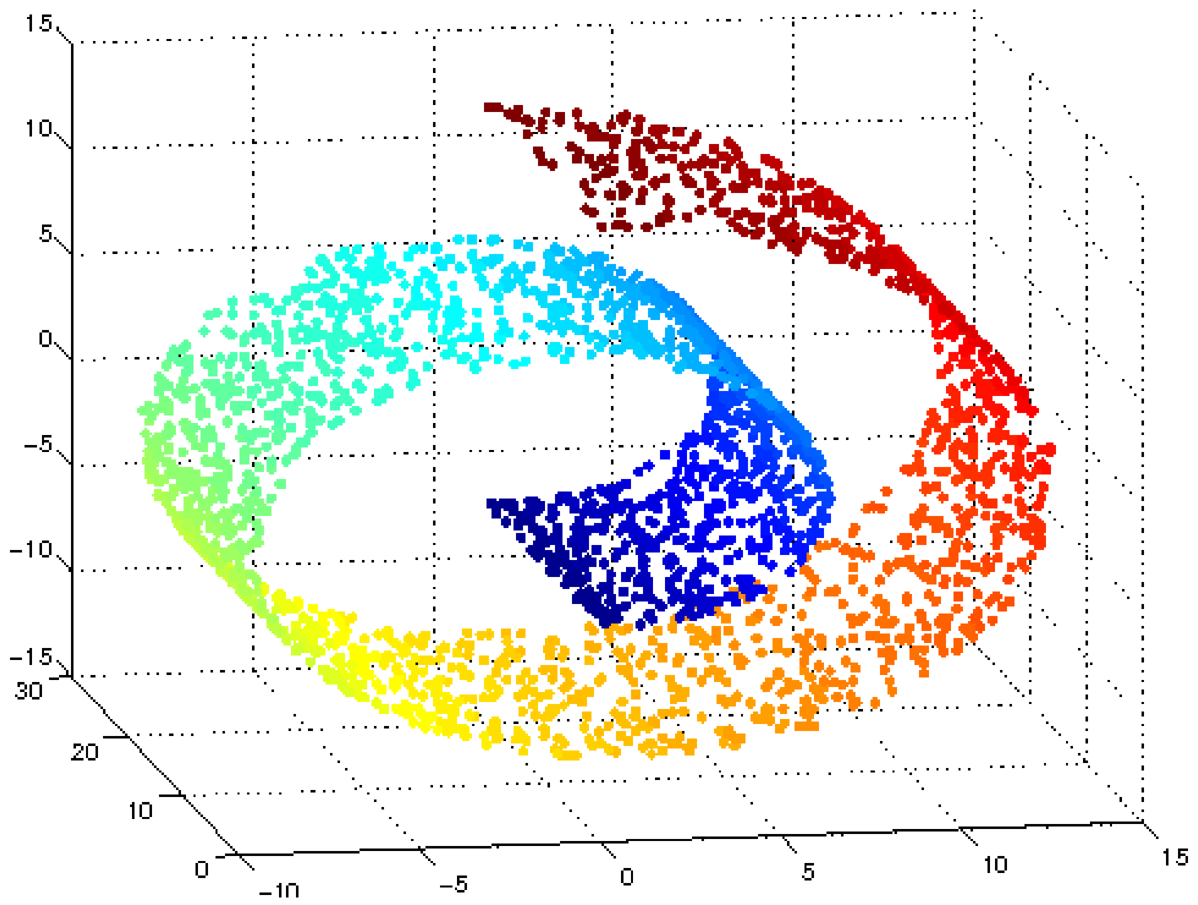
K. N. Toosi
University of Technology



Learning the Manifolds! Comparative Analysis of Different Dimensionality Reduction Methods

By Mahdi Lotfi, Araz Abedini

Dimensionality reduction simplifies high-dimensional data for easier visualization and analysis. In this project, we will implement and apply linear and non-linear dimensionality reduction methods to real-world datasets to reveal and visualize the hidden structures.

1. Implementation of PCA

Let's warm up by implementing the simplest and most straightforward method for dimensionality reduction: **P**rincipal **C**omponent **A**nalysis!

1.1. Implement the algorithm

Open the `pca.py` file and begin by completing the `PCA` class.

🚨 **Implementation note:** You need to vectorize your code using concepts from the labs. Avoid using loops for vector calculations; only use loops for the outermost iteration of the algorithms, alternating between computing the centroid and assigning points to clusters. Violating this incurs a **50% deduction from the related task score**. Also keep in mind that you may only use **basic NumPy features** to implement your algorithms. Using `numpy.linalg` or external libraries like `scikit-learn` is strictly prohibited unless explicitly allowed.

💡 **Coding practices:** Focus on writing clean, readable code by applying best practices like **Object-Oriented Programming (OOP)**. The cleaner and more well-structured your code is, the higher your chances of earning bonus points.

🧩 **Option:** Can you prove that the projection directions chosen by PCA are those that maximize the variance of the projected data points? Provide your argument using LaTeX.

1.2. Test on linear data

Complete the `generate_plane` function in the `dataset.py` file to create a linear hyperplane in a d-dimensional space. Add Gaussian noise to the generated points and apply a labeling strategy for improved visualization. For example, if you are working with four classes, you could partition the hyperplane into a 4-block grid and assign a different label to each partition.

Visualize a 2-dimensional hyperplane within a 3-dimensional space, ensuring that the four classes are clearly represented.

Next, apply the PCA algorithm to the hyperplane to project the data into a 2-dimensional space. After the projection, visualize the results and analyze the outcomes.

1.3. Test on swiss roll

Load the Swiss Roll dataset using the `dataset.py` file. Visualize it to analyze its structure and key properties.

Next, open the `pca.py` file and complete the main block to apply PCA to the Swiss Roll dataset. Reduce the dimension with 2 components and visualize the transformed data to observe dimensionality reduction.

Finally, reconstruct the data from its lower-dimensional representation and compare it with the original dataset to evaluate the reconstruction error.

2. Implementation of Isomap

A more general method for dimensionality reduction is **Iso**metric **Map**ping, which is a type of manifold learning algorithm.

We will implement Isomap step-by-step to illustrate its effectiveness in preserving the geometric properties of data while reducing its dimensionality.

2.1. Calculate pairwise distances

Isomap operates by assessing the similarities between data points.

Before proceeding, let's ensure that we have defined the necessary metrics for measuring similarity. In the `geo.py` file, we will implement two types of neighborhoods:

- **`KNearestNeighbors`**: An adjacency matrix is constructed based on the k nearest neighbors. Specifically, by sorting the points in ascending order of their distances, the first k points are identified as neighbors of the current point, while all other points are excluded.
- **`EpsNeighborhood`**: Defines a neighborhood around a given point based on a specified radius, denoted as ε. Points that fall within this radius are considered neighbors of the current point, while those outside this distance are excluded from the neighborhood.

2.2. Compute geodesic graph

After constructing the adjacency matrix, Isomap requires the computation of geodesic distances, which represent the meaningful distances between each pair of data points on the graph. These distances capture the shortest paths through the graph, reflecting the true geometric relationships among the points.

Open the `isomap.py` file and complete the `Isomap._compute_geodesic_distances` method to build the geodesic graph. Utilize the `adj_calculator` to obtain the adjacency matrix, and apply an algorithm such as Dijkstra's to compute the geodesic distances between each pair of points.

Note that the result will be a matrix where the geodesic distance between points P_i and P_j is located at the (i, j) component of the matrix.

2.3. Apply PCA to the graph

After computing the geodesic distances, the next step is to perform dimensionality reduction by implementing the `Isomap._decompose` function. However, before proceeding, we must first convert the distance matrix into an inner product matrix.

You can achieve this transformation using the following equations:

$$C = I - \frac{1}{n}J \qquad (I)$$

$$B = -\frac{1}{2}CD^2C \qquad (II)$$

Where:

- C is the centering matrix.
- D^2 is the element-wise squared of the distances.
- n is the number of datapoints.
- I is the identity matrix with dimension n.
- J is an n-by-n matrix of all 1's.
- B is the inner product matrix (*Gram* matrix).

Finally, apply a simple linear dimensionality reduction algorithm—PCA from the previous task—to the inner product matrix. Then, integrate all steps by completing the `Isomap.fit_transform` function to obtain the final low-dimensional embeddings.

2.4. Apply to the swiss roll

Just like in Section 1.3, complete this part by loading the Swiss Roll dataset in `isomap.py`. Apply the Isomap algorithm to project the data into a 2-dimensional space. Finally, visualize the transformed data and analyze the result.

3. Implementation of Locally Linear Embedding

Locally **L**inear **E**mbedding is a nonlinear dimensionality reduction technique that focuses on preserving the local structure of data. Unlike Isomap, which relies on geodesic distances, LLE reconstructs each data point as a linear combination of its nearest neighbors. This approach ensures that local geometric properties are retained in the lower-dimensional representation.

3.1. Compute local transformations

LLE works by preserving local geometric relationships between data points. The first step in this process is to define the neighborhood structure, which determines how points are connected.

Once the neighborhood structure is established, we express each data point as a weighted sum of its neighbors. These weights capture the local linear relationships between points and will be used to compute the final lower-dimensional embedding.

Open the `lle.py` file and implement the `LLE._compute_weights` function to solve the following least squares optimization problem:

$$min_W \sum_i \left\| X_i - \sum_j W_{ij} X_j \right\|^2 \qquad (III)$$

Subject to the following constraint:

$$\sum_j W_{ij} = 1 \qquad (IV)$$

Where:

- X_i represents a given data point.
- W_{ij} are the reconstruction weights for its neighbors.

💡 **Hint:** To satisfy the linear constraint while solving the least squares problem, you can express one of the weights as a linear combination of the others and solve for the remaining weights. For example, you may consider:

$$W_{i0} = 1 - \sum_{j=1} W_{ij} \qquad (V)$$

Then, solve the least squares problem for the remaining weights.

Write out your complete solution and reasoning step by step on paper. Using LaTeX is optional.

⚙️ **Using packages:** You may use `numpy.linalg.solve` method to compute the least squares solutions.

🚨 **Implementation note:** Your algorithms must be computationally stable. Ensure they can handle singular matrices when solving least squares problems by either adding a regularization parameter or implementing a robust strategy to detect and handle numerical instability.

3.2. Find the embeddings

After obtaining the weight matrix, the next step is to find the optimal low-dimensional representation that best preserves the local structure. This is done by solving the following optimization problem:

$$\min_Y \sum_i \left\| Y_i - \sum_j W_{ij} Y_j \right\|^2 \qquad (VI)$$

Where:

- Y_i represents the low-dimensional embedding of the data point X_i.
- W_{ij} are the reconstruction weights computed in the previous step.

The optimal low-dimensional representation Y can be obtained using the eigenvectors corresponding to the n smallest non-zero eigenvalues of the following matrix:

$$M = (I - W)^T (I - W) \qquad (VII)$$

Now, open the `lle.py` file and complete the `LLE._compute_embedding` function to implement the described method.

⚙️ **Using packages:** You can use `numpy.linalg.eig` function to perform eigenvalue decomposition.

🧩 **Option:** Can you prove that the eigenvectors corresponding to the n smallest nonzero eigenvalues of the equation (VII) are the unique solutions that minimize the equation (VI)? Provide your argument using LaTeX.

Finally, integrate all steps by completing the `LLE.fit_transform` function to obtain the final low-dimensional embeddings.

3.3. Implement the evaluation metrics

Now it's time to compare the implementations! How can we determine whether a dimensionality reduction algorithm is performing well on a specific dataset using statistical metrics?

There are various metrics we can use to evaluate these algorithms. Here, we will implement a simple yet effective one: *trustworthiness*.

The trustworthiness score $T(k)$ is defined as:

$$T(k) = 1 - \frac{2}{nk(2n-3k-1)} \sum_{i=1}^{n} \sum_{j \in N_i^k} max(0,\ (r(i,\ j) - k)) \qquad (VIII)$$

Where:

- n is the number of data points.
- k is the number of nearest neighbors considered.
- N_i^k is the set of the k nearest neighbors of point i in the original high-dimensional space.
- $r(i,\ j)$ is the rank of point j in the sorted list of nearest neighbors of i in the lower-dimensional space.

This sum penalizes points that were close in the original space but are far apart in the reduced space.

Open the `metrics.py` file and complete the **`trustworthiness`** function to compute the formula described earlier.

3.4. Compare the algorithms

Finally, integrate everything in `main.py` to compare, visualize, and evaluate the trustworthiness scores of the different dimensionality reduction algorithms.

💭 **Experiment:** Try different parameters for each algorithm to analyze their impact on the quality of dimensionality reduction. Observe how changing parameters affects the structure preservation for each method.

4. Learning the Manifold of Images

Toy datasets are simple, but real-world data is more challenging. Here, we apply our implementations to the *ORL* (Olivetti Faces) dataset, which contains 400 grayscale face images from 40 individuals.

Use the `dataset.py` file to load faces, a reduced version of ORL dataset which contains 32×32 images, flattened into 1024-dimensional vectors.

You need to implement the following sections in the `main.py` file.

4.1. Apply PCA to dataset

We will start by applying our simplest algorithm, PCA, to the dataset. Use your PCA implementation to reduce the dimensionality of the images, experimenting with different numbers of components. Visualize and compare sample images before and after reduction using the `PCA.inverse_transform` function.

In addition, you need to visualize the cumulative *Explained Variance Ratio* for this dataset.

The **E**xplained **V**ariance **R**atio measures how much of the total variance in the data is retained after dimensionality reduction:

$$EVR(i) = \frac{\lambda_i}{\sum_{j=1}^{d} \lambda_j} \qquad (IX)$$

Where:

- λ_i is the i-th largest eigenvalue of the covariance matrix.
- d is the total number of eigenvalues.

Add a property called `explained_variance_ratio_` to the `PCA` class to store the EVR vector after calling `transform` or `fit_transform` functions.

Finally, use the cumulative sum formula to visualize the EVR for the ORL dataset.

4.2. Compare and visualize algorithms

Similar to Section 3.4, load the ORL dataset, experiment with different algorithms and parameters and calculate the trustworthiness score to analyze the results.

5. Optional: Implementation of Laplacian Eigenmaps

Now it's time to put your knowledge to the test by implementing a dimensionality reduction algorithm from scratch.

Using reference [9], implement the *Laplacian Eigenmaps* algorithm, which reduces data dimensionality by leveraging the Laplacian graph spectrum.

Create a new file named `spectral.py` and implement the Laplacian Eigenmaps method based on your understanding and integrate this implementation into Sections 3.4 and 4.2 for dimensionality reduction experiments.

No additional hints are provided for this task!

References & Resources

1. https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm
2. https://www.science.org/doi/10.1126/science.290.5500.2323
3. https://www.science.org/doi/10.1126/science.290.5500.2319
4. https://en.wikipedia.org/wiki/Gram_matrix
5. https://en.wikipedia.org/wiki/Covariance_matrix
6. https://en.wikipedia.org/wiki/Centering_matrix
7. https://youtu.be/FgakZw6K1QQ?si=NGKmuoSXsIyQtJCd
8. Pattern Recognition and Machine Learning by Christopher M. Bishop
9. https://proceedings.neurips.cc/paper_files/paper/2001/hash/f106b7f99d2cb30c3db1c3cc0fde9ccb-Abstract.html