5

Linear local embedding

Pouya Jafari[a], Ehsan Espandar[a], Fatemeh Baharifard[b], and Snehashish Chakraverty[c]

[a]Department of Computer Science, Iran University of Science and Technology, Tehran, Iran [b]School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran [c]Department of Mathematics, National Institute of Technology Rourkela, Rourkela, Odisha, India

5.1 Introduction

5.1.1 What is nonlinear dimensionality reduction?

Dimension reduction refers to a collection of techniques used to transform high-dimensional data into a lower-dimensional space while maintaining the relationships and structure between data points. This is carried out because high-dimensional data can be difficult and expensive to analyze and interpret, due to the sheer number of dimensions involved. Dimension reduction techniques aim to reduce the dimensionality of these datasets while retaining as much relevant information as possible and thereby simplifying the data analysis process [1,2].

Dimensionality reduction techniques can be broadly categorized as linear and nonlinear methods. Linear methods attempt to preserve the most variance in the data through a linear transformation. One example of a widely used linear method is Principal Component Analysis (PCA) [3–6], which finds a set of orthogonal axes that capture the most variance in the data and projects the data onto those axes. PCA can be used to reduce the noise in images. Another method for dimension reduction is Linear Discriminant Analysis (LDA) [7]. Linear discriminant analysis's goal is to maximize the separation between classes and minimize the variation within each class. This means LDA is a supervised method that finds a linear combination between high-dimensional points and maps it into a lower dimension. By applying PCA or LDA to the image data, you can identify the most important features, remove the noise, and make machine learning algorithms faster. This can be useful in a variety of applications such as facial recognition, object detection, and medical imaging.

Dimensionality linear methods such as PCA assume data on a linear subspace. However, in many situations, data can lie in a nonlinear subspace. In these cases, data structures like nonlinear manifolds, nonlinear interactions, non-Gaussian distributions, and high-dimensional data may not be properly recognized by linear methods. Hence, nonlinear dimension reduction methods attempt to preserve nonlinear relationships between data points by preserving the local structure of the data. One example of a nonlinear method is Locally Linear Embedding (LLE) [8], which represents each data point as a lin-

Dimensionality Reduction in Machine Learning. https://doi.org/10.1016/B978-0-44-332818-3.00015-0

ear combination of its neighbors, which is a nonlinear relationship between points, and then finds a low-dimensional embedding by optimizing a cost function that preserves the local structure of the data. LLE can be used to identify the underlying structure of high-dimensional data, including images, texts, and gene expressions. By reducing the dimensionality of the data, LLE can identify the most important features that contribute to image recognition, natural language processing and particular diseases. For an example of applying the LLE method, we have embedded high-dimensional data like handwritten images of digits '4' and '9' by LLE and transformed our images into two-dimensional space in Fig. 5.1. This figure indicates that images of digit '4' are close to each other and '9's behave in the same way.

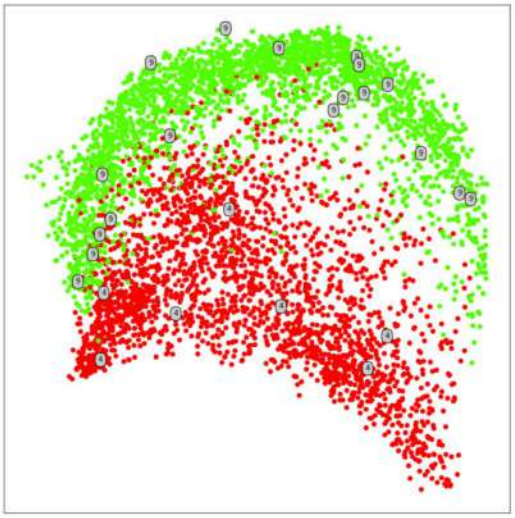

FIGURE 5.1 Dimension reduction on MNIST dataset using LLE for digits '4' and '9'.

Nonlinear dimensionality reduction techniques have a wide range of applications in fields such as computer vision, natural language processing, and bioinformatics. For example, these techniques are used for feature extraction and image classification in computer vision, document clustering, and topic modeling in natural language processing, and gene expression analysis and drug discovery in bioinformatics [1,9].

5.1.2 Why do we need nonlinear dimensionality reduction?

High-dimensional data is ubiquitous in many fields, such as biology, finance, and computer science [10]. This data can be difficult to visualize, process, and analyze due to its high dimensionality. Furthermore, high-dimensional data may contain complex nonlinear relationships between features, which can be difficult to model with linear methods.

Dimensionality reduction techniques can alleviate these challenges by transforming high-dimensional data into lower-dimensional spaces while preserving its essential char-

acteristics. Linear dimensionality reduction methods, such as PCA and LDA, are widely used due to their efficiency and ease of implementation. However, they make the assumption that the relationship between the features is linear, which may not hold for many real-world datasets including image datasets.

Nonlinear dimensionality reduction techniques are needed to capture the nonlinear relationships between features that may exist in high-dimensional data. These techniques model the data as a nonlinear function of the input variables, allowing for a more accurate representation of the data structure. Furthermore, nonlinear dimensionality reduction techniques can reveal the underlying structure of the data, which may not be visible in the original high-dimensional space.

For example, consider a dataset of images of handwritten digits. In the high-dimensional space, each image is represented as a vector of pixel values. Linear dimensionality reduction techniques may not be able to capture the complex relationships between the pixels that correspond to the curvature of the digit strokes or the angles between them. Nonlinear techniques, such as t-SNE [11] or Isomap [2], can capture these relationships and produce a more informative visualization of the data.

Nonlinear dimensionality reduction techniques can also be useful for clustering or classification tasks. In high-dimensional data, the inherent structure of the data may be obscured by noise or irrelevant features. Nonlinear dimensionality reduction can help identify the relevant features and reduce the impact of noise in the data, leading to more accurate clustering or classification results.

In conclusion, nonlinear dimensionality reduction techniques are necessary to handle high-dimensional data with complex nonlinear relationships between features. These techniques can capture the underlying structure of the data, reveal the relevant features, and reduce the impact of noise, leading to more accurate modeling, visualization, and analysis of the data.

5.1.3 What is embedding?

An embedding is a mathematical concept used in machine learning, data analysis, natural language processing, and computer vision to represent complex data or objects in a lower-dimensional space [12,13]. The goal is to reduce the dimensionality of the data while preserving certain properties or relationships between the objects. There are several techniques for constructing embeddings, but they all aim to maintain the structure of the original data.

One common approach for constructing embeddings is to use dimensionality reduction techniques such as multi-dimensional scaling (MDS), PCA, LLE or t-SNE. These methods aim to find a lower-dimensional representation of the data that captures as much of the original structure as possible. For instance, if one needs to visualize a dataset of high-dimensional points in a two-dimensional plot, applying PCA or MDS can help obtain a two-dimensional embedding that shows the structure of the data. One interesting use of LLE is microarray data analysis [14]. The microarray data are high dimensional in nature

and we can use them for cancer classification. t-SNE is particularly useful when visualizing high-dimensional data in two or three dimensions.

Machine learning algorithms can also learn embeddings from data. The primary goal is to learn an embedding that is useful for tasks such as classification, clustering, or recommendation. For instance, in natural language processing, word embeddings are used to represent words as vectors that capture their semantic meaning. These embeddings can be used as input to neural networks for sentiment analysis, language translation, or question-answering.

In image recognition, deep convolutional neural networks can be used to learn feature representations of images that capture important features of the images, such as edges, textures, and shapes. These embeddings can be used for tasks such as object recognition or image captioning.

In summary, embeddings are a crucial component of many modern data-driven technologies. They are constructed using various techniques, including dimensionality reduction and machine learning algorithms, and are useful for several applications in fields such as machine learning, data analysis, and computer vision.

5.2 Locally linear embedding

One of the most important tasks in the data science world is feature selection and extraction. Let us imagine you have a dataset with thousands of features and you want to train a model on it. This number of features can be a problem. The computation power needed to train on such data is too high and not efficient. Feature selection [15], refers to choosing a subset of our features that are more useful to us. However, sometimes finding these features is not that easy and features can be complicated. In this case, we have another concept named feature extraction. Feature extraction algorithms [16], are techniques to make data more relevant and easy to analyze and compute. One of these techniques that is a nonlinear algorithm is *Locally Linear Embedding*.

Locally linear embedding (LLE) [8] is an embedding method, which is a dimension reduction method that we can use for manifold learning and feature extraction. The algorithm is based on a simple principle, "the points that are close to each other in high-dimensional space must be close to each other in reduced space" and this principle also works for far points. You can imagine that LLE takes a folded line and straightens it in a way that close points in that folded line are still close in a straight line in embedded space. For illustration, in Fig. 5.2 there are sphere shaped data points in $\mathbb{R}^3$ space and with LLE we can transform this sphere into a $\mathbb{R}^2$ space and by saving the features, the points that were close to each other in $\mathbb{R}^3$ space are still close to each other in $\mathbb{R}^2$ space. In fact, we just unfolded the sphere with the LLE algorithm. Hence, we can summarize the LLE algorithm in the three following steps:

1. For each high-dimensional data point $\{x_i \in \mathbb{R}^d\}_{i=1}^n$, find k nearest neighbors (we will discuss this algorithm in the next section).
2. Find weights between every point and its neighbors.

3. Find $\{y_i \in \mathbb{R}^p\}_{i=1}^n$, which are the new coordinates in reduced space, where $p \ll d$.

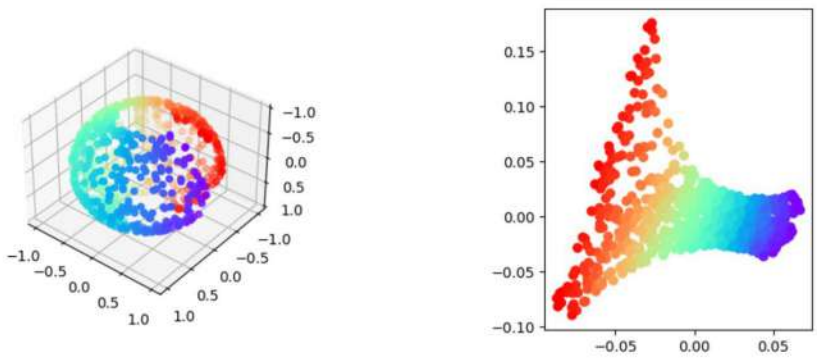

FIGURE 5.2 Unfolding (embedding) a sphere from 3D space into 2D space with LLE.

5.2.1 k nearest neighbors

A classic supervised machine learning algorithm for classification is k-nearest-neighbors algorithms [17]. k nearest neighbors (k-NN) is a non-parametric supervised algorithm, which means there is no learnable parameter in this algorithm. The idea is when we want to predict the new point x belongs to which class, we find the k nearest neighbors of it with a distance metric and predict the class that has a high number of neighbors. As you may have guessed, the k-NN algorithm does more computation in test time rather than training time. Parameter k is a hyperparameter that we should choose based on our data and experiment.

As mentioned, different metrics can be used to calculate the distance between points, some of which will be explained below.

5.2.2 Distance metrics

A metric distance is a function that defines the distance between two elements, which can be anything like data points in $\mathbb{R}^N$ space or words in a dictionary. In this case, our elements are $\mathbb{R}^N$ points and if $\mathrm{d}(x, y)$ is a distance function it must have these properties:

1. The distance from a point to itself is 0: $\mathrm{d}(x, x) = 0$.
2. The distance between two points cannot be negative: $\mathrm{d}(x, y) \geq 0$.
3. The distance between x and y must be the same as y and x: $\mathrm{d}(x, y) = \mathrm{d}(y, x)$.
4. For points x, y, and z: $\mathrm{d}(x, z) \leq \mathrm{d}(x, y) + \mathrm{d}(y, z)$.

Here, we explain some common metrics.

- Minkowski distance: Minkowski distance is a measure to calculate the distance between two points in N-dimensional space. You may hear about Euclidean distance or Manhattan distance, these distances are Minkowski distances. The general formula of the

Minkowski distance is:

$$distance(X, Y) = |\sum_{i=1}^{N} |x_i - y_i|^r|^{\frac{1}{r}},$$

where X and Y are two arbitrary N-dimensional points. Here, if you put $r = 1$ you obtain Manhattan distance, and $r = 2$ you obtain Euclidean distance.

- Jaccard distance: In Minkowski distance, we measure the distance between points in $\mathbb{R}^N$, but what if we want to define a metric for sets? For this, we introduce the Jaccard index and with that, we define Jaccard's distance. The Jaccard index is:

$$J_{index}(A,\ B) = \frac{|A \cap B|}{|A \cup B|},$$

where A and B are two arbitrary sets. This index measures the similarity between two sets, but in distances, we need the opposite, hence:

$$distance(A,\ B) = 1 - J_{index} = 1 - \frac{|A \cap B|}{|A \cup B|}.$$

- Cosine distance: Cosine distance shows the distance in direction view and points that are near with Cosine distance can be very far away in Minkowski distance. This metric can be calculated from the dot product of two vectors. Hence, we can use the following formula to construct the Cosine distance between two vectors X and Y, each of which has N elements:

$$distance(X, Y) = \cos(\alpha) = \frac{X.Y}{||X||\ ||Y||} = \frac{\sum_{i=1}^{N} X_i Y_i}{\sqrt{\sum_{i=1}^{N} X_i^2}\sqrt{\sum_{i=1}^{N} Y_i^2}}.$$

The introduced distances as well as other available distances, can help us gain information about how two points are similar or different. However, how to choose one among them? This depends on the intended application. However, in experience, it is shown that the Cosine metric works much better with face image datasets and by calculating the directions of face embeddings, we can achieve better evaluations by this metric. This is because the pose and dressing of the faces are different and the features are not that close. An example of that is ArcFace embedding [18] that uses Cosine similarity. However, in other datasets such as manifolds and curves datasets, the Euclidean distance works better, which is shown in Fig. 5.2 with a sphere shape.

5.2.3 Weights

After finding the k nearest neighbors for each point in our data, we have to find weights. In Fig. 5.3 you can see that we find five nearest neighbors of $x_i \in \mathbb{R}^3$ with Euclidean distance and then map them in $\mathbb{R}^2$ space with the found weights. Remember these weights are not the new distances!

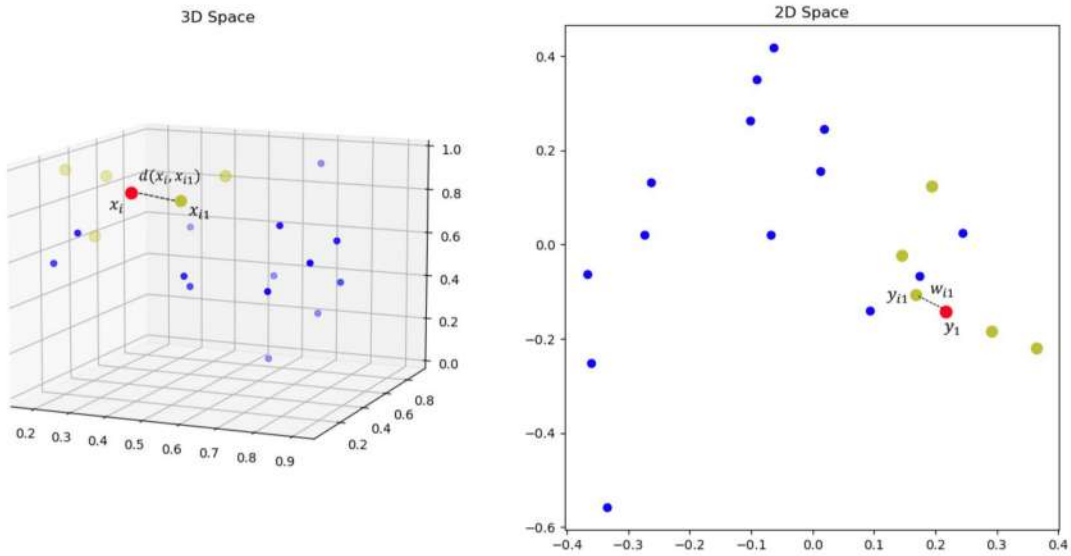

FIGURE 5.3 Embedding $\mathbb{R}^3$ space to $\mathbb{R}^2$ space using five neighbors.

The idea is to find the best weights for reconstruction. We will discuss this reconstruction later, but now we just make the weights to reduce our space dimension, to travel between folded space and embedded space. For weights, we are dealing with an optimization problem:

$$\underset{(W)}{\text{minimize}} \quad \epsilon(W) = \sum_{i=1}^{n} ||x_i - \sum_{j=1}^{k} w_{ij} x_{ij}||_2^2,$$

$$\text{subject to} \quad \sum_{j=1}^{k} w_{ij} = 1, \qquad i \in \{1, 2, ..., n\}.$$

In this problem n is the number of data points, k is the number of neighbors for each data point, W is the weight matrix, and w_{ij} is the weight between the ith data point to the jth neighbor, which is x_{ij}. As you can see, the term $\sum_{j=1}^{k} w_{ij} = 1$ means that the sum of the weights for every point is one. The question is can we have negative weights through this optimization problem?

5.2.4 Coordinates

After solving the weights optimization problem, in the next step, we should embed data to a lower-dimensional embedding space with our weights. Here again, we have an optimiza-

tion problem in finding the embedded coordinates:

$$\begin{aligned} \underset{(Y)}{\text{minimize}} \quad & \sum_{i=1}^{n} ||y_i - \sum_{j=1}^{n} w_{ij} y_j||_2^2, \\ \text{subject to} \quad & \frac{1}{n} \sum_{i=1}^{n} y_i y_i^T = I, \\ & \sum_{i=1}^{n} y_i = 0, \end{aligned}$$

where I is the identity matrix, Y are the new coordinates in reduced space ($y_i \in \mathbb{R}^p$ in which p is the dimension of the reduced space) and w_{ij} is the weight, we calculated for data point x_i and its jth neighbor.

In the above formula, there are points that are not the neighbors of point x_i, so there are no weights for them and in this case, we put zero for their weights:

$$w_{ij} = \begin{cases} w_{ij}, & \text{if } x_j \in k\text{−NN}(x_i), \\ 0, & \text{otherwise.} \end{cases}$$

Another feature that we understand from this optimization problem is that the sum of our embedded coordinates is 0. Hence, this optimization problem makes sure that the mean of embedded data is 0!

To solve this problem let us define a matrix 1 in which $1_i = [0, ..., 1, ...0]$ in a way that the ith element is one. Now, we can rewrite y_i as $Y^T 1_i$. After rewriting the second term, we can write the problem again in this new form:

$$\sum_{i=1}^{n} ||Y^T 1_i - Y^T \mathbf{w}_i||_2^2 = ||Y^T 1_i - Y^T w_i||_F^2,$$

where every row of W ($\mathbf{w}_i$) is the weight vector for each data point and F is the Frobenius norm of the matrix. We continue to simplify the problem:

$$\begin{aligned} ||Y^T 1_i - Y^T w_i||_F^2 &= \mathbf{tr}((I - W)YY^T(I - W)^T) \\ &= \mathbf{tr}(Y^T(I - W)^T(I - W)Y) = \mathbf{tr}(Y^T MY). \end{aligned}$$

Here, $\mathbf{tr}(.)$ means the trace of the matrix and we can define the Laplacian matrix of W as $(I - W)$, as every column of this matrix adds up to 1. If we consider $M = (I - W)^T(I - W)$, the final equation is:

$$\begin{aligned} \underset{(Y)}{\text{minimize}} \quad & \mathbf{tr}(Y^T MY), \\ \text{subject to} \quad & \frac{1}{n} Y^T Y = I, \end{aligned}$$

$$Y^T 1 = 0,$$

where 1 and 0 are matrices and $1 \in \mathbb{R}^n$ and $0 \in \mathbb{R}^p$. Now, we approach solving this problem by the assumption that the second constraint is satisfied implicitly [8] and write the Lagrangian for our final equation as:

$$\boldsymbol{L} = \boldsymbol{tr}(Y^T MY) - \boldsymbol{tr}(\Lambda^T(\frac{1}{n}Y^T Y - I)),$$

where $\Lambda \in \mathbb{R}^{n \times n}$ are Lagrangian multipliers in a diagonal form. To solve the equation, we have to obtain a derivative of $\mathcal{L}$ and put it equal to zero:

$$\frac{\partial \mathcal{L}}{\partial Y} = 2MY - \frac{2}{n}Y\Lambda = 0,$$
$$\Rightarrow MY = Y(\frac{1}{n}\Lambda),$$

which is the eigenvalue problem for M [19] and it means the columns of Y are the eigenvectors of M that are the diagonal elements of $\frac{1}{n}\Lambda$.

In summary, LLE is a technique used to reduce the dimensionality of high-dimensional data while preserving its local structure. LLE first constructs a graph by connecting each data point to its k nearest neighbors in the high-dimensional space. It then finds a set of low-dimensional embeddings that preserve the pairwise distances between neighboring points in the graph. Actually, the LLE technique is based on the assumption that high-dimensional data points can be approximated linearly by their nearest neighbors. The algorithm minimizes the difference between the original data and their reconstructed counterparts to reconstruct the high-dimensional data points. Implementation of standard LLE is reviewed below to refer them to variants of LLE implementation steps, which is the subject of our next section:

1. *Parameter selection*: Choose the number of nearest neighbors, k, and the number of dimensions for the low-dimensional embedding, p.
2. *Finding nearest neighbors*: Calculate the pairwise Euclidean distances between all data points then find the k nearest neighbors for each point.
3. *Computing weights*: For all data points, compute the weights that minimize the cost of reconstructing the point from its neighbors. This will be done by solving a least squares problem with constraints. The weights W can be computed by minimizing the cost function: $\epsilon(W) = \sum ||x_i - \sum(w_{ij}x_{ij})||_2^2$ for all i and all j in the set of k nearest neighbors of x_i, subject to the constraint that for each i, $\sum w_{ij} = 1$ for all j in the set of k nearest neighbors of x_i.
4. *Computing embeddings*: Weights will be utilized to compute the low-dimensional embeddings. This is done by solving an eigenvalue problem on the matrix $(I - W)^T(I - W)$, where I is the identity matrix and W is the weight matrix.

5.3 Variations of LLE

In the previous section, we reviewed implementation steps for standard LLE and now we can explain the implementation of variants of LLE including inverse LLE, kernel LLE, incremental LLE, robust LLE, weighted LLE, landmark LLE, supervised and semi-supervised LLE, using references to standard LLE steps.

5.3.1 Inverse LLE

Inverse LLE (ILLE) is a technique used for nonlinear dimensionality reduction [20]. It is designed to reconstruct high-dimensional data points from their low-dimensional embeddings. Inverse LLE is an inverse problem of LLE. It takes a set of low-dimensional embeddings obtained from LLE and aims to reconstruct the original high-dimensional data points that generated those embeddings. To achieve this, ILLE identifies the k nearest neighbors for each embedding in the low-dimensional space and uses them to interpolate the high-dimensional data points. This interpolation process is computationally expensive and involves solving a set of optimization problems to minimize the difference between the interpolated high-dimensional data points and the original data points [21]. Here are the ILLE implementation steps:

(a) *Finding nearest neighbors in embedded space*: For each low-dimensional point, find its k nearest neighbors in the embedded space.
(b) *Computing weights*: For all low-dimensional points, compute the weights that minimize the reconstruction cost of the point from its neighbors in the embedded space. This can be done by solving a constrained least squares problem, similar to the forward LLE process.
(c) *High-dimensional points reconstruction*: Weights are used for each high-dimensional point reconstruction from its neighbors. This is done by taking a weighted sum of the high-dimensional coordinates of the neighbors, using the weights computed in the previous step by the formula $x_i = \sum w_{ij} x_j$ for all j in the set of k nearest neighbors of x_i in the embedded space.

Inverse LLE can be useful in a variety of applications, such as image and speech processing, where the original high-dimensional data may not be easily accessible, but a low-dimensional representation is available. It can also be useful in data compression where the low-dimensional embeddings can be used to store the data more efficiently. LLE cannot be used in these cases due to a lack of original data.

5.3.2 Kernel LLE

Kernel Locally Linear Embedding (KLLE) is a nonlinear dimensionality reduction technique that aims to preserve the local structure of high-dimensional data in a low-dimensional space using a kernel function [22]. KLLE is an extension of the standard LLE algorithm and is particularly useful for nonlinear manifolds. Kernel methods including kernel LLE use a concept called the kernel trick [23]. In dimensionality reduction, the ker-

nel trick is utilized to enable performing linear operations on data that has been mapped into a higher-dimensional space using a nonlinear function called a kernel function. This is beneficial for kernel LLE as it can capture more intricate relationships between data points and preserve more of the original data's local structure. However, selecting an appropriate kernel function and parameter values is critical to prevent overfitting or underfitting of the data. The steps to implement KLLE are as follows:

(a) *Kernel mapping*: A kernel function $K(x, y)$ must be chosen to map the data points from the original space into a higher-dimensional space. Gaussian kernel, polynomial kernel, and sigmoid kernel are the usual kernel functions. For instance, the Gaussian kernel is defined as:

$$K(x, y) = e^{\frac{-||x-y||_2^2}{2\sigma^2}},$$

where the σ parameter controls the width of the Gaussian function. A discussion on the geometric properties of various kernel functions and a tutorial on how to implement them in Python are presented in [24–27].

(b) *Finding nearest neighbors*: For all data points in the mapped space, find its k nearest neighbors using the Euclidean distance metric. The distance between two points x_i and x_j in the mapped space is computed using the kernel function:

$$d(x_i, x_j) = K(x_i, x_i) - 2K(x_i, x_j) + K(x_j, x_j).$$

(c) *Computing weights*: For all data points, compute the weights that reconstruct the point from its neighbors in the mapped space in the best way. Computing weights needs least-squares problem solving with constraints, which is given by:

$$\underset{(W)}{\text{minimize}} \quad \epsilon(W) = \sum_{i=1}^{n} ||\Phi(x_i) - \sum_{j=1}^{k} w_{ij}\Phi(x_{ij})||_2^2,$$

$$\text{subject to} \quad \sum_{j=1}^{k} w_{ij} = 1, \qquad i \in \{1, 2, ..., n\}.$$

Here, $\Phi(x_i)$ is the mapped data point, $\Phi(x_{ij})$ is the jth nearest neighbor of $\Phi(x_i)$ in the mapped space, and w_{ij} is the weight to represent how well $\Phi(x_i)$ reconstruction is done from $\Phi(x_{ij})$. To solve this optimization problem, a matrix equation can be made and solved using a method like the Moore–Penrose pseudo-inverse:

$$W = (C + \lambda I)^{-1} 1,$$
$$W = \frac{W}{\sum w_{ij}}.$$

Here, C is the Gram matrix of the neighbors, λ is a regularization parameter, I is the identity matrix, and 1 is a column vector of ones. For more details, you can read about the Moore–Penrose method in [28].

(d) *Computing embeddings*: Compute the low-dimensional embedding vectors reconstructed by the weights calculated in the previous step in the best way. Here, solving another optimization problem is needed, which is given by:

$$\begin{aligned} \underset{(Y)}{\text{minimize}} \quad & \sum_{i=1}^{n} ||y_i - \sum_{j=1}^{n} w_{ij} y_j||_2^2, \\ \text{subject to} \quad & \frac{1}{n} \sum_{i=1}^{n} y_i y_i^T = I, \\ & \sum_{i=1}^{n} y_i = 0. \end{aligned}$$

Here, y_i is the low-dimensional embedding of $\Phi(x_i)$, y_j is the low-dimensional embedding of the jth nearest neighbor of $\Phi(x_i)$, and I is the identity matrix. To solve this optimization problem, you can compute the eigenvectors and eigenvalues of the matrix $M = (I - W)^T (I - W)$, and then select the eigenvectors corresponding to the p smallest non-zero eigenvalues as the columns of the embedding matrix Y. p is chosen by parameter selection in standard LLE, as we mentioned before.

Kernel LLE is a powerful technique for reducing dimensionality that can capture complex relationships between data points better than other LLE methods. KLLE utilizes the kernel function to map the original data into a higher-dimensional space where linear operations can be performed to maintain the local structure of the data. This makes KLLE much more useful for data with nonlinear dependencies, such as natural language processing or computer vision datasets. In addition, KLLE has been successfully used on unsupervised feature learning, allowing it to learn complex hierarchical illustrations of data with no explicit labeling.

5.3.3 Incremental LLE

Big data refers to large datasets that are not easy to process by traditional methods. Data with a large number of entries needs huge statistical power and can end up with a more false discovery rate. LLE is not always useful particularly for big datasets as it requires all the data to be stored in the memory (RAM). This can cause problems for large datasets that can not fit into the available RAM capacity. Incremental LEE (INLLE) is an extension of the LLE algorithm, which is a popular nonlinear dimensionality reduction technique [29]. The main advantage of incremental LEE is that it can handle large datasets that cannot be stored in memory by processing the data incrementally, one batch at a time. The algorithm first computes the LLE embeddings for the initial batch of data and then updates the embeddings, when new data is added.

Incremental LLE is an approach that addresses memory problems by dividing the dataset into smaller batches and computing the low-dimensional embeddings for each batch, separately. Then, the embeddings are merged to obtain the final low-dimensional

representation of the entire dataset. In this way, incremental LLE can handle larger datasets than standard LLE without requiring excessive memory. However, there are some disadvantages to using incremental LLE. One drawback is that splitting the data into batches can result in a loss of information and introduce artifacts in the embeddings. Another issue is that the choice of batch size and overlap can affect the quality of the final embedding and require careful selection. The steps to implement INLEE are as follows:

(a) *Initialization*: Assume we already have n data points and the embedding is obtained by standard LLE methods. The eigenvectors Y are orthonormal, so the matrix Y is orthogonal. With respect to Section 5.2.4, we have $MY = Y(\frac{1}{n}\Lambda)$ and this can be restated as:

$$Y^T MY = \frac{1}{n}\Lambda.$$

Y represents the matrix of embedded coordinates obtained from the original data points. Let us consider truncated Y and so we have p eigenvalues. Λ represents the diagonal matrix of eigenvalues in the LLE formulation, M is the weighting matrix used in LLE to determine the local relationships between data points. Therefore Y is in $\mathbb{R}^{n\times p}$, Λ is in $\mathbb{R}^{p\times p}$, and M is in $\mathbb{R}^{n\times n}$.

(b) *Adding new data points*: Consider we have ℓ new data points. The equation becomes:

$$Y_U^T M_U Y_U = \frac{1}{n}\Lambda_U.$$

Here, the U index is related to *updated* points (union of old and new points) and so Y_U is in $\mathbb{R}^{(n+\ell)\times p}$ and M_U is in $\mathbb{R}^{(n+\ell)\times(n+\ell)}$.

(c) *Updating the eigenvalues*: Since we are considering the smallest eigenvalues when truncating, the eigenvalues in both Λ_U and Λ are very small, so we can say that we approximately have $\Lambda_U \simeq \Lambda$.

(d) *Updating the embedded coordinates*: Solve the following optimization problem to update the coordinates of new data points in embedded space:

$$\begin{aligned} \underset{(Y_U)}{\text{minimize}} \quad & ||Y_U^T M_U Y_U - \frac{1}{n}\Lambda||_F^2, \\ \text{subject to} \quad & \frac{1}{n}Y_U^T Y_U = I, \\ & Y_U^T 1 = 0. \end{aligned}$$

Here, $||.||_F$ represents the Frobenius norm.

(e) *Solving the optimization problem*: This optimization problem can be solved using the interior point method [30]. The Lagrangian of this problem is (by ignoring the second constraint):

$$\boldsymbol{L} = Y_U^T M_U Y_U - \frac{1}{n}\Lambda_F^2 - \boldsymbol{tr}(\Lambda^T(\frac{1}{n}Y_U^T Y_U - I)).$$

The derivative of this Lagrangian concerning Y_U is:

$$\frac{\partial \mathcal{L}}{\partial Y_U} = 4(Y_U^T M_U Y_U - \frac{1}{n}\Lambda) M_U Y_U .$$

(f) *Final embedding*: The solution $Y_U \in \mathbb{R}^{(n+\ell)\times p}$ obtained by optimization contains the row-wise p-dimensional embeddings of both old and new data.

Overall, incremental LLE is a powerful tool for dimensionality reduction in large, evolving datasets. It has applications in a variety of fields. By allowing for the incremental addition of new data points and the efficient computation of embeddings for large datasets, it opens up new possibilities for data analysis and machine learning. However, as with any algorithm, it is important to carefully tune the parameters and consider the limitations and assumptions of the method to obtain the best results.

The stability of incremental LLE can be influenced by the selection of the batch size. If the batch size is too small, the low-dimensional embeddings may not accurately represent the local structure of the data. Conversely, if the batch size is too large, incremental LLE may not be able to capture the global structure of the data. For instance, suppose there is a dataset with a complex and nonlinear structure, where the local structure varies greatly across the dataset. If the batch size is chosen to be very small, incremental LLE might only capture the local structure of each batch, resulting in low-dimensional embeddings that fail to reflect the global structure of the dataset. Alternatively, if the batch size is set to be very large, incremental LLE may not be able to capture the fine details of the local structure, and the embeddings might lose significant information.

In real-world applications, the batch size for incremental LLE depends on the size and complexity of the dataset, as well as the available computational resources. Finding an optimal batch size that balances capturing the local and global structure of the data may require some experimentation.

5.3.4 Robust LLE

An outlier is a data point that is significantly different from other data points in a dataset. Outliers can occur in any type of dataset, including numerical, categorical, and textual data. They can be the result of measurement errors, data processing errors, or represent actual extreme values. Outliers can cause several problems in data analysis, such as skewing statistical measures, affecting model accuracy, and reducing the interpretability of results. They can also lead to false assumptions about the underlying data distribution and relationships between variables. Therefore it is important to identify and handle outliers appropriately before conducting any analysis or modeling.

Robust Locally Linear Embedding (RLLE) is a variant of Locally Linear Embedding (LLE) that is designed to handle outliers and noise in the data [31,32]. RLLE achieves this by using a robust estimate of the reconstruction weights that are less sensitive to outliers. There are several methods to implement robust LLE and in this chapter we will discuss the implementation steps of RLLE using the least squares problem method given by [31]:

(a) *Initialization:* Initialize the reliability weights, biases, and PCA projection matrices for each data point x_i.

(b) *Iteration steps:* Iterate between the following steps until convergence:
i. For all data points like x_i, use PCA to minimize the error of weighted reconstruction by solving the following least squares problem:

$$\underset{(P_i)}{\text{minimize}} \quad \sum_{j=1}^{k}(a_{ij}e_{ij}) = \sum_{j=1}^{k} a_{ij}||x_{ij} - b_i - P_i y_{ij}||_2^2,$$

where $b_i \in \mathbb{R}^d$ is a bias, $P_i \in \mathbb{R}^{d\times p}$ is PCA projection matrix, $y_{ij} \in \mathbb{R}^p$ is the embedding of x_{ij}, a_{ij} for $j = 1$ to k are the reliability weights, and e_{ij} is the reconstruction error between the original data point x_{ij} and its reconstruction using the weighted PCA projection matrix and bias.
ii. Update the bias b_i using the following formula:

$$b_i = \frac{\sum_{j=1}^{k}(a_{ij}x_{ij})}{\sum_{j=1}^{k} a_{ij}}.$$

iii. Update the columns of P_i as the top p eigenvectors of the covariance matrix over the neighbors:

$$S_i = \frac{1}{k}\sum_{j=1}^{k} a_{ij}(x_{ij} - b_i)(x_{ij} - b_i)^T.$$

iv. Use the Huber formula to update the reliability weights $\{a_{ij}\}_{j=1}^{k}$:

$$a_{ij} = \begin{cases} 1, & \text{if } e_{ij} \le c_i, \\ \frac{c_i}{e_{ij}}, & \text{if } e_{ij} > c_i, \end{cases}$$

where e_{ij} is defined in previous steps and c_i is the mean error residual, i.e., $c_i = \frac{1}{k}\sum_{j=1}^{k} e_{ij}$.

(c) *Calculating mean weights:* Calculate the mean reliability weights over the neighbors of each point x_i as $s_i = \frac{1}{k}\sum_{j=1}^{k} a_{ij}$.

(d) *Weighting the objective function:* Weight the objective function of the standard LLE embedding optimization problem using the mean reliability weights to make the embeddings robust to outliers, the formula of which is given by:

$$\underset{(Y)}{\text{minimize}} \quad \sum_{i=1}^{n} s_i||y_i - \sum_{j=1}^{n}(w_{ij}y_j)||_2^2,$$

with the constraints in standard LLE embedding.

Hence, we have changed the optimization problem in a manner that is robust to outliers, so finding the optimum solution to this problem can lead us to an errorless embedding.

Stock prices, medical data, and customer data are some examples of datasets with probable outliers. In the context of dimensionality reduction techniques like LLE and RLLE, outliers can lead to distortions in the low-dimensional embeddings, making it difficult to accurately capture the structure of the original high-dimensional data. However, RLLE is designed to be more robust to outliers than LLE, by using a robust estimator of the local covariance matrix to better handle the influence of outliers. As a result, RLLE may be a better choice for datasets with outliers, compared to LLE.

5.3.5 Weighted LLE

Weighted LLE (WLLE) is an extension of LLE that allows for the incorporation of user-defined weights to control the contribution of each data point to the local reconstruction of its neighbors [33]. The idea behind WLLE is to adjust the reconstruction weights matrix in such a way that data points that are more important or informative for the task at hand receive a higher weight. Weighted LLE can be implemented using various methods, in this section we will discuss two of them including weighted LLE for deformed distributed data and weighted LLE using the probability of occurrence.

5.3.5.1 Weighted LLE for deformed distributed data

The steps to implement this method are as follows:

(a) *Computing weighted distance*: For all pairs of data points like x_i and x_j, compute the weighted distance using the formula:

$$dist(x_i, x_j) = \frac{||x_i - x_j||_2}{\alpha_i + \beta_i \frac{(x_i - x_j)^T \tau_i}{||x_i - x_j||_2}}.$$

Here, $||x_i - x_j||_2$ is the Euclidean distance between x_i and x_j, α_i and β_i are constants, and τ_i is a data-dependent parameter.

(b) *Calculating parameters*: Compute the parameters α_i, β_i, and τ_i using the formulas:

$$\tau_i = \frac{m_i}{||m_i||_2}, \quad \alpha_i = \frac{q_i}{c_1}, \quad \beta_i = \frac{||m_i||_2}{c_2},$$

where m_i is the average of vectors from x_i to its k nearest neighbors, q_i is the average of the squared lengths of these vectors, and c_1 and c_2 are constants that depend on the dimensionality of the input space, d [8].

(c) *Finding nearest neighbors*: Compute the k nearest neighbors for each data point using weighted distance. This makes the graph of k nearest neighbors.

(d) *Computing weights*: For all data points, compute the weights that minimize the cost of reconstructing the point from its neighbors. This needs to solve a constrained least squares problem, similar to the standard LLE process as we mentioned before.

(e) *Computing embeddings*: Compute the low-dimensional embeddings using weights. Here, solving an eigenvalue problem on the matrix $(I - W)^T (I - W)$ is needed, where I is the identity matrix and W is the weight matrix.

5.3.5.2 Weighted LLE using the probability of occurrence

In this method, similar steps need to be taken, but the distance and the Gram matrix are weighted by the probabilities of occurrence of the data points:

(a) *Computing weighted distance*: For all pairs of data points x_i and x_j, compute the weighted distance using the formula given by:

$$dist^2(x_i, x_j) = \frac{||x_i - x_j||_2^2}{p_i},$$

where p_i is the probability of occurrence of data point x_i.

(b) *Finding nearest neighbors*: This step is similar to the previous method.

(c) *Computing weighted Gram matrix*: Compute the Gram matrix and weight its elements by the probabilities of occurrence of the data points:

$$G_i(a, b) = \sqrt{p_i p_j} G_i(a, b),$$

where, p_i and p_j are the probabilities of occurrence of data points x_i and x_j, respectively.

(d) *Computing weights*: This step is also similar to the previous method and the standard LLE section.

(e) *Computing embeddings*: Compute the low-dimensional embeddings using weights similar to the previous method.

In both methods, the final steps of the algorithm are the same as in standard LLE. The most important difference is how distance and weight are computed.

The main advantage of weighted LLE is that it allows the user to incorporate prior knowledge or domain-specific information into the embedding process by assigning different weights to the data points.

5.3.6 Landmark LLE for big data

Landmark LLE (LLLE) is an extension of the LLE algorithm designed for handling big data [34]. LLE is a popular technique for nonlinear dimensionality reduction, but it is not scalable for datasets that cannot fit into memory. Landmark LLE solves this issue by processing data incrementally. As we have mentioned before in Section 5.3.3, Landmark LLE, similar to incremental LLE, is for handling large datasets but they are different in their approach. Landmark LLE is a technique that involves selecting a small subset of data points called landmarks and calculating low-dimensional embeddings for each data point concerning these landmarks. This technique helps reduce the computational complexity of the LLE algorithm by computing the pairwise distances only between the landmarks and data points,

rather than between all data points. Landmark LLE is especially useful when the dataset is too large to fit into memory but the landmarks can be loaded into memory. In contrast, incremental LLE is a technique that processes data in smaller batches, calculates the low-dimensional embeddings for each batch separately, and then combines them to produce the final embedding. This technique allows incremental LLE to handle larger datasets that exceed the available memory by processing only a subset of the data at a time. These are the steps to implement LLLE using locally linear landmarks:

(a) *Landmark selection*: Choose a subset point from data points $X \in \mathbb{R}^{d\times n}$ as your landmark and call it $\widetilde{X} \in \mathbb{R}^{d\times m}$, where $m \ll n$. This reveals that m landmarks are chosen from n main data points.

(b) *Calculating projection matrix*: Compute the projection matrix $\tilde{U} \in \mathbb{R}^{n\times m}$ that maps the $Y \in \mathbb{R}^{n\times p}$ to the $\tilde{Y} \in \mathbb{R}^{m\times p}$. This is given by $Y = \tilde{U}\tilde{Y}$. The projection will also work for the input space, which is given by $X^T = \tilde{U}\widetilde{X}^T$.

(c) *Optimization for finding projection matrix*: To find the optimal projection matrix, the following optimization problem must be solved:

$$\begin{aligned} \underset{(\tilde{U})}{\text{minimize}} \quad & \sum_{i=1}^{n} \|x_i - \widetilde{X}\tilde{u}_i\|_2^2, \\ \text{subject to} \quad & 1^T\tilde{u}_i = 1, \quad i \in \{1,\dots,n\}. \end{aligned}$$

The solution to this optimization problem is given by:

$$\tilde{u}_i = (\tilde{G}_i)^{-1}\frac{1}{1^T(\tilde{G}_i)^{-1}1},$$

where $\tilde{G}_i = (x_i 1^T - \widetilde{X})^T(x_i 1^T - \widetilde{X})$, and 1 is a vector of ones.

(d) *Landmark embedding*: Embeddings of the landmark points must be computed by solving the following optimization problem:

$$\begin{aligned} \underset{(\tilde{Y})}{\text{minimize}} \quad & \mathbf{tr}(\tilde{Y}^T\tilde{U}^T M\tilde{U}\tilde{Y}), \\ \text{subject to} \quad & \frac{1}{n}\tilde{Y}^T\tilde{U}^T\tilde{U}\tilde{Y} = I. \end{aligned}$$

Here, $\widetilde{M} = \tilde{U}^T M\tilde{U}$, and $\widetilde{M}$ is a square matrix of size $m \times m$, where m is the number of landmarks selected. The entries of the matrix $\widetilde{M}$ are computed based on the pairwise distances and relationships between the landmark points. The solution to this optimization problem is given by the p smallest eigenvectors of $\widetilde{M}$, not considering eigenvectors with zero eigenvalues.

(e) *Data embedding*: Eventually, use the obtained embeddings of the m landmarks in order to approximate the embeddings of all n data points, by the $Y = \tilde{U}\tilde{Y}$ formula as given in step (b).

Landmark LLE using locally linear landmarks is particularly useful for large datasets, where $m \ll n$ because computational complexity is highly reduced by only computing the embeddings for the m landmarks and then approximating the embeddings for the remaining data points.

5.3.7 Supervised and semi-supervised LLE

Locally Linear Embedding is a popular nonlinear dimensionality reduction technique that has been widely used in various fields. However, LLE suffers from several limitations, such as its inability to handle large datasets and the need for the entire dataset to be stored in memory. Semi-supervised LLE and supervised LLE are extensions of LLE that address some of these limitations. Supervised and semi-supervised LLE are different from landmark and incremental LLE in that they incorporate extra information to aid in the process of dimensionality reduction, whereas the latter two methods only rely on the inherent structure of the data itself.

5.3.7.1 Supervised LLE

Supervised LLE (SLLE) is a variant of LLE that incorporates class labels to guide the embedding process [35]. The objective of SLLE is to preserve the local structure of the data while also preserving the class labels. The steps to implement SLLE are as follows:

(a) *Distance matrix modification*: In SLLE, the Euclidean distance matrix $D \in \mathbb{R}^{n \times n}$ is changed to increase the inter-class variance of data artificially. This can be done by increasing the distances of points that are from different classes. The new modified distance matrix D' is given by:

$$D' = D + \theta(d_{max})(11^T - \Omega).$$

Here, $11^T \in \mathbb{R}^{n \times n}$ is the matrix of ones, $d_{max} \in \mathbb{R}$ represents the diameter of data given by:

$$d_{max} = max_{i,j}(||x_i - x_j||_2)$$

and Ω is a matrix with the following elements:

$$\Omega(i, j) = \begin{cases} 1, & \text{if } c_i \neq c_j, \\ 0, & \text{otherwise,} \end{cases}$$

where c_i represents the class label of x_i, and $\theta \in [0, 1]$. When $\theta = 0$, SLLE becomes LLE, which is no longer supervised. When $\theta = 1$, SLLE is fully supervised. When $\theta \in (0, 1)$, we have partially supervised SLLE that is neither supervised nor unsupervised.

(b) *k-nearest-neighbors graph*: Find the k-nearest-neighbors graph using the modified distances similar to the way we did in the standard LLE section.

(c) *LLE algorithm*: The rest of the algorithm is completely similar to the standard LLE. The only important point where SLLE differs from other LLE algorithms is its modified distance matrix.

5.3.7.2 Semi-supervised LLE

The semi-supervised LLE (SSLLE) algorithm has several advantages over its supervised counterpart [36]. For one, it can handle missing labels and noisy label assignments, which are common in real-world datasets. Additionally, it can be used to learn embeddings in a low-dimensional space that captures both the geometric structure and the semantic relationships between data points. This makes it useful for a wide range of applications, including image and speech recognition, natural language processing, and bioinformatics. The steps to implement SSLLE are as follows:

(a) *Distance matrix modification*: In semi-supervised LLE, not all data points have labels but a certain number of them do have labels. The distances are computed as:

$$D' = \begin{cases} (\sqrt{1 - e^{-D''^2/\gamma}} - \theta, & \text{if } c_i = c_j, \\ \sqrt{1 - e^{-D''^2/\gamma}}, & \text{if } x_i \text{ or } x_j \text{ is unlabeled}, \\ \sqrt{e^{D''^2/\gamma}}, & \text{otherwise.} \end{cases}$$

Here, $\gamma = \text{average}_{i,j}(||x_i - x_j||_2)$, $D''(i,j) = \frac{D(i,j)}{\sqrt{m_i \times m_j}}$, where $m_i = \text{average}_r(||x_i - x_r||_2)$ for all $r \in 1, ..., n$.

(b) *k-nearest-neighbors graph*: Similar to the supervised method.

(c) *LLE algorithm*: Similar to the supervised method.

The main technique of both supervised and semi-supervised methods is to compute modified distances between data points based on their class labels, and the rest of the problem can be easily addressed using standard LLE methods using modified distances.

In summary, semi-supervised LLE is a powerful technique for learning embeddings in a low-dimensional space that preserves both the geometric structure and the semantic relationships between data points. By incorporating label information in the optimization problem, it can handle missing or noisy labels and provide more informative embeddings. The algorithm has a wide range of applications and can be used in combination with other machine learning methods to improve performance on various tasks.

5.3.8 LLE with other manifold learning methods

LLE is a nonlinear method for embedding, but some of the parts are not nonlinear, such as finding the k nearest neighbors commonly uses Euclidean distance, which is a linear method. However, we can use other distances that are used in manifolds and curves that are not linear. In this section, we want to talk about *geodesic distance* [37] and we call this method ISOLLE. Geodesic distance is the shortest path between two data points on a manifold and finding this distance is not as simple as the Euclidean one. Here, we should travel in our manifold to find the shortest path between points and this calculation needs high-level mathematics. ISOLLE finds this distance with an interesting idea, in which we construct a graph with a k-NN algorithm that uses Euclidean distance and then finds the shortest path between two data points using a classic shortest path algorithm in

the weighted graphs. Since the weights are the Euclidean distance, a Dijkstra algorithm in this case will do the job in a good time complexity $O(n \log_2 n)$. By gathering all these ideas the final formula for geodesic distance is:

$$D_{ij}^{(g)} = \min_r \sum_{i=2}^{z} ||r_i - r_{i+1}||_2.$$

In this formula, z is a sequence of data points and $r_i \in \{x_i\}_{i=1}^n$ and matrix $D^{(g)}$ is the geodesic distance matrix after calculating our distance metrics. Then, the rest of ISOLLE is the same as LLE.

5.4 Implementation and use cases

5.4.1 How to use LLE in Python?

Utilizing LLE inside a Python environment is possible using the "sklearn.manifold" library. This library includes a considerable number of manifold learning embeddings and algorithms, which can be easily accessed. In this case, our goal is to make use of the library on the S curve manifold. S curve simply illustrates the Latin letter 'S', which is 3D shaped. In order to use the above-mentioned library, you can import it from "scikit-learn" and use it for embedding in the following way:

```
from sklearn.manifold import LocallyLinearEmbedding
embedding = LocallyLinearEmbedding (
        n_neighbors : number of neighbors,
        n_components : number reduced coordinates,
        reg : regularization constant,
        eigen_solver('auto', 'arpack', 'dense'):solver on eigenvectors,
        max_iter : maximum number of iterations,
        method : ('standard', 'hessian', 'modified', 'ltsa;),
 different variations of LLE that are in sklearn.manifold
)
```

Scikit-learn provides various facilities and hyperparameters, as is clear in the code, the method "LocallyLinearEmbedding" has an "n_neighbors" parameter. It is the number of nearest neighbors that must be extracted for all data points. "n_components" is the lower-dimensional space that the original data must reduce to it. The parameter "reg" is the regularization constant, which users are not advised to change, and the best practice is to leave it with its default values. Regularization is a sensitively important method, and scikit-learn's developers have found a useful functional constant for it. The "eigen_solver" as it literally sounds, is the solver on eigenvectors. The "max_iter" is the number of iterations to fit the weights for our embedding, and last but not least, the "method" is the algorithm that is used to find the embedding.

At this stage, our goal is to use LLE on the S curve but to achieve this, data points need to be created. The "s_curve" has already been implemented in the "sklearn.datasets" li-

brary as a dataset. A number of 3600 data points are utilized for the curve, then it has been plotted and the results are illustrated in Fig. 5.4.

```
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d

from sklearn import datasets

n_samples = 3600
S_points, S_color = datasets.make_s_curve(n_samples, random_state=0)

x, y, z = S_points.T
fig, ax = plt.subplots(subplot_kw={'projection' : '3d'})
ax.scatter(x, y, z, c=S_color, cmap=plt.cm.rainbow)
plt.show()
```

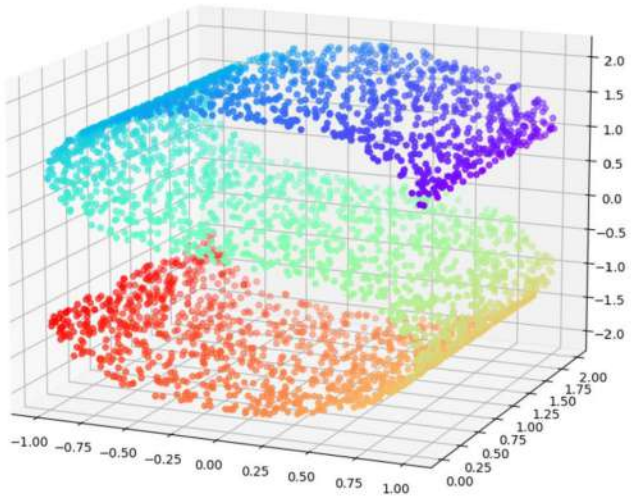

FIGURE 5.4 *S* curve dataset.

Next, our intention is to reduce the above-mentioned S curve dimensions, from $\mathbb{R}^3$ to $\mathbb{R}^2$ using LLE. The value 10 is determined to be the parameter for k nearest neighbors, and 2 is the value for the dimension of reduced space. The standard LLE is the method that has been used, so an LLE object is created and assigned to a variable with a similar name in the following piece of code. In line number 10, data points are fitted and transformed from the S curve and saved into the coordinates. Having been plotted, the reduced coordinates reveal 2D data points that can be seen in Fig. 5.5.

```
from sklearn.manifold import LocallyLinearEmbedding

n_neighbors = 10
n_components = 2

LLE = LocallyLinearEmbedding(method = "standard",
n_neighbors = n_neighbors,
n_components = n_components)

coordinates = LLE.fit_transform(S_points)
x, y = coordinates.T
plt.scatter(x, y, c = S_color, cmap=plt.cm.rainbow)
plt.show()
```

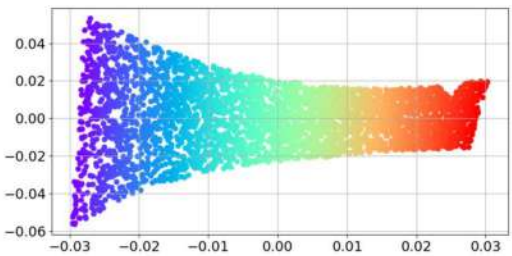

FIGURE 5.5 Embedded S curve with standard LLE.

As you can see in Fig. 5.5, the S curve has transformed successfully, satisfying the conditions mentioned in Section 5.2. Hence, the points have kept their distances in $\mathbb{R}^2$, relative to the distances they used to have in $\mathbb{R}^3$.

In addition, LLE in scikit-learn has different methods. In Fig. 5.6 the different outcomes of the S curve using standard LLE, modified LLE [38] and PCA are shown, respectively. The comparison between standard and modified LLE indicates that using LLE variants can provide useful reduced data for analysis purposes. Additionally, by making use of PCA, it is clear that both distances and shapes remain the same, but due to the difference in principles, it cannot be claimed that the reduced coordinates present the unfolded data and the same curve with reduced dimensions. Note that the output of LLE on the S curve has some differences in Figs. 5.5 and 5.6. The reason for these small differences is the random state and choices that the scikit-learn generates to solve the optimization problem. Now, let us test LLE on a more applied dataset in the next section.

5.4.2 Using LLE in MNIST

A well-known recommended dataset to use, while working on data science-related fields is the MNIST dataset. The MNIST dataset includes a massive number of images made of digits '0' to '9', where images are in 28×28 resolution. This resolution seems to be high and complex indeed, so it may cause a slow recognition for our application. In this case, our

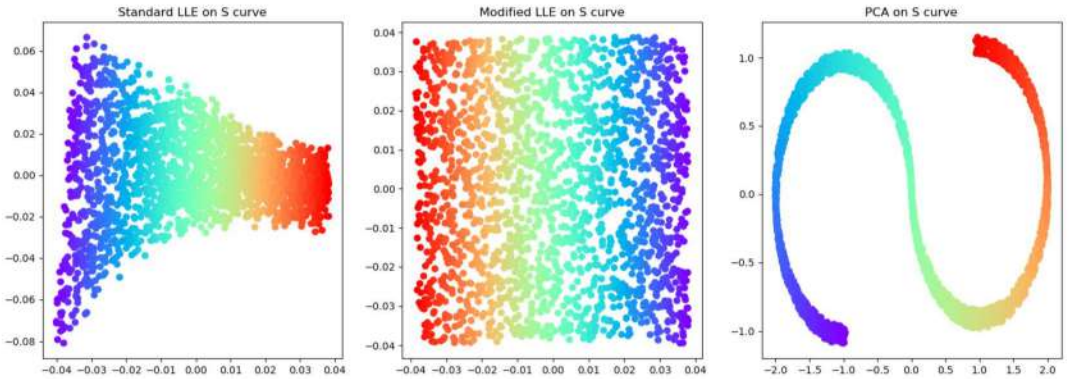

FIGURE 5.6 Comparing standard LLE, modified LLE, and PCA.

purpose is to use LLE for dimensionality reduction, followed by a k-nearest-neighbor classifier. For our demo project, the dataset needs to be loaded first, and because of time and processing issues, a smaller version of the dataset named "digits" is used in "scikit-learn". As is clear in the given code, the dataset is loaded into an axis, where X stands for images of digits, and y illustrates the digit that all images represent. All images in the mentioned dataset are in a resolution of 28×28, that simply means if images are flattened, the size of the high-dimensional space is 784, our goal is to make use of LLE in order to reduce it to the number 2.

```
from sklearn.datasets import load_digits

digits = load_digits(n_class=6)
X, y = digits.data, digits.target
fig:mnistembed
print("X Shape: ", X.shape)
print("Y Shape: ", y.shape)
```

Next, the dataset is split into train and test sets for evaluation. It is declared that 10 percent of existing data will be used for the test set and the other 90 percent for training:

```
from sklearn.model_selection import  train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size = 0.1, random_state=42)
```

Therefore LLE may be used for our dimensionality reduction purpose. As before, LLE needs to be imported from "sklearn.manifold", the number 10 as the value of k in k nearest neighbors, and the number 2 as the value for reduced space dimension. At the next stage, LLE is fitted, the weights are trained on the training dataset, and then both the train set and test set are transformed with it. Fig. 5.7 illustrates the embedding in $\mathbb{R}^2$.

```
import matplotlib.pyplot as plt
from sklearn.manifold import LocallyLinearEmbedding
embedding = LocallyLinearEmbedding(
        n_neighbors=10, n_components=2, method="modified")
transitions_train = embedding.fit_transform(X_train)
trainsitions_test = embedding.transform(X_test)
X, y = transitions_train.T
plt.scatter(X, y, c=y_train)
```

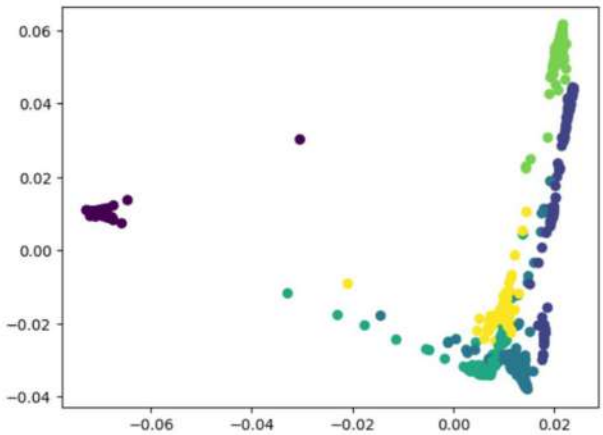

FIGURE 5.7 The digits dataset after embedding.

Now, k-NN may be used as a classifier. k-NN is mentioned specifically in Section 5.2.1 in order to help with LLE weights finding, here it is used for classification. We use k-NN in "scikit-learn" as the following approach:

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=10)
knn.fit(transitions_train, y_train)

pred = knn.predict(trainsitions_test)
```

Ultimately, model evaluation is done by checking *accuracy*, *precision*, and *recall*. Accuracy demonstrates how accurate the model is in label prediction, and it is calculated by dividing true predictions by the size of the test set. Precision represents the quality rate, and recall is the rate of quantity. Let TP be true positive, FP be false positive, TN be true negative, and FN be false negative, accuracy, precision, and recall can be calculated as follows:

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN},$$

$$\text{Precision} = \frac{TP}{TP + FP},$$
$$\text{Recall} = \frac{TP}{TP + FN}.$$

Built-in ratings are used in the sklearn.metric for the evaluation of this project.

```
from sklearn.metrics import accuracy_score,precision_score,recall_score

print("Accuracy: ", accuracy_score(y_test, pred))
print("Precision: ", precision_score(y_test, pred, average="macro"))
print("Recall: ", recall_score(y_test, pred, average="macro"))
```

The evaluation of the test leads us to an accuracy of 0.954, a precision of 0.954, and a recall of 0.955, which are significantly reliable rates.

5.5 Conclusion

LLE is a powerful nonlinear technique for dimensionality reduction, that has found many applications including in computer vision, bioinformatics, and image processing. In this chapter, we learned about a simple principle that LLE uses for dimension reduction and we defined an optimization problem to solve it. We transformed the problem into an eigenvector problem and then expanded the idea with other approaches by explaining the variants of LLE such as weighted LLE, inverse LLE, kernel LLE, etc. Then, we used LLE as a tool used in data science and machine learning applications. We used the scikit-learn library with Python to implement LLE and compared it with other dimension reduction methods like PCA. After that, we used this technique on the MNIST dataset to solve a categorical classification problem and evaluated it with some metrics. As we say, LLE is not the best embedding that could be found but it is not the worst either, if you have the right data, LLE can be very helpful.

References

[1] S.T. Roweis, L.K. Saul, Nonlinear dimensionality reduction by locally linear embedding, Science 290 (5500) (2000) 2323–2326.
[2] J.B. Tenenbaum, V.D. Silva, J.C. Langford, A global geometric framework for nonlinear dimensionality reduction, Science 290 (5500) (2000) 2319–2323.
[3] M. Ringnér, What is principal component analysis?, Nature Biotechnology 26 (3) (2008) 303–304.
[4] A. Ghaderi-Kangavari, J.A. Rad, M.D. Nunez, A general integrative neurocognitive modeling framework to jointly describe EEG and decision-making on single trials, Computational Brain & Behavior 6 (2023) 317–376.
[5] A. Ghaderi-Kangavari, K. Parand, R. Ebrahimpour, M.D. Nunez, J.A. Rad, How spatial attention affects the decision process: looking through the lens of Bayesian hierarchical diffusion model & EEG analysis, Journal of Cognitive Psychology 35 (2023) 456–479.
[6] A. Ghaderi-Kangavari, J.A. Rad, K. Parand, M.D. Nunez, Neuro-cognitive models of single-trial EEG measures describe latent effects of spatial attention during perceptual decision making, Journal of Mathematical Psychology 111 (2022) 102725.

[7] P. Xanthopoulos, P.M. Pardalos, T.B. Trafalis, Linear discriminant analysis, in: Robust Data Mining, 2013, pp. 27–33.
[8] B. Ghojogh, A. Ghodsi, F. Karray, M. Crowley, Locally linear embedding and its variants: tutorial and survey, arXiv preprint, arXiv:2011.10925, 2020.
[9] O. Bousquet, U. Luxburg, G. Rätsch (Eds.), Advanced Lectures on Machine Learning: ML Summer Schools 2003, Revised Lectures, Canberra, Australia, February 2–14, 2003, Tübingen, Germany, August 4–16, 2003, vol. 3176, Springer, 2011.
[10] C.M. Bishop, N.M. Nasrabadi, Pattern Recognition and Machine Learning, vol. 4, Springer, New York, 2006, p. 738.
[11] L. Van der Maaten, G. Hinton, Visualizing data using t-SNE, Journal of Machine Learning Research 9 (11) (2008).
[12] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016.
[13] K.P. Murphy, Machine Learning: A Probabilistic Perspective, MIT Press, 2012.
[14] S. Chao, C. Lihui, Feature dimension reduction for microarray data analysis using locally linear embedding, in: Proceedings of the 3rd Asia-Pacific Bioinformatics Conference, 2005, pp. 211–217.
[15] J. Li, K. Cheng, S. Wang, F. Morstatter, R.P. Trevino, J. Tang, H. Liu, Feature selection: a data perspective, ACM Computing Surveys 50 (6) (2017) 1–45.
[16] I. Guyon, S. Gunn, M. Nikravesh, L.A. Zadeh (Eds.), Feature Extraction: Foundations and Applications, vol. 207, Springer, 2008.
[17] T. Cover, P. Hart, Nearest neighbor pattern classification, IEEE Transactions on Information Theory 13 (1) (1967) 21–27.
[18] J. Deng, J. Guo, N. Xue, S. Zafeiriou, ArcFace: additive angular margin loss for deep face recognition, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 4690–4699.
[19] B. Ghojogh, F. Karray, M. Crowley, Locally linear image structural embedding for image structure manifold learning, in: International Conference on Image Analysis and Recognition, Springer International Publishing, Cham, August 2019, pp. 126–138.
[20] L.K. Saul, S.T. Roweis, Think globally, fit locally: unsupervised learning of low dimensional manifolds, Journal of Machine Learning Research 4 (Jun 2003) 119–155.
[21] M. Delkhosh, K. Parand, A.H. Hadian-Rasanan, A development of Lagrange interpolation, Part I: theory, arXiv preprint, arXiv:1904.12145, 2019.
[22] X. Zhao, S. Zhang, Facial expression recognition using local binary patterns and discriminant kernel locally linear embedding, EURASIP Journal on Advances in Signal Processing 2012 (2012) 1–9.
[23] J. Shawe-Taylor, N. Cristianini, Kernel Methods for Pattern Analysis, Cambridge University Press, 2004.
[24] A.H. Hadian Rasanan, S. Nedaei Janbesaraei, D. Baleanu, Fractional Chebyshev kernel functions: theory and application, in: J. Amani Rad, K. Parand, S. Chakraverty (Eds.), Learning with Fractional Orthogonal Kernel Classifiers in Support Vector Machines, in: Industrial and Applied Mathematics, Springer, Singapore, 2023.
[25] A.H. Hadian Rasanan, J. Amani Rad, M.S. Tameh, A. Atangana, Fractional Jacobi kernel functions: theory and application, in: J. Amani Rad, K. Parand, S. Chakraverty (Eds.), Learning with Fractional Orthogonal Kernel Classifiers in Support Vector Machines, in: Industrial and Applied Mathematics, Springer, Singapore, 2023.
[26] A.H. Hadian Rasanan, S. Nedaei Janbesaraei, A. Azmoon, M. Akhavan, J. Amani Rad, Classification using orthogonal kernel functions: tutorial on ORSVM package, in: J. Amani Rad, K. Parand, S. Chakraverty (Eds.), Learning with Fractional Orthogonal Kernel Classifiers in Support Vector Machines, in: Industrial and Applied Mathematics, Springer, Singapore, 2023.
[27] J.A. Rad, S. Chakraverty, K. Parand, Learning with Fractional Orthogonal Kernel Classifiers in Support Vector Machines: Theory, Algorithms, and Applications, Springer, 2023.
[28] J.C.A. Barata, M.S. Hussein, The Moore–Penrose pseudoinverse: a tutorial review of the theory, Brazilian Journal of Physics 42 (2012) 146–165.
[29] O. Kouropteva, O. Okun, M. Pietikäinen, Incremental locally linear embedding, Pattern Recognition 38 (10) (2005) 1764–1767.
[30] S.P. Boyd, L. Vandenberghe, Convex Optimization, Cambridge University Press, 2004.
[31] H. Chang, D.Y. Yeung, Robust locally linear embedding, Pattern Recognition 39 (6) (2006) 1053–1065.

[32] Y. Zhang, D. Ye, Y. Liu, Robust locally linear embedding algorithm for machinery fault diagnosis, Neurocomputing 273 (2018) 323–332.
[33] Y. Pan, S.S. Ge, A. Al Mamun, Weighted locally linear embedding for dimension reduction, Pattern Recognition 42 (5) (2009) 798–811.
[34] M. Vladymyrov, M.Á. Carreira-Perpinán, Locally linear landmarks for large-scale manifold learning, in: Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2013, Prague, Czech Republic, September 23–27, 2013, Proceedings, Part III 13, Springer, Berlin, Heidelberg, 2013, pp. 256–271.
[35] D. De Ridder, O. Kouropteva, O. Okun, M. Pietikäinen, R.P. Duin, Supervised locally linear embedding, in: International Conference on Artificial Neural Networks, Springer, Berlin, Heidelberg, June 2003, pp. 333–341.
[36] S. Zhang, K.W. Chau, Dimension reduction using semi-supervised locally linear embedding for plant leaf classification, in: Emerging Intelligent Computing Technology and Applications: 5th International Conference on Intelligent Computing, ICIC 2009, Ulsan, South Korea, September 16–19, 2009, Proceedings 5, Springer, Berlin, Heidelberg, 2009, pp. 948–955.
[37] B. Ghojogh, A. Ghodsi, F. Karray, M. Crowley, Multidimensional scaling, Sammon mapping, and isomap: tutorial and survey, arXiv preprint, arXiv:2009.08136, 2020.
[38] Z. Zhang, H. Zha, Principal manifolds and nonlinear dimensionality reduction via tangent space alignment, SIAM Journal on Scientific Computing 26 (1) (2004) 313–338.