achieve a wide range of average recalls but the speedup over the baseline does not improve after a certain value. We can observe this from both Table 5.6 and Figure 5.3. From the table we can see that in this context the best performing run below the recall threshold 0.95 (HNSW with M=16, efConstruction = 80 and efSearch = 32) is also the fastest run with speedup 2.04x, meaning that indexes with average recall lower than 0.908 do not improve UMAP execution time. On the other hand, the top recall configuration, HNSW32 with efConstruction = 80 and efSearch = 256, achieves recall higher than NNDescent (0.990 vs 0.983) and a speedup of 2.00x over the baseline, which is reasonably similar to the faster, less accurate index speedup.

MNIST Figure 5.4 shows the general distribution of runs for the MNIST dataset. We can observe that in this dataset both the tested classes of index produced runs that surpassed the 0.75 threshold of average recall under both values of n_neighbors parameter, while also obtaining a resonable speedup with respect to the baseline. In table 5.7 we can observe that at high recall threshold HNSW16 is able to achieve optimal trade-off between average recall and execution time in different settings, reaching baseline-level recall (0.990) without significant speed-up (1.00x) when using efConstruction = 20 and efSearch = 64. When using more conservative

Recall Th.	UMAP Time	Speedup	Avg. Recall	Index Key
-	25.86	1.00	0.969	NNDescent
0.99	20.04	1.29	0.990	HNSW64_C40_S128
0.95	16.71	1.55	0.976	HNSW24_C40_S8
0.90	16.71	1.55	0.976	HNSW24_C40_S8
0.85	16.09	1.61	0.897	HNSW32_C20_S1
0.80	16.09	1.61	0.897	HNSW32_C20_S1
0.75	13.58	1.90	0.785	HNSW24_C40_S1
0.70	11.35	2.28	0.745	HNSW16_C80_S4

Table 5.5: Shuttle dataset runs with min_dist = 0.1, n_neighbors = 25. The first row shows UMAP total time and index recall when using NNDescent. Other rows show UMAP execution time, speedup ratio and average recall of the fastest runs that obtain average recall above the threshold. The speedup is calculated as the ratio between the time using NNDescent over the time for the other index.

Recall Th.	UMAP Time	Speedup	Avg. Recall	Index Key
-	32.14	1.00	0.983	NNDescent
0.99	16.10	2.00	0.990	HNSW32_C80_S256
0.95	15.99	2.01	0.987	HNSW64_C40_S32
0.90	15.73	2.04	0.908	HNSW16_C80_S32
0.85	15.73	2.04	0.908	HNSW16_C80_S32
0.80	15.73	2.04	0.908	HNSW16_C80_S32
0.75	15.73	2.04	0.908	HNSW16_C80_S32
0.70	15.73	2.04	0.908	HNSW16_C80_S32

Table 5.6: Shuttle dataset runs with min_dist = 0.1, n_neighbors = 75. The first row shows UMAP total time and index recall when using NNDescent. Other rows show UMAP execution time, speedup ratio and average recall of the fastest runs that obtain average recall above the threshold. The speedup is calculated as the ratio between the time using NNDescent over the time for the other index.

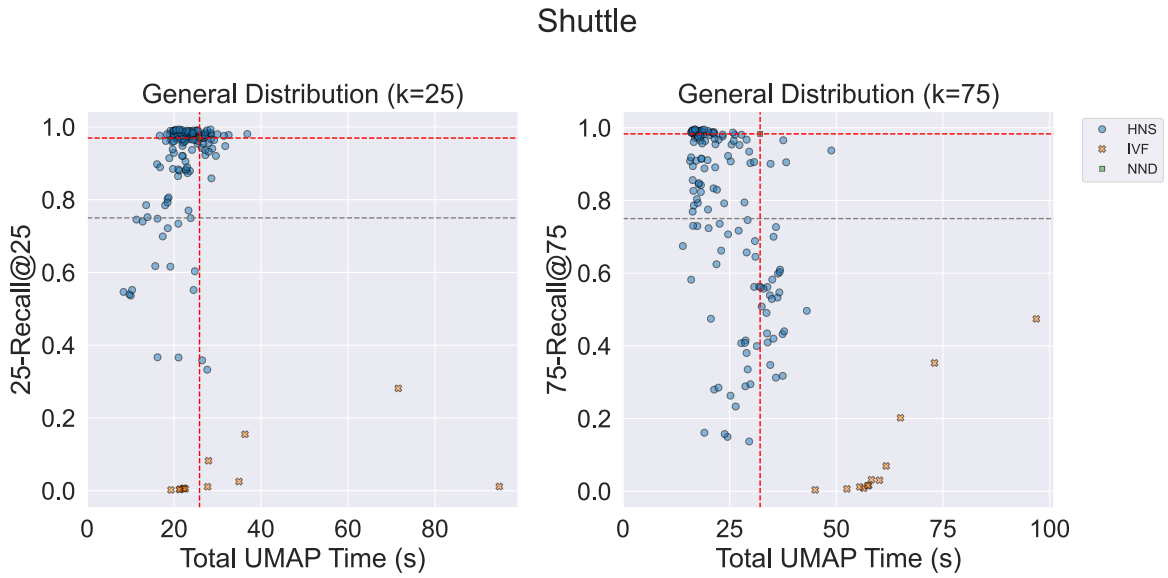


Figure 5.3: Shuttle dataset runs using min_dist=0.1, n_neighbors={25,75} for the different index classes tested. On the vertical axis figures the average k-recall@k (higher is better), on the horizontal axis the total UMAP execution time in seconds(lower is better). Red dotted lines signal the average recall and runtime of UMAP using NNDescent.

construction and search parameters, this same index is able to reach 1.30x speedup and 0.826 average recall at the 0.85 recall threshold. At lower recall threshold (less than 0.75), IVF-PQ is able to provide a modest speedup of 1.37x while still reaching 0.761 recall and achieving a better trade off than HNSW indexes in this case.

Table 5.8 shows how HNSW16 is in this case able to provide baseline-level accuracy (0.990) with a modest speedup of 1.22x at the highest recall threshold of 0.99. At lower recall thresholds HNSW32 and HNSW16 achieve respectively the best tradeoff at 0.95 and 0.90 recall threshold, with 0.955 and 0.926 average recall and 1.46, 1.51 speedup. When the recall threshold drops below 0.80, the IVF-PQ indices provide the optimal trade-off between average recall and speed-up. At recall threshold 0.75, an IVF index with $nlist = 528 \approx 2\sqrt{n}$, m_pq = 98 and nprobe = 32, achieves a speedup of 1.73x and an average 75-recall@75 of 0.793.

FMNIST Figure 5.5 shows the distribution of runs on the Fashion MNIST dataset. We can observe that with lower value of n_neighbors the execution time across the runs is more concentrated around the NNDescent baseline than with an higher number of neighbors. In the latter case we can see that many HNSW runs are able to match the baseline recall and provide a reasonable speedup, much more significant than with a low number of neighbors.

Table 5.9 shows the best speedup obtained by Faiss indexes with a decreasing recall threshold and n_neighbors = 25. In particular, we can observe that the class of in-

Recall Th.	UMAP Time	Speedup	Avg. Recall	Index Key
-	17.36	1.00	0.998	NNDescent
0.99	17.34	1.00	0.990	HNSW16_C20_S64
0.95	14.78	1.17	0.952	HNSW16_C20_S16
0.90	14.29	1.21	0.921	HNSW16_C10_S32
0.85	13.46	1.29	0.862	HNSW16_C10_S16
0.80	13.33	1.30	0.826	HNSW64_C10_S8
0.75	12.69	1.37	0.761	IVF264,PQ98x4fs_C264_S32
0.70	12.69	1.37	0.761	IVF264,PQ98x4fs_C264_S32

Table 5.7: MNIST dataset runs with min_dist = 0.1, n_neighbors = 25. The first row shows UMAP total time and index recall when using NNDescent. Other rows show UMAP execution time, speedup ratio and average recall of the fastest runs that obtain average recall above the threshold. The speedup is calculated as the ratio between the time using NNDescent over the time for the other index.

Recall Th.	UMAP Time	Speedup	Avg. Recall	Index Key
-	34.43	1.00	0.999	NNDescent
0.99	28.29	1.22	0.990	HNSW16_C20_S128
0.95	23.55	1.46	0.955	HNSW32_C20_S32
0.90	22.73	1.51	0.926	HNSW16_C10_S64
0.85	21.87	1.57	0.865	HNSW24_C10_S32
0.80	20.19	1.71	0.808	IVF528,PQ112x4fs_C528_S32
0.75	19.91	1.73	0.793	IVF528,PQ98x4fs_C528_S32
0.70	19.91	1.73	0.793	IVF528,PQ98x4fs_C528_S32

Table 5.8: MNIST dataset runs with min_dist = 0.1, n_neighbors = 75. The first row shows UMAP total time and index recall when using NNDescent. Other rows show UMAP execution time, speedup ratio and average recall of the fastest runs that obtain average recall above the threshold. The speedup is calculated as the ratio between the time using NNDescent over the time for the other index.

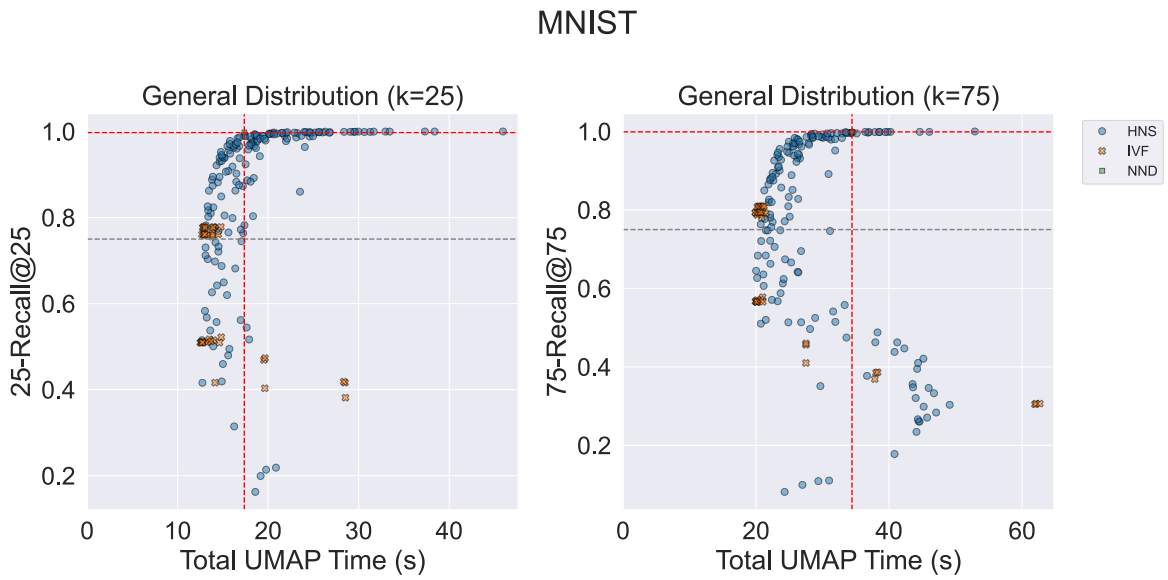


Figure 5.4: MNIST dataset runs using min_dist=0.1, n_neighbors={25,75} for the different index classes tested. On the vertical axis figures the average k-recall@k (higher is better), on the horizontal axis the total UMAP execution time in seconds(lower is better). Red dotted lines signal the average recall and runtime of UMAP using NNDescent.

dex HNSW32 is able to achieve near baseline recall performance with reasonable speedup (R_{avg}=0.991 $Sp.$=1.02x @R_{thresh}=0.99; R_{avg}=0.967 $Sp.$=1.15x @R_{thresh}=0.95; R_{avg}=0.921 $Sp.$=1.22x @R_{thresh}=0.90;). Lowering the recall threshold, HNSW with M = 4 is a robust lower-accuracy solution which provides a consistent speedup at reduced recall (R_{avg}=0.828 $Sp.$=1.28x @R_{thresh}=0.80; R_{avg}=0.828 $Sp.$=1.28x @R_{thresh}=0.75; R_{avg}=0.703 $Sp.$=1.34x @R_{thresh}=0.70;).

Table 5.10 shows the best speedup obtained by Faiss indexes with a decreasing recall threshold and n_neighbors = 75. In particular, we observe that the index class HNSW32 is consistently able to achieve near-baseline recall with significant speedups (R_{avg} = 0.990, $Sp.$ = 1.59x @R_{thresh} = 0.99; R_{avg} = 0.962, $Sp.$ = 1.74x @R_{thresh} = 0.95; R_{avg} = 0.908, $Sp.$ = 1.85x @R_{thresh} = 0.90).

At lower recall thresholds, the HNSW4, HNSW32 with lower parameters and IVF classes provide competitive solutions with higher speedups. For instance, HNSW4 achieves R_{avg} = 0.828, $Sp.$ = 1.97x @R_{thresh} = 0.80 and HNSW32 R_{avg} = 0.797, $Sp.$ = 1.97x @R_{thresh} = 0.75. In this case HNSW32 is able to achieve optimal trade offs at both high and low recall thresholds. The IVF class offers the fastest results, reaching R_{avg} = 0.705 with a speedup of $Sp.$ = 2.11x @R_{thresh} = 0.70, demonstrating its suitability for scenarios with relaxed accuracy requirements.

Recall Th.	UMAP Time	Speedup	Avg. Recall	Index Key
-	17.47	1.00	0.999	NNDescent
0.99	17.18	1.02	0.991	HNSW32_C20_S32
0.95	15.21	1.15	0.967	HNSW32_C10_S32
0.90	14.35	1.22	0.921	HNSW32_C10_S16
0.85	14.02	1.25	0.863	HNSW64_C10_S8
0.80	13.68	1.28	0.828	HNSW4_C20_S32
0.75	13.68	1.28	0.828	HNSW4_C20_S32
0.70	13.06	1.34	0.703	HNSW4_C10_S32

Table 5.9: Fashion MNIST dataset runs with min_dist = 0.1, n_neighbors = 25. The first row shows UMAP total time and index recall when using NNDescent. Other rows show UMAP execution time, speedup ratio and average recall of the fastest runs that obtain average recall above the threshold. The speedup is calculated as the ratio between the time using NNDescent over the time for the other index.

Recall Th.	UMAP Time	Speedup	Avg. Recall	Index Key
-	40.72	1.00	1.000	NNDescent
0.99	25.56	1.59	0.990	HNSW32_C20_S64
0.95	23.37	1.74	0.962	HNSW32_C20_S32
0.90	22.06	1.85	0.908	HNSW32_C10_S32
0.85	21.77	1.87	0.882	HNSW4_C40_S64
0.80	20.71	1.97	0.828	HNSW4_C20_S64
0.75	20.65	1.97	0.797	HNSW32_C10_S16
0.70	19.29	2.11	0.705	IVF264,PQ112x4fs_C264_S32

Table 5.10: Fashion MNIST dataset runs with min_dist = 0.1, n_neighbors = 75. The first row shows UMAP total time and index recall when using NNDescent. Other rows show UMAP execution time, speedup ratio and average recall of the fastest runs that obtain average recall above the threshold. The speedup is calculated as the ratio between the time using NNDescent over the time for the other index.

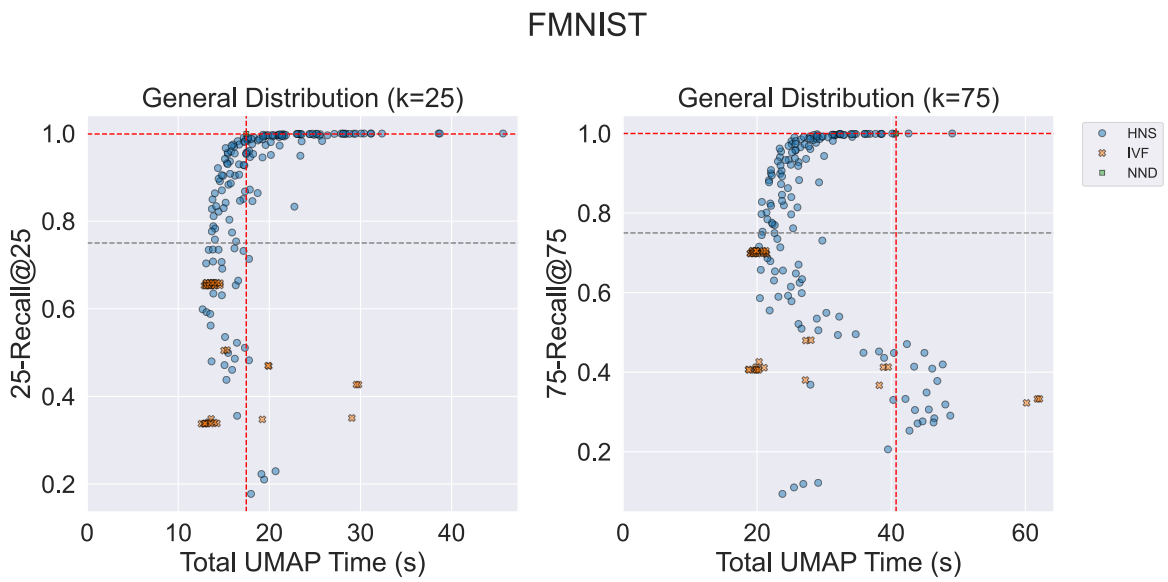


Figure 5.5: Fashion MNIST dataset runs using min_dist=0.1, n_neighbors={25,75} for the different index classes tested. On the vertical axis figures the average k-recall@k (higher is better), on the horizontal axis the total UMAP execution time in seconds(lower is better). Red dotted lines signal the average recall and runtime of UMAP using NNDescent.

Google News word vectors Figure 5.6 shows the distribution of runs on the Google News word vectors 500'000 samples dataset. Compared with the baseline, the HNSW indexes provide faster embedding at the cost of a slightly reduced index recall. We can also observe how the different HNSW configurations cover a wide spectrum of recalls for both values of n_neighbors, showing that in this case each parameter has a sensible impact on the performance of the index.

On the other hand IVF indexes are unable to achieve recall greater than 0.3 on this dataset. Changing the parameters has no positive effect on the performance of this index class that remains unable to reach sufficient accuracy.

Table 5.11 shows the speedup table for this dataset with 25 nearest neighbors used for the embedding. At the highest recall threshold ($R_{thresh} = 0.99$) we can see that HNSW with M = 64, efConstruction = 80 and efSearch = 256, is able to reach near perfect recall, achieving both higher accuracy and higher execution time than the baseline. Lower index parameters (M = 24, efConstruction = 40, efSearch = 128) allow HNSW to match the baseline execution time with a slightly higher average recall (0.956 vs 0.947) at the 0.95 Recall threshold. HNSW32 gains a 1.12x speedup over the baseline at the 0.90 recall level, providing 0.910 average 25-recall@25. Lowering further the recall threshold shows that indexes such as HNSW24 and HNSW16 reach a speedup of up to 1.28x while providing 0.780 recall on average.

Table 5.12 shows the speedup table for 75 nearest neighbors. Also in this case, the configuration HNSW64 with efConstruction = 80 and efSearch = 256, which is the "biggest" config-

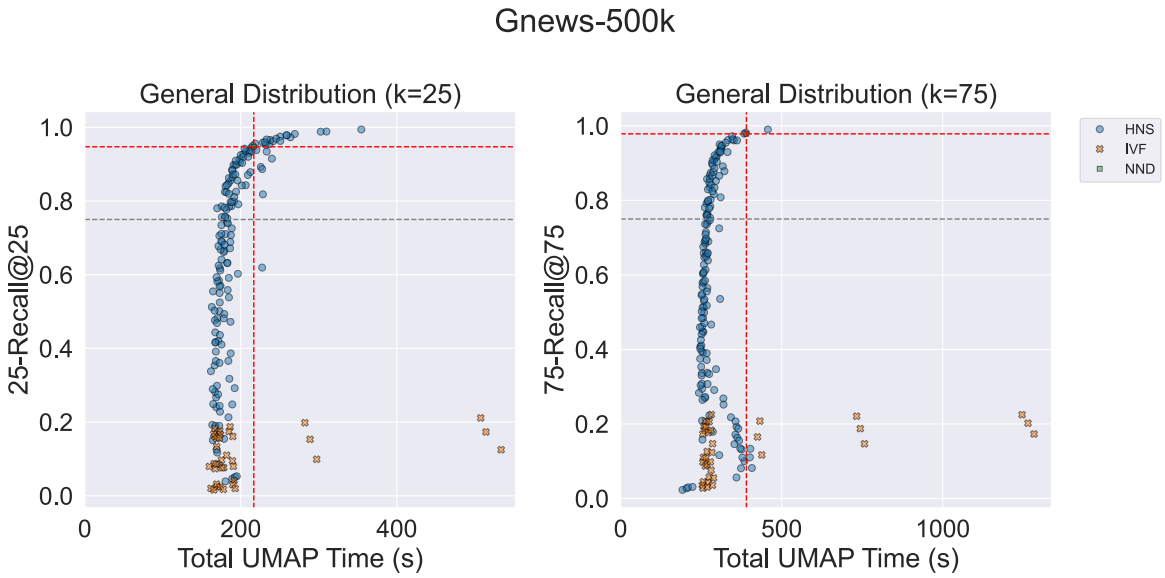


Figure 5.6: Google News dataset runs using min_dist=0.1, n_neighbors={25,75} for the different index classes tested. On the vertical axis figures the average k-recall@k (higher is better), on the horizontal axis the total UMAP execution time in seconds(lower is better). Red dotted lines signal the average recall and runtime of UMAP using NNDescent.

uration we tested parameters-wise, is the fastest 99% 75-recall@75 configuration. It provides a more accurate k nearest neighbors graph than NNDescent despite taking ~17% more time. At 0.95 recall threshold, HNSW is able to reach 0.95 recall with 1.27x speedup, in the HNSW24 with efConstruction = 80 and efSearch = 256 configuration. At the 0.90 recall threshold the speedup is raised to 1.37x, which corresponds to a difference of around ~100 seconds. This result is obtained using the HNSW32 index. Lowering the average recall of the index to 0.781 allows for a speedup of 1.49x, which is around 2 minutes shorter than the baseline's ~6.5 minutes. This configuration (HNSW with M=16, efConstruction = 40 and efSearch = 64) is the fastest for the two lowest recall thresholds.

5.2 Embedding Quantitative Performance

In this section we will analyze quantitatively the performance of the proposed implementation, and compare it with the performance of NNDescent. We will use a accuracy of a k Nearest neighbors Classifier to measure how well an embedding is able to encode the structure present in the original space. In practice, we train the kNN Classifier on an embeddings using 10 or 20 fold cross-validation and we record the average accuracy and its standard deviation of the classifier across the folds. The score expresses how well the k nearest neighbors of each point in the embedding space are representative of the label of the point (i.e. how well the kNN in the

Recall Th.	UMAP Time	Speedup	Avg. Recall	Index Key
-	216.73	1.00	0.947	NNDescent
0.99	354.50	0.61	0.994	HNSW64_C80_S256
0.95	218.56	0.99	0.956	HNSW24_C40_S128
0.90	193.60	1.12	0.910	HNSW32_C40_S32
0.85	184.65	1.17	0.862	HNSW24_C80_S32
0.80	179.64	1.21	0.825	HNSW24_C20_S32
0.75	169.79	1.28	0.780	HNSW16_C40_S32
0.70	169.79	1.28	0.780	HNSW16_C40_S32

Table 5.11: Google News word vectors dataset runs with min_dist = 0.1, n_neighbors = 25. The first row shows UMAP total time and index recall when using NNDescent. Other rows show UMAP execution time, speedup ratio and average recall of the fastest runs that obtain average recall above the threshold. The speedup is calculated as the ratio between the time using NNDescent over the time for the other index.

Recall Th.	UMAP Time	Speedup	Avg. Recall	Index Key
-	390.60	1.00	0.979	NNDescent
0.99	457.12	0.85	0.990	HNSW64_C80_S256
0.95	307.35	1.27	0.950	HNSW24_C80_S256
0.90	286.01	1.37	0.908	HNSW32_C20_S128
0.85	268.43	1.46	0.857	HNSW16_C40_S128
0.80	268.43	1.46	0.857	HNSW16_C40_S128
0.75	261.48	1.49	0.781	HNSW16_C40_S64
0.70	261.48	1.49	0.781	HNSW16_C40_S64

Table 5.12: Google News word vectors dataset runs with min_dist = 0.1, n_neighbors = 75. The first row shows UMAP total time and index recall when using NNDescent. Other rows show UMAP execution time, speedup ratio and average recall of the fastest runs that obtain average recall above the threshold. The speedup is calculated as the ratio between the time using NNDescent over the time for the other index.

Dataset	k	NNDescent	M16,C10,S64	M16,C80,S32	M32,C10,S32
Shuttle		*(-, R=98%)*	*(1.43, R=96%)*	*(**2.04**, R=90%)*	*(1.04, R=90%)*
	100	0.994 ± 0.002	0.992 ± 0.002	0.991 ± 0.002	0.991 ± 0.001
	200	0.991 ± 0.002	0.989 ± 0.002	0.987 ± 0.002	0.989 ± 0.002
	400	0.989 ± 0.002	0.988 ± 0.002	0.983 ± 0.002	0.988 ± 0.002
	800	0.983 ± 0.003	0.983 ± 0.002	0.974 ± 0.003	0.983 ± 0.002
	1600	0.974 ± 0.003	0.970 ± 0.003	0.961 ± 0.003	0.976 ± 0.003
	3200	0.959 ± 0.003	0.960 ± 0.003	0.940 ± 0.003	0.960 ± 0.003
MNIST		*(-, R=100%)*	*(1.51, R=92%)*	*(1.31, R=93%)*	*(**1.56**, R=88%)*
	100	0.964 ± 0.007	0.951 ± 0.008	0.963 ± 0.007	0.948 ± 0.008
	200	0.963 ± 0.007	0.950 ± 0.008	0.963 ± 0.007	0.948 ± 0.009
	400	0.963 ± 0.007	0.950 ± 0.008	0.962 ± 0.007	0.947 ± 0.009
	800	0.962 ± 0.007	0.950 ± 0.008	0.962 ± 0.007	0.947 ± 0.009
	1600	0.961 ± 0.008	0.949 ± 0.009	0.960 ± 0.008	0.946 ± 0.009
	3200	0.959 ± 0.008	0.947 ± 0.009	0.958 ± 0.008	0.945 ± 0.009
FMNIST		*(-, R=100%)*	*(1.73, R=95%)*	*(1.55, R=93%)*	*(**1.85**, R=91%)*
	100	0.777 ± 0.007	0.777 ± 0.006	0.778 ± 0.007	0.777 ± 0.007
	200	0.774 ± 0.007	0.774 ± 0.007	0.775 ± 0.007	0.773 ± 0.007
	400	0.768 ± 0.007	0.768 ± 0.008	0.769 ± 0.007	0.767 ± 0.007
	800	0.758 ± 0.006	0.757 ± 0.006	0.758 ± 0.006	0.755 ± 0.006
	1600	0.736 ± 0.006	0.738 ± 0.006	0.737 ± 0.007	0.734 ± 0.006
	3200	0.729 ± 0.007	0.725 ± 0.007	0.727 ± 0.007	0.718 ± 0.006

Table 5.13: kNN Classifier Accuracy for varying values of k on embeddings with n_neighbors = 75. The classifier has been trained and tested on the embeddings of Shuttle, MNIST, Fashion MNIST produced using different algorithms for the UMAP k nearest neighbors calculation. For HNSW indexes a synthetic notation is used. For instance M16, C10, S64 stands for HNSW index with parameters M = 16, efConstruction = 10 and efSearch = 64. For these algorithms we also include values for speedup and average recall. The average accuracy scores are given over 10-fold or 20-fold cross validation.

embedding correspond to the kNN in the original data). Varying the value of k allows to explore different contexts. Lower values of k give information about the local structure, while higher values of k describe how well the global structure has been captured by the embedding. For the above reasons, the accuracy of the kNN classifier encapsulates important information about the usability of the embedding by possible downstream tasks. The datasets used in this analysis are Shuttle, MNIST and Fashion MNIST since they are the ones which possess labels.

We tested 3 HNSW configurations: M = 16, efConstruction = 10 and efSearch = 64; M = 16, efConstruction = 80 and efSearch = 32; M = 32, efConstruction = 10 and efSearch = 32. These parameters combinations strike a balance between high-recall-oriented configurations and high-speed configurations in our parameter grid. Table 5.13 shows the comparison between

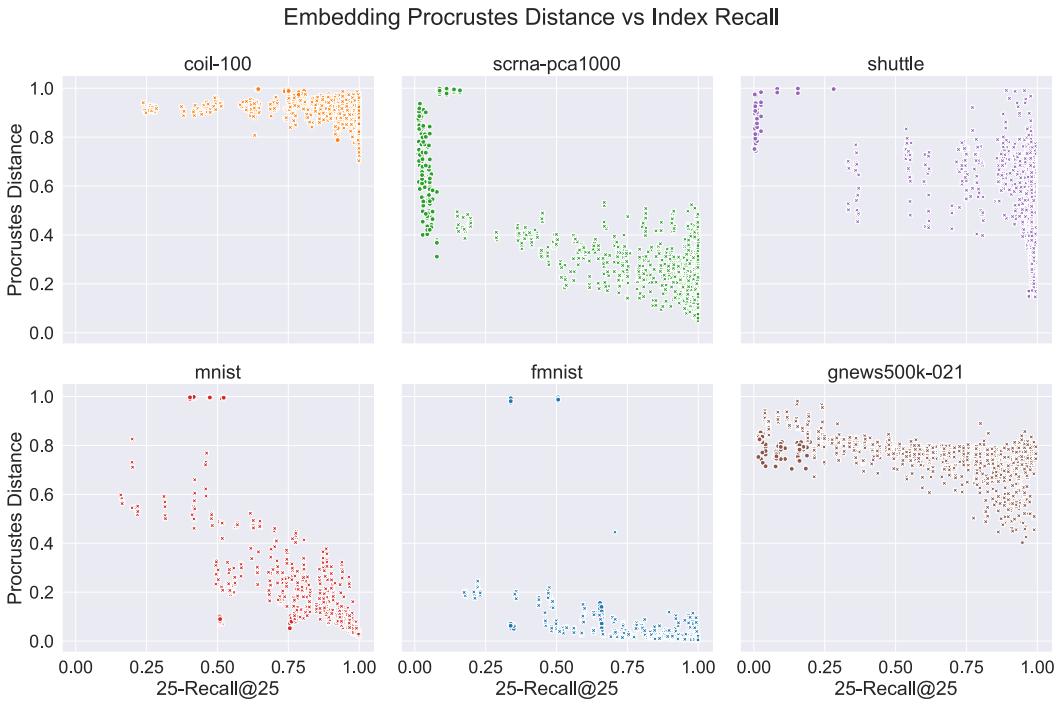


Figure 5.7: Scatter plot of Procrustes distance values vs the recall of the index that produced the embedding for the runs with n_neighbors = 25

these configuration and NNDescent when running the kNN classifier on embeddings obtained with the UMAP parameter n_neighbors = 75. On the Shuttle dataset, we can observe that there is minimal discrepancy between the accuracy of NNDescent and the HNSW configurations, which are still able to achieve high accuracy while providing a reasonable speedup. On the MNIST dataset, the difference in accuracy is more pronounced than in the Shuttle dataset and in particular with higher values of k but still not significantly different given the confidence bounds. In this case the best performing Faiss index is the one with the highest efConstruction parameter. On the more challenging Fashion MNIST dataset we can see that increasing k to 3200, the accuracy decreases significantly for the lowest recall HNSW index. Nonetheless the difference from NNDescent is present but not completely outside of confidence bounds.

5.3 Embedding Qualitative Analysis and Stability

In this section we will analyze the quality of the embeddings and relate it to the recall of the index used to generate the embedding. We use the value of the Procrustes distance of each embedding calculated with respect to a full-recall embedding and analyze the distribution for each dataset.

Figure 5.7 and Figure 5.8 show the value of the Procrustes distances of each embedding by the recall of the index that was employed to produce it, for n_neighbors = 25 and n_neighbors

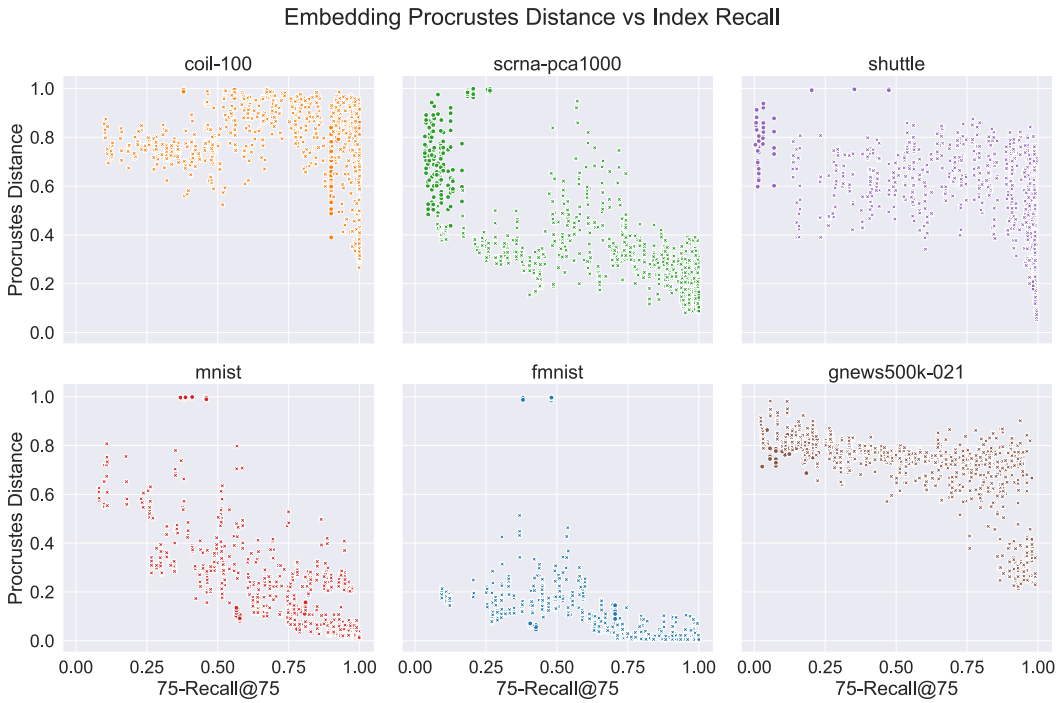


Figure 5.8: Scatter plot of Procrustes distance values vs the recall of the index that produced the embedding for the runs with n_neighbors = 75

= 75 respectively. We can observe that with n_neighbors = 75 the point distribution is more scattered than in the lower number of neighbors case. This could be a consequence of the fact that an higher number of neighbor leads UMAP to bring closer the local clusters to enhance the global representation and sacrificing detail structures. This is evident in the COIL-100 dataset whose characteristic loop local structures resent of the added focus on the global structure under this particular metric.

We can see from Figure 5.7 and Figure 5.8 that the recall value of the index and the Procrustes distance are weakly correlated, that is, a sufficiently high recall value is necessary to have a low Procrustes distance but it is not sufficient, as for a single recall value, even very high, there exist many possible values of recall. This behavior corresponds to vertically aligned points in the plot, which belong to different runs under the same parameters.

Figure 5.9 and Figure 5.10 show more in detail the relation between Procrustes distance vs average recall on the Google news embeddings dataset, for n_neighbors = 25 and n_neighbors = 75 respectively. In these plots we can appreciate the distribution for each index class. In particular, we can see that the 5 runs of NNDescent (green diamond shapes) achieve all very different values of recall, despite being the same exact configuration. This behavior is also present for the configurations of the other index classes and corresponds as before to vertically aligned points, meaning that the runs achieve similar recall but different Procrustes distance.

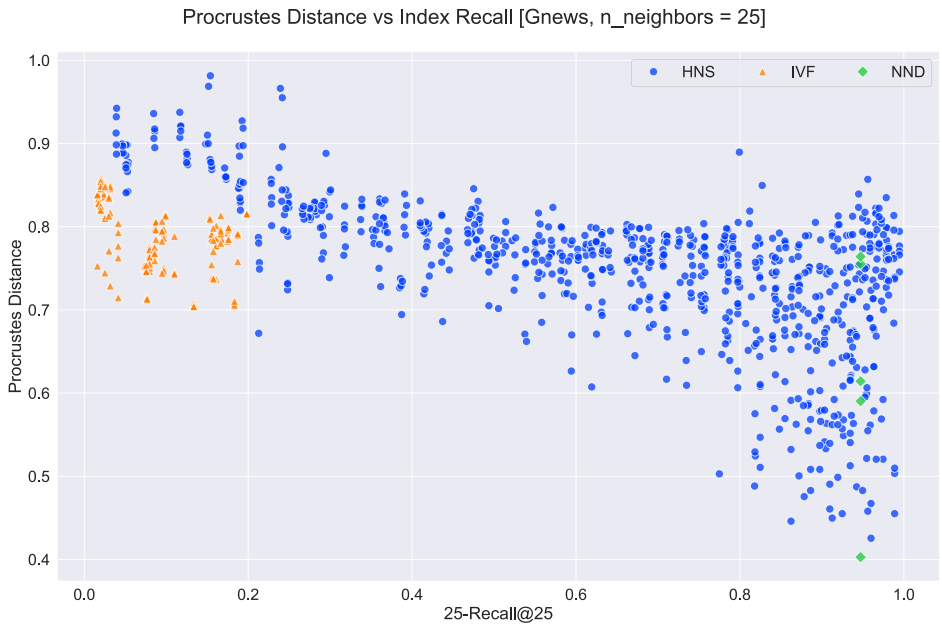


Figure 5.9: Scatter plot of Procrustes distance values vs the recall of the index that produced the embedding for the runs with n_neighbors = 25

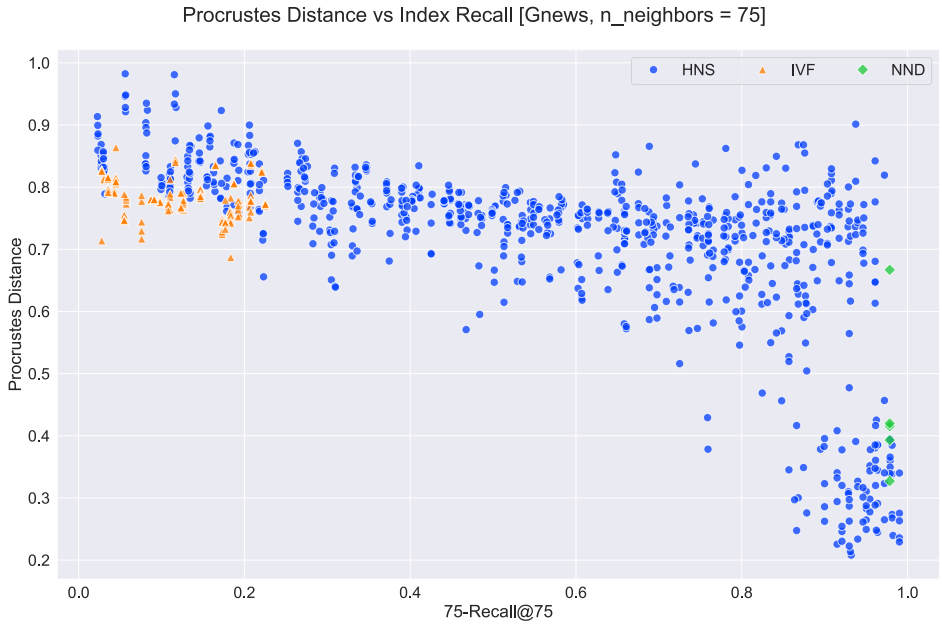


Figure 5.10: Scatter plot of Procrustes distance values vs the recall of the index that produced the embedding for the runs with n_neighbors = 75

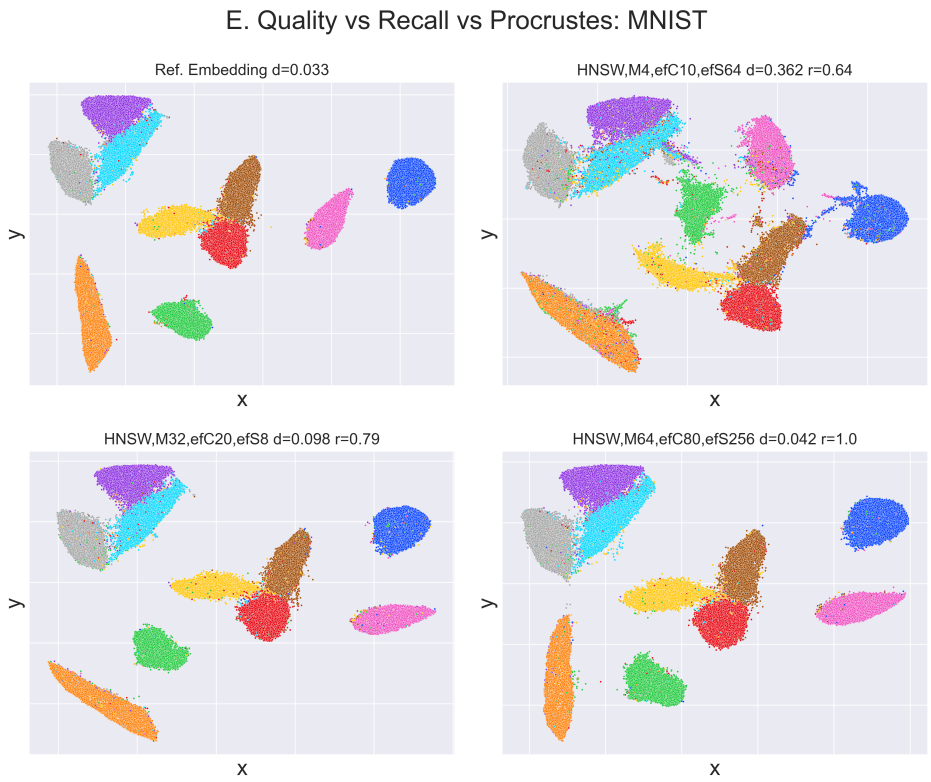


Figure 5.11: MNIST embeddings with n_neighbors = 75 and min_dist = 0.1, using different indexes. The titles include the index configuration, the Procrustes distance d, the index average recall r. A Procrustes superimposition w.r.t. the reference embedding has been applied to the embeddings

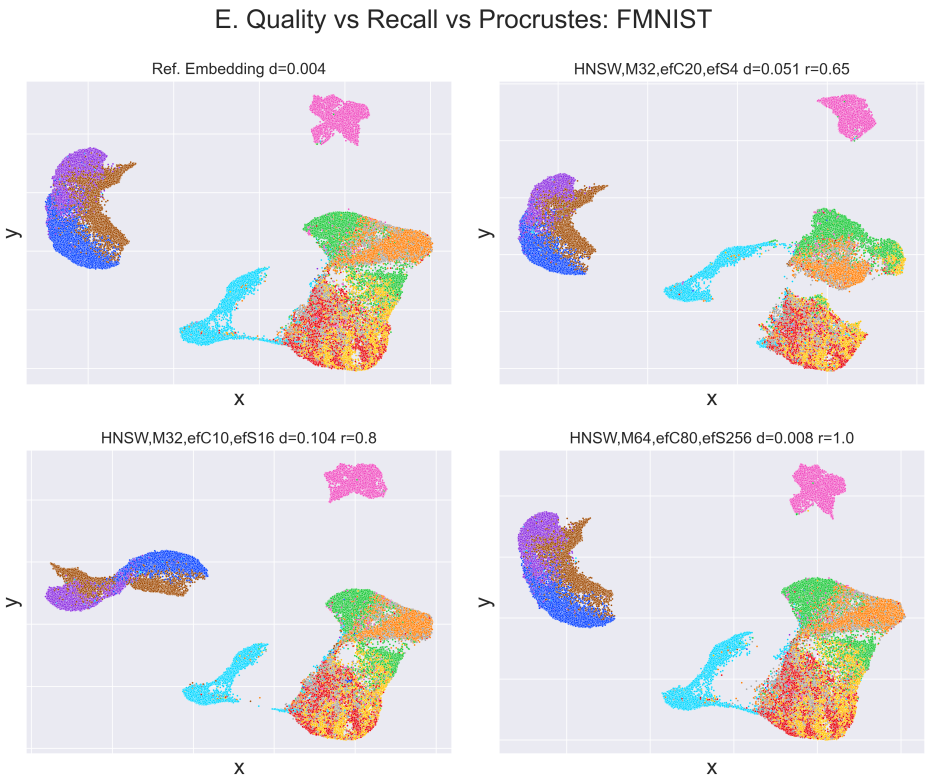


Figure 5.12: FMNIST embeddings with n_neighbors = 75 and min_dist = 0.1, using different indexes. The titles include the index configuration, the Procrustes distance d, the index average recall r. A Procrustes superimposition w.r.t. the reference embedding has been applied to the embeddings

Figure 5.11 shows some embeddings under different recall values for the MNIST dataset, along with the configuration that produced the embedding and the Procrustes distance. In this case we can observe how both the bottom plots are qualitatively similar to the reference NNDescent embedding on the top left, while the embedding on the top right, with average recall 0.64, presents significant global and local differences. We can also observe how Procrustes distance values under 0.1 seem to be acceptable in the context of this dataset.

Figure 5.12 shows the embedding grid for the Fashion MNIST dataset. On the top left we have the reference NNDescent embedding, with 0.004 Procrustes distance from a full recall reference embedding. On the top right we can see a 0.65 recall HNSW embedding which presents significant differences from the reference embedding. In particular most of the characteristic local structures are altered and resemble loosely the reference embedding. The global structure is more or less present, except for some missing ”bridges” between the clusters. The embedding on the bottom leftis able to represent correctly most of the global and local structures, despite the 0.8 recall. However, since it does not manage to embed correctly the leftmost cluster, its Procrustes distance is the highest of this batch, even if the embedding is qualitatively accept-

Figure 5.13: Google News embeddings with n_neighbors = 75 and min_dist = 0.1, using different indexes. The titles include the index configuration, the Procrustes distance d, the index average recall r. A Procrustes superimposition w.r.t. the reference embedding has been applied to the embeddings

able. The full recall HNSW embedding on the bottom right achieves an almost indistinguishable embedding from the reference.

Figure 5.13 shows embeddings grid for the Google News dataset, which does not posses labels. The reference NNDescent embedding on the top left corner achieves Procrustes distance = 0.327, highlighting how values for this measure should be evaluated based on the dataset context. Among the HNSW embeddings, the top right and the bottom left do not seem to correctly represent the global structure of the dataset, despite seemingly being able to capture some local detail like the lowest star-like structure. This is reflected by their Procrustes distance above the value of 0.7. The bottom right, near-exact recall HNSW embedding is able to achieve an embedding very similar both the reference and to the ground truth as reflected by its Procrustes distance of 0.201, lower than the reference.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

We evaluated an extended implementation by examining the trade-off between recall and speedup, studying the resulting embedding's quantitative performance, and analyzing its qualitative properties. We observed that approximate nearest neighbor algorithms like HNSW can significantly accelerate UMAP embeddings, achieving up to a 9.14x speedup while producing high quality embeddings for most datasets. Table 6.1 shows a summary of the speedup values across the tested datasets for the fastest indexes that achieve recall greater than a set threshold. The maximum achievable speedup is influenced by the specific characteristics of the dataset, with the number of samples and dimensions being evident factors, as demonstrated by variations in performance across datasets like COIL-100 and Google News.

The choice of UMAP parameters, particularly the number of neighbors, significantly impacts the total runtime and, consequently, the speedup. As shown in the tables from Section 5.1, using a higher number of neighbors (e.g. n_neighbors = 75) increases the complexity of the nearest neighbors calculation phase, resulting in longer runtimes compared to configurations with fewer neighbors (e.g. n_neighbors = 25). The proposed implementation scales well with the number of neighbors, as the relative increase in runtime for higher n_neighbors values is smaller compared to the baseline (NNDescent). This results in higher speedup ratios at increased neighbor counts, as seen for instance in Tables 5.11 and 5.12, where the speedup improves from 1.12x at n_neighbors=25 and Recall >90% to 1.37x at n_neighbors=75 and the same recall threshold. This demonstrates the robustness of the proposed approach in handling more intensive configurations while maintaining its efficiency over the baseline.

Experiments in Section 5.1 show how specific configurations are able to provide effective trade-off between recall and execution time in multiple datasets, under different accuracy thresholds and embedding parameters. Some of these configurations have been analyzed in Section

Dataset	n samples	d	Sp.@R>75%	Sp.@R>90%
COIL-100	7'200	45'152	9.14	7.61
Mouse	20'921	1000	2.25	2.04
Shuttle	58'000	9	2.04	2.04
MNIST	70'000	784	1.73	1.51
FMNIST	70'000	784	1.97	1.85
Gnews	500'000	300	1.49	1.37

Table 6.1: Speedup table summary for each dataset of the fastest runs with recall greater than 0.75 and 0.9, using n_neighbors = 75

5.2 to assess the resulting embedding's quantitative performance by studying the accuracy of a kNN Classifier trained on the embedding. In particular, Table 5.13 highlights that configurations such as HNSW with M=16 or M=32 can produce embeddings that achieve classifier accuracy comparable to the baseline (NNDescent) while delivering a significant speedup, proving that the tested HNSW indexes are able to compute high quality embeddings for downstream tasks while optimizing runtime efficiency.

On the other hand, we could not find any configuration for the IVF class of indexes able to consistently achieve significant trade offs in speed and accuracy on the tested datasets. In the COIL-100 and MNIST datasets, some of the tested IVF configurations were able to obtain good recall/speedup results, beating the NNDescent baseline execution time with acceptable recall values (Figures 5.1 and 5.4). These were two isolated cases as similar configurations did not perform as well on the other datasets, indicating that for this type of index could be necessary particular care on the choice of parameters.

In section 5.2 we showed that the proposed precision trade-off has no significant impact the accuracy of a downstream kNN Classifier trained on the embedding at local or global scale. This leads us to the conclusion that the quantitative quality of the embedding is preserved.

In section 5.3 we performed a qualitative analysis of the embeddings and tried to relate it to the Procrustes distance. We studied how the UMAP parameter n_neighbors, the index recall, the Procrustes distance and dataset characteristics interact to influence the embedding quality. In particular, increasing the neighbor count improves UMAP representation of the global structure while penalizing local detail, which can lead to greater variation in the Procrustes distance. This effect is particularly pronounced in datasets composed of numerous smaller local structures, as is the case with COIL-100.

On the other hand, for datasets where the structure is more globally oriented (i.e. characterized by larger, less detailed clusters) the growth of Procrustes distance variability with the number of neighbors is less pronounced. In these cases, increasing the neighbor count primarily enhances the alignment of the global structure without significantly impacting local detail.

As a result, it is often possible to identify a recall threshold above which the embedding sufficiently captures the dominant global structure of the dataset. Beyond this threshold, a stronger relationship emerges between Procrustes distance, recall value, and the overall quality of the embedding, as observed in datasets such as MNIST (Figure 5.11 and Fashion MNIST (Figure 5.12). In these datasets, an average recall value of around 0.7 already captures most of the global structure and increasing the recall leads to a refinement of the local patterns' boundaries. This suggests that the dataset's intrinsic structure influences how recall affects embedding quality and Procrustes distance, highlighting the importance of dataset-specific considerations.

6.2 Future Work

While this thesis demonstrates the effectiveness of Approximate Nearest Neighbor search algorithms to accelerate UMAP embeddings, it also highlights possible directions for future investigation.

As index configurations play an important role in the performance and the quality of the embedding, exploring a broader grid of parameters, especially for the IVF class of indexes, could provide further optimizations to the algorithm. Additionally, including new classes of indexes could allow to discover other robust choices, similar to HNSW, that achieve comparable or superior speedups. Also, since selecting the optimal parameter combination for an index is an hard task, devising methods to perform this choice automatically could allow the seamless adoption of fast algorithms that do not have a robust reference configuration.

To further expand and validate the results in this work, future studies could scale the analysis to larger datasets, providing more challenging and diverse scenarios for the indexes tested.

Bibliography

[1] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[2] J. N. Campbell, E. Z. Macosko, H. Fenselau, *et al.*, "A molecular census of arcuate hypothalamus and median eminence cell types," *Nature neuroscience*, vol. 20, no. 3, pp. 484–496, 2017.

[3] G. Carlsson, T. Ishkhanov, V. De Silva, and A. Zomorodian, "On the local behavior of spaces of natural images," *International journal of computer vision*, vol. 76, pp. 1–12, 2008.

[4] A. Collins, A. Zomorodian, G. Carlsson, and L. J. Guibas, "A barcode shape descriptor for curve point cloud data," *Computers & Graphics*, vol. 28, no. 6, pp. 881–894, 2004.

[5] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the twentieth annual symposium on Computational geometry*, 2004, pp. 253–262.

[6] W. Dong, C. Moses, and K. Li, "Efficient k-nearest neighbor graph construction for generic similarity measures," in *Proceedings of the 20th international conference on World wide web*, 2011, pp. 577–586.

[7] M. Douze, A. Guzhva, C. Deng, *et al.*, "The faiss library," 2024. arXiv: 2401.08281 [cs.LG].

[8] D. Dua and C. Graff, *UCI machine learning repository: Statlog (shuttle) data set*, 2019. [Online]. Available: http://archive.ics.uci.edu/ml/datasets/Statlog+(Shuttle).

[9] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast approximate nearest neighbor search with the navigating spreading-out graph," *arXiv preprint arXiv:1707.00143*, 2017.

[10] A. Gionis, P. Indyk, R. Motwani, *et al.*, "Similarity search in high dimensions via hashing," in *Vldb*, vol. 99, 1999, pp. 518–529.

[11] H. Hotelling, "Analysis of a complex of statistical variables into principal components.," *Journal of educational psychology*, vol. 24, no. 6, p. 417, 1933.

[12] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2010.

[13] Y. LeCun, C. Cortes, C. Burges, *et al.*, *Mnist handwritten digit database*, 2010.

[14] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne.," *Journal of machine learning research*, vol. 9, no. 11, 2008.

[15] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 42, no. 4, pp. 824–836, 2018.

[16] L. McInnes, J. Healy, and J. Melville, "Umap: Uniform manifold approximation and projection for dimension reduction," *arXiv preprint arXiv:1802.03426*, 2018.

[17] T. Mikolov, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, vol. 3781, 2013.

[18] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[19] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE transactions on pattern analysis and machine intelligence*, vol. 36, no. 11, pp. 2227–2240, 2014.

[20] S. A. Nene, S. K. Nayar, H. Murase, *et al.*, "Columbia object image library (coil-100)," 1996.

[21] S. A. Nene, S. K. Nayar, H. Murase, *et al.*, "Columbia object image library (coil-20)," 1996.

[22] S. M. Omohundro, "Five balltree construction algorithms," 1989.

[23] J. B. Tenenbaum, V. d. Silva, and J. C. Langford, "A global geometric framework for nonlinear dimensionality reduction," *science*, vol. 290, no. 5500, pp. 2319–2323, 2000.

[24] L. Van Der Maaten, E. O. Postma, H. J. van den Herik, *et al.*, "Dimensionality reduction: A comparative review," *Journal of Machine Learning Research*, vol. 10, no. 66-71, p. 13, 2009.

[25] L. Vietoris, “Über den höheren zusammenhang kompakter räume und eine klasse von zusammenhangstreuen abbildungen,” *Mathematische Annalen*, vol. 97, no. 1, pp. 454–472, 1927.

[26] H. Xiao, K. Rasul, and R. Vollgraf, “Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms,” *arXiv preprint arXiv:1708.07747*, 2017.