Dimensionality Reduction Techniques 2

Agenda

- Generalized Discriminant Analysis (GDA)
- Singular Value Decomposition
- t-Distributed Stochastic Neighbor Embedding (t-SNE)
- Autoencoders
- Isomap (Isometric Mapping)
- Use Case: Image Classification for Handwritten Digit Recognition

Generalized Discriminant Analysis (GDA)

- Generalized Discriminant Analysis (GDA) is an advanced extension of Linear Discriminant Analysis (LDA) designed to address the limitations of LDA in handling non-linearly separable data.
- While LDA works well when the classes can be separated by linear boundaries, it struggles in cases where the data's structure is more complex and nonlinear.
- GDA overcomes this by leveraging the kernel trick, a powerful technique often used in machine learning to implicitly map data into a higher-dimensional feature space without actually computing the mapping explicitly.
- In GDA, the kernel trick enables the transformation of the original data into a new, higher-dimensional space where the classes can be separated by linear boundaries, even if they are non-linearly separable in the original space.
- This allows GDA to capture more complex relationships between the features and improve class discrimination in cases where LDA's linear assumptions fall short.

Key Concepts of Generalized Discriminant Analysis

- Kernel Trick:
  - The kernel trick is a method used to implicitly compute dot products in a higher-dimensional space without the need to explicitly perform the transformation.
  - By applying a kernel function (such as polynomial, Gaussian, or radial basis function (RBF)), GDA maps the data into a high-dimensional feature space where linear techniques like LDA can be applied effectively.

  - The key advantage is that this mapping is done implicitly, making the computations feasible even when the dimensionality of the new space is very high.
- Non-Linear Separability:
  - In many real-world applications, data is not linearly separable. GDA addresses this by allowing for the separation of classes using non-linear decision boundaries.
  - For instance, in datasets with curved or intricate boundaries, LDA may fail because it can only model linear relationships.
  - GDA, by using the kernel trick, can find linear separations in a transformed space that correspond to non-linear separations in the original space.
- Extension of LDA:
  - GDA retains the fundamental idea of LDA, which is to maximize the ratio of between-class variance to within-class variance.
  - However, while LDA looks for linear combinations of features in the original space, GDA applies this idea in the higher-dimensional space after the kernel transformation, making it capable of dealing with more complex class distributions.
- Higher Dimensional Feature Space:
  - The kernel function used in GDA allows data points to be represented in a space with many more dimensions than the original feature space.
  - In this new space, classes that were not separable in the lower dimension may now be linearly separable. The choice of kernel (e.g., polynomial, Gaussian) defines how the data is transformed into this higher-dimensional space.
- Application:
  - GDA is particularly useful in problems where the data contains intricate patterns or non-linear relationships, which LDA cannot handle.
  - Common examples include image classification, pattern recognition, and bioinformatics, where classes may have complex boundaries that require non-linear separation.

Key Steps in Generalized Discriminant Analysis

- Select a Kernel Function:
  - A kernel function (such as the polynomial or RBF kernel) is chosen based on the nature of the data and the desired mapping to higher-dimensional space.
- Compute the Kernel Matrix:
  - Using the kernel function, compute the kernel matrix, which implicitly represents the dot products between all pairs of data points in the higher-dimensional space.

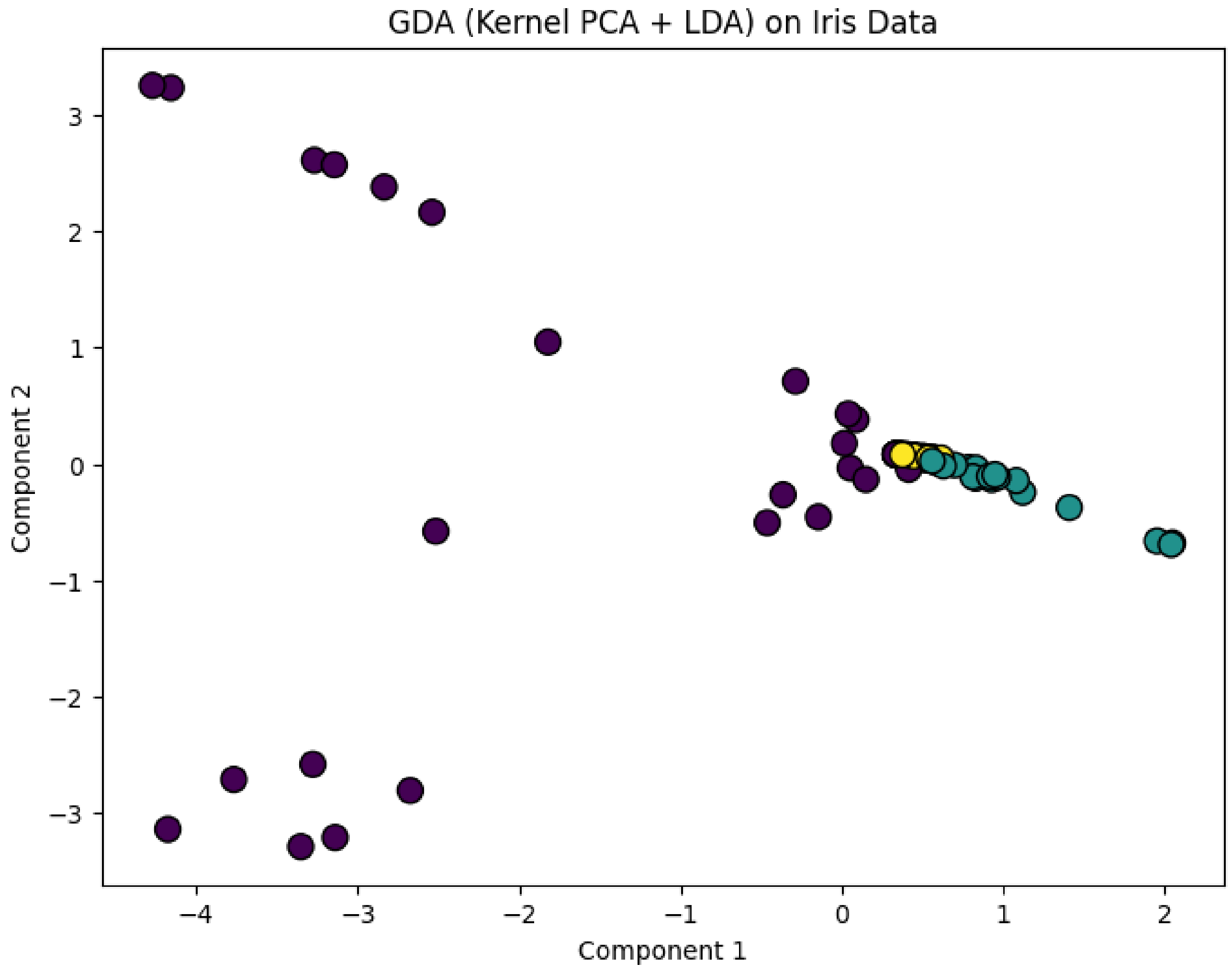
- Perform LDA in the Kernel Space:
    - Apply the principles of LDA (maximizing the between-class variance while minimizing the within-class variance) in the kernel-transformed space to find the optimal decision boundaries.
- Classify New Data Points:
    - Once the model is trained in the kernel space, it can be used to classify new data points by applying the same kernel function to map the new points and using the learned decision boundaries.

Example:

- Kernel PCA will transform the data into a higher-dimensional space.
- LDA will then be applied to this transformed data to perform classification.

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import KernelPCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score

# Load the Iris dataset
iris = load_iris()
X, y = iris.data, iris.target

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.3, r

# Step 1: Apply Kernel PCA (using RBF kernel)
kpca = KernelPCA(n_components=2, kernel='rbf', gamma=15)
X_kpca_train = kpca.fit_transform(X_train)
X_kpca_test = kpca.transform(X_test)

# Step 2: Apply LDA on the kernel-transformed data
lda = LDA(n_components=2)
X_lda_train = lda.fit_transform(X_kpca_train, y_train)
X_lda_test = lda.transform(X_kpca_test)

# Step 3: Train a classifier on the LDA-reduced data (e.g., Logistic Regression)
from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression()
classifier.fit(X_lda_train, y_train)

# Step 4: Make predictions on the test set
y_pred = classifier.predict(X_lda_test)

# Evaluate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of GDA (Kernel PCA + LDA): {accuracy:.2f}")

# Visualize the Kernel PCA + LDA transformed data
plt.figure(figsize=(8, 6))
plt.scatter(X_lda_train[:, 0], X_lda_train[:, 1], c=y_train, cmap='viridis', edge
plt.title("GDA (Kernel PCA + LDA) on Iris Data")
plt.xlabel("Component 1")
plt.ylabel("Component 2")
plt.show()
```

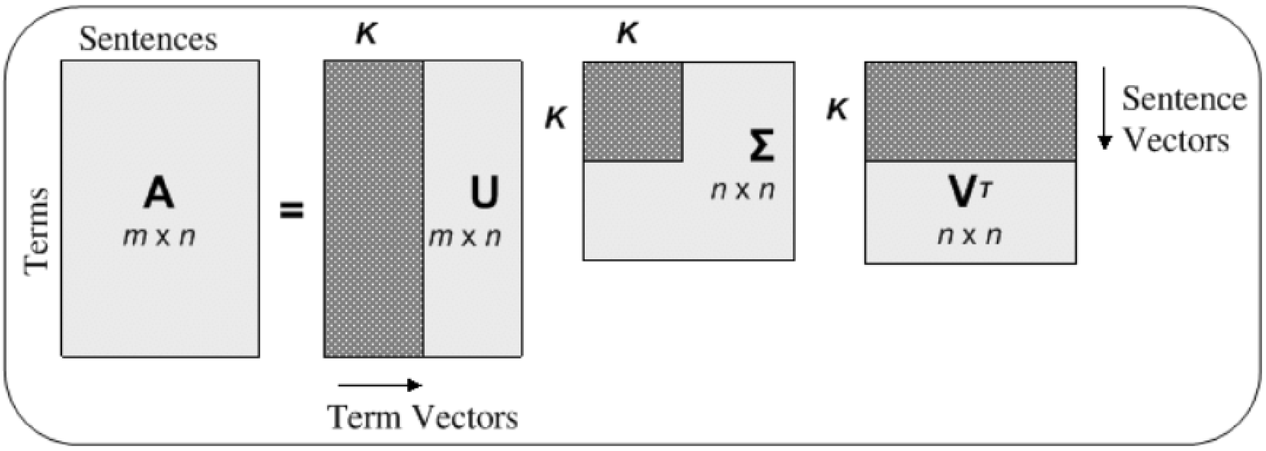
```
Accuracy of GDA (Kernel PCA + LDA): 0.64
```



Singular Value Decomposition (SVD)

- Singular Value Decomposition (SVD) is a crucial matrix factorization technique in linear algebra that decomposes a given matrix into three components: U, Σ (Sigma), and V^T (V transpose).
- This decomposition enables efficient analysis and manipulation of matrices, making SVD applicable in a variety of fields.
- It is commonly used for dimensionality reduction, where high-dimensional data is transformed into a lower-dimensional form while retaining significant information.
- SVD also plays a vital role in data compression, allowing for the storage and transmission of data with minimal loss.
- Additionally, it is widely utilized in image processing to enhance image quality and in recommendation systems to identify patterns and relationships within user-item interactions.

- By separating the matrix into its constituent components, SVD facilitates a deeper understanding of the underlying structure and properties of the data.



The Singular Value Decomposition (SVD) of an m × n rectangular matrix A

Mathematical Definition of SVD

Given a matrix A of size mxn, the Singular Value Decomposition is defined as:

$A = U\Sigma V^T$

Where:

- A is the original matrix of size mxn.
- U is an orthogonal matrix of size mxm, often called the left singular vectors.
- Σ is a diagonal matrix of size m×n, containing the singular values of A. These singular values are the square roots of the eigenvalues of A^TA.
- V^T is the transpose of the orthogonal matrix V of size n×n, where V contains the right singular vectors.

Components of SVD

- Left Singular Vectors (U): The columns of U are the eigenvectors of AA^T, which describe the direction of the variance in the row space of the original matrix.
- Singular Values (Σ): The diagonal entries in the matrix Σ are non-negative and arranged in decreasing order. They represent the magnitude of variance along each corresponding direction.
- Right Singular Vectors (V): The columns of V are the eigenvectors of A^TA, which describe the direction of the variance in the column space of the original matrix.

Applications of SVD

1. Dimensionality Reduction
   - SVD can be used to reduce the dimensions of large datasets. By truncating Σ to keep only the largest k singular values, we can approximate the original matrix with fewer components, capturing most of the variance and eliminating noise.
     - Principal Component Analysis (PCA): PCA is closely related to SVD. In fact, the decomposition used in PCA can be derived from SVD. PCA applies SVD to center the data and project it onto lower-dimensional spaces.
2. Latent Semantic Analysis (LSA) in NLP
   - In natural language processing, SVD is used in Latent Semantic Analysis (LSA) to identify relationships between words and concepts in textual data. SVD helps in reducing the dimensionality of word-document matrices and extracting latent structures.
3. Image Compression
   - SVD is often used to compress images. By truncating the singular value matrix, the image matrix can be approximated with fewer values, significantly reducing its size without losing much visual quality.
4. Recommendation Systems
   - SVD is used in recommendation systems (like those in Netflix or Amazon) to decompose user-item interaction matrices. This helps to identify latent factors representing user preferences and item attributes.
5. Truncated SVD for Approximation
   - Truncated SVD is a version of SVD where only the top k singular values and corresponding singular vectors are retained, reducing the dimensions and capturing the most essential data features.
   - The reduced approximation is:

     $A \approx U_k \Sigma_k V_k^T$

Where:

- U_k consists of the first k columns of U.
- Σ_k is a k×k diagonal matrix containing the top k singular values.
- V_k contains the first k columns of V.

The advantage of truncated SVD is that it provides a good approximation to the original matrix while reducing the amount of data we need to store and process.

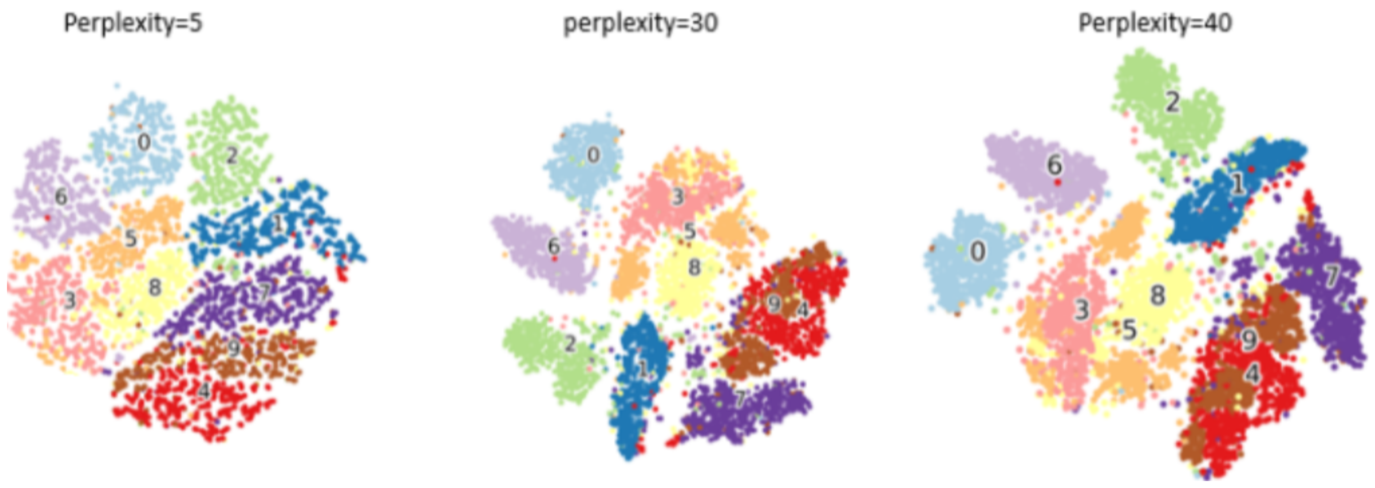
Example: Let's apply SVD to a matrix and interpret the results

```
import numpy as np

# Create a sample matrix A (m x n)
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Apply SVD
U, S, Vt = np.linalg.svd(A)

# Display the results
print("U matrix:\n", U)
print("Singular values (S):\n", S)
print("V^T matrix:\n", Vt)

# Reconstruct the original matrix using U, S, and V^T
S_diag = np.diag(S)
A_reconstructed = U @ S_diag @ Vt
print("Reconstructed A:\n", A_reconstructed)
```

```
U matrix:
 [[-0.21483724  0.88723069  0.40824829]
 [-0.52058739  0.24964395 -0.81649658]
 [-0.82633754 -0.38794278  0.40824829]]
Singular values (S):
 [1.68481034e+01 1.06836951e+00 4.41842475e-16]
V^T matrix:
 [[-0.47967118 -0.57236779 -0.66506441]
 [-0.77669099 -0.07568647  0.62531805]
 [-0.40824829  0.81649658 -0.40824829]]
Reconstructed A:
 [[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
```

t-Distributed Stochastic Neighbor Embedding (t-SNE)

- t-SNE (t-Distributed Stochastic Neighbor Embedding) is a powerful non-linear dimensionality reduction technique designed to visualize high-dimensional data in a lower-dimensional space, usually 2D or 3D.
- Developed by Laurens van der Maaten and Geoffrey Hinton in 2008, t-SNE excels in preserving local structures within the data while revealing global patterns, making it particularly effective for identifying clusters or manifolds.
- This capability makes it a preferred method for exploring complex high-dimensional datasets, such as word embeddings, image collections, and gene expression profiles.
- By transforming data into a visually interpretable format, t-SNE facilitates insights into the relationships and similarities between data points, enhancing the understanding of intricate patterns in diverse fields, including natural language processing and bioinformatics.
- Its intuitive visualizations have made t-SNE an essential tool in data analysis and exploration.



How t-SNE Works

- t-SNE converts high-dimensional distances between data points into probabilities that represent similarities. It tries to ensure that the similarity between points in the high-dimensional space is also reflected in the low-dimensional space.
- Here's a high-level breakdown of the steps in t-SNE:
    - Pairwise Similarities in High Dimensions:
        - For each point in the dataset, t-SNE computes the pairwise similarity with every other point.
        - This is done by modeling the similarities using a conditional probability distribution based on a Gaussian distribution (normal distribution). In other words, nearby points in high-dimensional space are assigned a high probability of being neighbors.
    - Pairwise Similarities in Low Dimensions:
        - The algorithm randomly initializes points in the lower-dimensional space (usually 2D or 3D) and then computes the pairwise similarity of these points using a Student's t-distribution (with one degree of freedom).
        - The t-distribution, which has heavier tails than the Gaussian distribution, is used to allow moderate distances in the low-dimensional space to have a larger effect and help prevent crowding of points.
    - Minimization of Divergence:
        - t-SNE attempts to minimize the Kullback-Leibler (KL) divergence between the two probability distributions: one in the high-dimensional space (based on the Gaussian) and one in the low-dimensional space (based on the t-distribution).
        - This is done through an optimization process (usually gradient descent), aiming to make similar points in high-dimensional space stay close together in

the low-dimensional space, while dissimilar points are pushed further apart.

Algorithm in Steps

- Input Data: The dataset with N points in a high-dimensional space.
- Compute Pairwise Similarities: For every pair of points in high-dimensional space, compute the similarity based on the conditional probability distribution.
- Initialize Points in Low Dimensions: Randomly initialize the points in the lower-dimensional space.
- Compute Low-Dimensional Similarities: Compute pairwise similarities between points in the low-dimensional space using a t-distribution.
- Minimize Divergence: Use gradient descent to minimize the difference (KL divergence) between the two similarity distributions (high-dimensional vs low-dimensional).
- Output: A lower-dimensional representation of the data, usually visualized as a 2D or 3D scatter plot.

Mathematical Formulation

- High-dimensional Similarity (Gaussian Distribution): The similarity between two points x_i and x_j in the high-dimensional space is modeled as a conditional probability:

  $$p_{j|i} = \frac{\exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma_i^2}\right)}{\sum_{k \neq i} \exp\left(-\frac{\|x_i - x_k\|^2}{2\sigma_i^2}\right)}$$

  Where σ_i is the variance of the Gaussian centered on point x_i.

- The joint probability is then symmetrized as:

  $$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N}$$

2. Low-dimensional Similarity (t-distribution): The similarity between the corresponding points y_i and $y_j$in the low-dimensional space is modeled using a Student's t-distribution:

   $$q_{ij} = \frac{\left(1 + \|y_i - y_j\|^2\right)^{-1}}{\sum_{k \neq l} \left(1 + \|y_k - y_l\|^2\right)^{-1}}$$

   This allows points to be pushed apart more forcefully if they are far from each other.

3. Cost Function: The objective of t-SNE is to minimize the KL divergence between the high-dimensional similarities p_{ij} and the low-dimensional similarities q_{ij}:

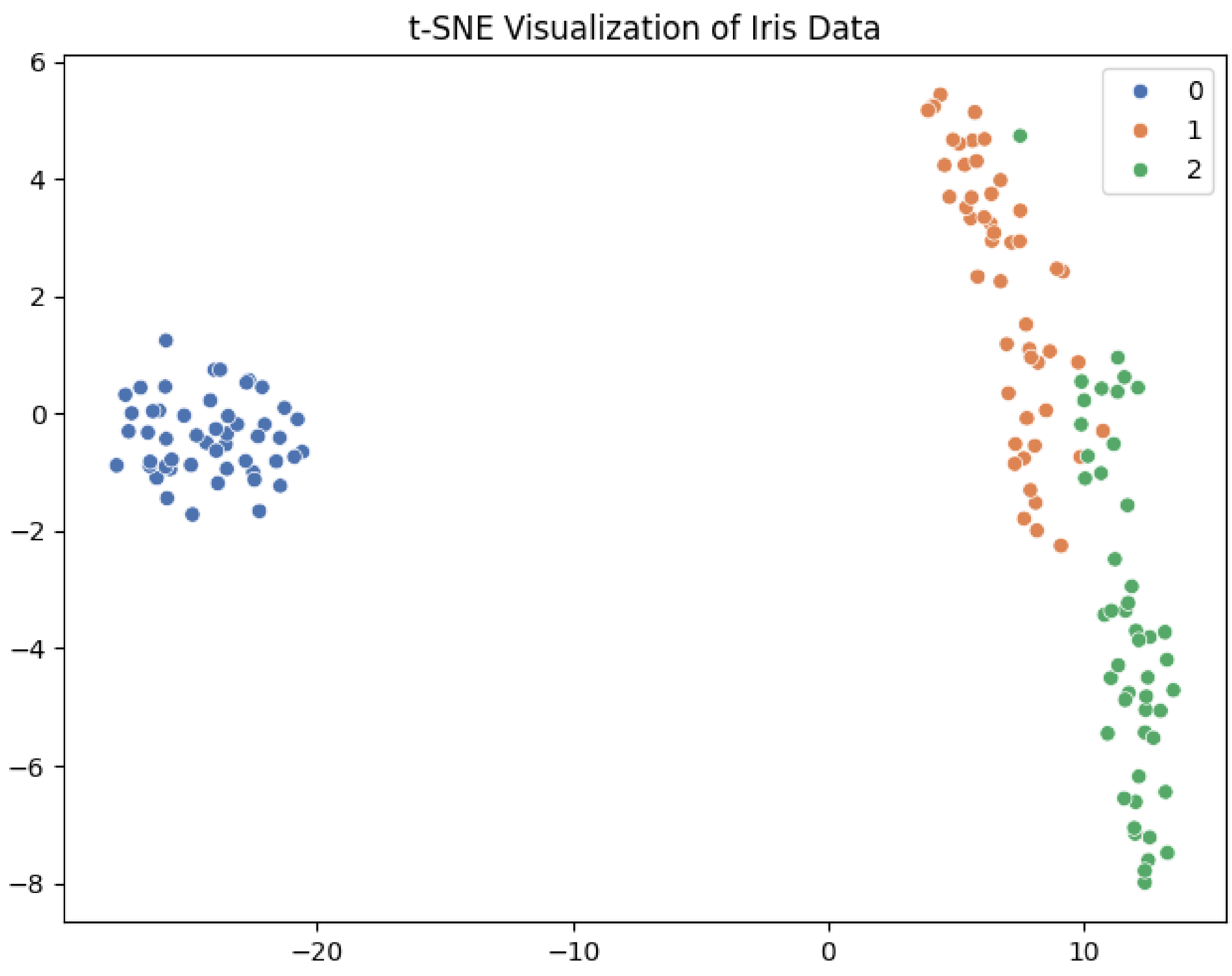
   $$C = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

   This is minimized using gradient descent.

Example:

- n_components: Number of dimensions to reduce the data to (commonly 2 or 3).
- perplexity: Controls the balance between local and global structure (recommended values between 5 and 50).
- learning_rate: Impacts the optimization process; typical values are between 10 and 1000.
- n_iter: Number of iterations for optimization (at least 250 is recommended).

```
from sklearn.manifold import TSNE
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
import seaborn as sns

# Load iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Apply t-SNE
tsne = TSNE(n_components=2, perplexity=30, random_state=42)
X_tsne = tsne.fit_transform(X)

# Visualize the result
plt.figure(figsize=(8, 6))
sns.scatterplot(x=X_tsne[:, 0], y=X_tsne[:, 1], hue=y, palette='deep')
plt.title('t-SNE Visualization of Iris Data')
plt.show()
```

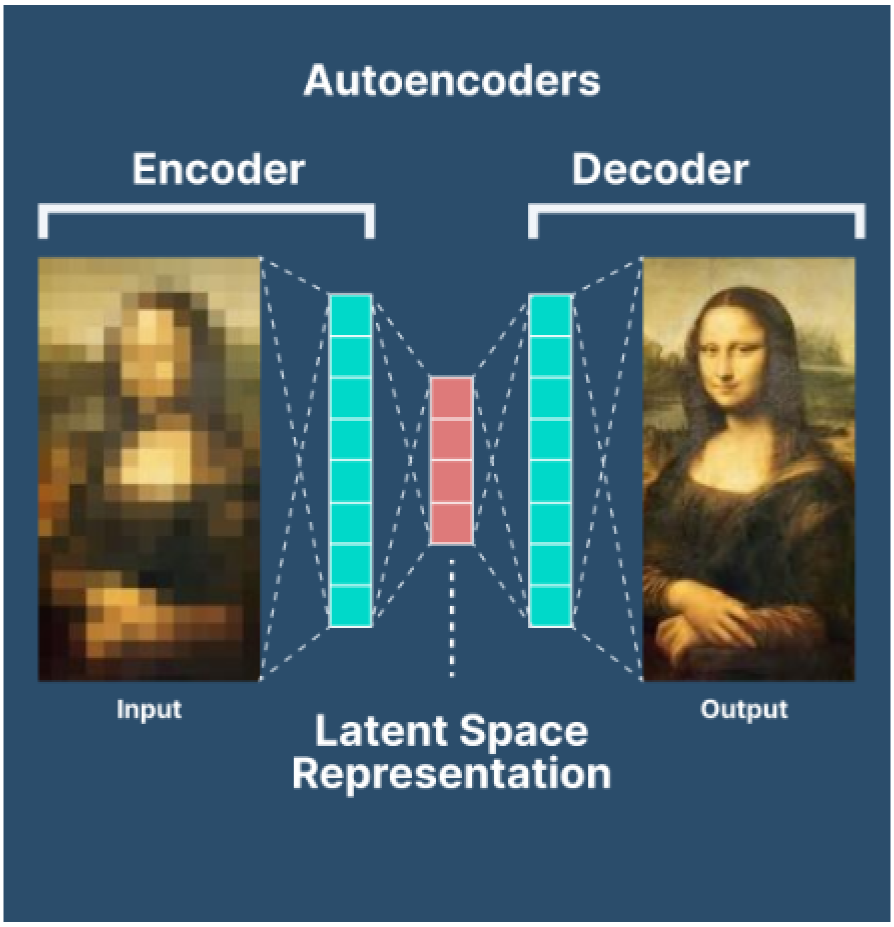


- This code provides a visual representation of the Iris dataset using t-SNE, allowing you to explore how well the dataset can be separated into distinct clusters based on the input features.
- It helps in assessing whether the features are useful for classification or further dimensionality reduction.

Autoencoders

- Autoencoders are a specialized type of artificial neural network primarily employed for unsupervised learning tasks.
- Their main objective is to learn a compact and efficient representation (encoding) of input data, which is useful for dimensionality reduction, noise reduction, and feature extraction.
- Structurally, autoencoders consist of two main parts: an encoder that compresses the input into a lower-dimensional representation and a decoder that reconstructs the original input from this compressed form.
- This design enables autoencoders to effectively capture the underlying patterns in the data while minimizing reconstruction error.

- They are widely used in various applications, including data compression, anomaly detection, and as generative models for creating new data points.
- By leveraging the learned representations, autoencoders facilitate improved performance in subsequent tasks such as classification and clustering, making them valuable tools in the machine learning landscape.



Structure of an Autoencoder

- Encoder: The encoder compresses the input data into a lower-dimensional representation (latent space or code). It captures the most relevant features of the data by learning to discard irrelevant information.
- Latent Space (Code): This is the compressed version of the input, often referred to as a bottleneck. The number of neurons in the bottleneck layer is smaller than in the input layer, forcing the autoencoder to learn a more compact representation.
- Decoder: The decoder reconstructs the original input from the compressed representation. The goal is to output data as close as possible to the input.

The overall process is designed to minimize the reconstruction error, which is the difference between the input and the reconstructed output.

Types of Autoencoder

1. **Vanilla Autoencoders:**
   - Basic form of an autoencoder.
   - The architecture is symmetrical: the encoder and decoder are mirrored.
   - Used for simple tasks like data compression and dimensionality reduction.
2. **Sparse Autoencoders:**
   - Introduces a sparsity constraint to ensure that only a small number of neurons in the hidden layer are active at a time.
   - Helps in learning more useful features and improving generalization.
3. **Denoising Autoencoders (DAE):**
   - Designed to reconstruct the original data from noisy input.
   - Useful for noise reduction in images or data cleaning.
   - The encoder learns to remove noise by trying to reconstruct the clean version of the input.
4. **Contractive Autoencoders (CAE):**
   - Introduces a penalty term that ensures that the learned representation is robust to small variations in the input.
   - Used for feature extraction and learning robust representations.
5. **Variational Autoencoders (VAE):**
   - A generative model that learns to encode data as a probability distribution, rather than fixed points in latent space.
   - Used for generating new data (e.g., synthetic images) by sampling from the latent space.
   - VAEs have probabilistic layers and are trained using the Kullback-Leibler (KL) divergence to enforce distribution learning.
6. **Convolutional Autoencoders (CAE):**
   - Uses convolutional layers for encoding and decoding, making it well-suited for image data.
   - Used for tasks such as image reconstruction and denoising.
   - Reduces the spatial resolution through pooling layers during encoding and restores it during decoding.

Working Mechanism of Autoencoders

1. Encoding:

- The encoder maps the input data X to a compressed form Z (latent space) using a series of transformations:

  $$Z=f(X)$$

where f is the function representing the encoder (usually a set of neural network layers).

2. Decoding:
   - The decoder maps the compressed data Z back to the original data space to form the reconstruction X':

     $$X' = g(Z)$$

where g is the function representing the decoder (another set of neural network layers).

3. Optimization:
   - The network is trained to minimize the reconstruction error, commonly measured as the mean squared error (MSE) between the input and the output:

     $$\text{Loss}=\|X-X'\|^2$$

The encoder and decoder are optimized simultaneously during training.

Example Code of Autoencoder in Python (Keras)

The model is trained for 50 epochs with a batch size of 32. The target for the model is the original input data (X_train), as autoencoders are trained to reconstruct their inputs.

```
from keras.layers import Input, Dense
from keras.models import Model
import numpy as np

# Example data
X_train = np.random.rand(1000, 20)  # 1000 samples, 20 features

# Define the input layer
input_layer = Input(shape=(20,))

# Encoder layers
encoded = Dense(10, activation='relu')(input_layer)  # Compress to 10 dimensions
encoded = Dense(5, activation='relu')(encoded)  # Further compress to 5 dimension

# Decoder layers
decoded = Dense(10, activation='relu')(encoded)  # Decompress to 10 dimensions
decoded = Dense(20, activation='sigmoid')(decoded)  # Decompress to original size

# Autoencoder model
autoencoder = Model(input_layer, decoded)

# Compile the model
autoencoder.compile(optimizer='adam', loss='mse')

# Train the autoencoder
autoencoder.fit(X_train, X_train, epochs=50, batch_size=32, shuffle=True)

# Encoder model for feature extraction
encoder = Model(input_layer, encoded)

# Encode the input data
encoded_data = encoder.predict(X_train)
print(encoded_data)
```

```
Epoch 1/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 0.0838
Epoch 2/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0828
Epoch 3/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0818
Epoch 4/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0813
Epoch 5/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0797
Epoch 6/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0798
Epoch 7/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0788
Epoch 8/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0779
Epoch 9/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0773
Epoch 10/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0775
Epoch 11/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0754
Epoch 12/50
```

```
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0755
Epoch 13/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0741
Epoch 14/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0734
Epoch 15/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0730
Epoch 16/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0712
Epoch 17/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0702
Epoch 18/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0700
Epoch 19/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0696
Epoch 20/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0684
Epoch 21/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0676
Epoch 22/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0667
Epoch 23/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0670
Epoch 24/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0664
Epoch 25/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0663
Epoch 26/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0661
Epoch 27/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0657
Epoch 28/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0652
Epoch 29/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0651
```

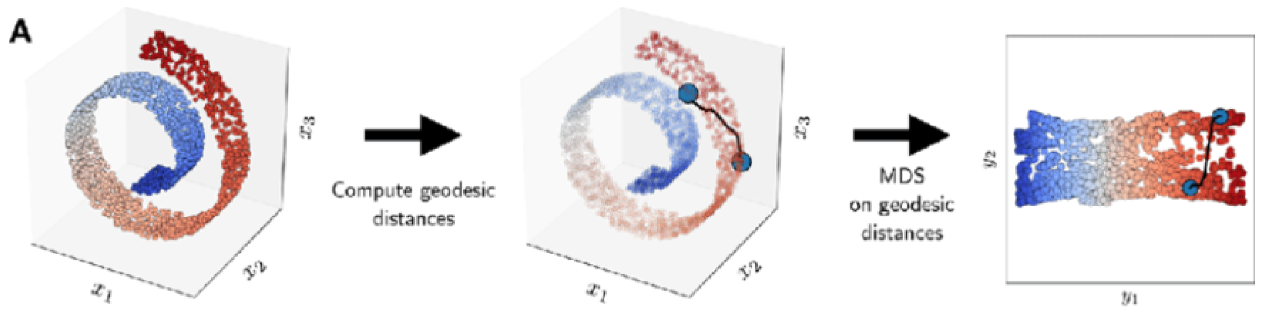
- Autoencoder's Purpose: The autoencoder is trained to reduce the dimensionality of the input data and then reconstruct it. The difference between the original and the reconstructed data is minimized using MSE.
- Latent Representation: After training, the encoder part of the model can be used to get a lower-dimensional (5D) representation of the original 20-dimensional data. This compressed data can be used for tasks such as dimensionality reduction, feature extraction, or as input for other machine learning algorithms.
- Output (Encoded Data): The printed output (encoded_data) will be the latent space representation (5 dimensions) of the input data. This represents a compressed version of the input data, retaining the most critical information for reconstruction.
- Reconstruction Process: The decoder attempts to recreate the original 20-dimensional input from the 5-dimensional latent space, aiming for minimal reconstruction error.

Isomap (Isometric Mapping)

- Isomap (Isometric Mapping) is a non-linear dimensionality reduction technique that seeks to embed high-dimensional data into a lower-dimensional space while preserving the intrinsic geometric structure of the data.
- It is particularly valuable when the data lies on a complex or curved manifold in high-dimensional space, which traditional linear techniques like PCA cannot capture effectively.
- Isomap is part of the manifold learning family and extends Multidimensional Scaling (MDS) by incorporating geodesic distances along the manifold, rather than relying solely on Euclidean distances.

Why Isomap?

- Real-world data often exhibits non-linear relationships that cannot be effectively modeled by linear techniques. For instance:
    - In image processing, face images taken from different angles might be described by a complex manifold.
    - In speech recognition, phonetic changes in speech data form a non-linear pattern.
- When such data is embedded into a lower-dimensional space using linear techniques like PCA, important structural information is often lost. Isomap helps in such cases by capturing the non-linear structure of the data, preserving the manifold's intrinsic geometry.



Key Concepts of Isomap

- Manifold Learning:
    - Isomap assumes that high-dimensional data points lie on a low-dimensional, non-linear manifold that can be embedded into a lower-dimensional space.
    - The goal of Isomap is to find this low-dimensional representation while preserving the distances between points as measured along the manifold, rather than in the original high-dimensional Euclidean space.
- Geodesic Distance:
    - Isomap calculates the geodesic distance between data points instead of the Euclidean distance.

  - The geodesic distance is the shortest path along the manifold that connects two points, which is approximated using a graph-based approach.
  - This helps capture the true structure of the manifold, especially for datasets that are non-linearly distributed in high-dimensional space.

- Graph Construction:
  - The first step in Isomap is to construct a neighborhood graph.
  - Each point is connected to its nearest neighbors (using k-nearest neighbors or a fixed radius) based on Euclidean distances. This graph represents the local relationships between points.
- Geodesic Distance Calculation:
  - After constructing the neighborhood graph, Isomap uses a shortest-path algorithm (like Dijkstra's or Floyd-Warshall) to compute the geodesic distances between all pairs of points, taking into account the structure of the graph.
- Multidimensional Scaling (MDS):
  - Once the geodesic distance matrix is computed, Isomap applies classical MDS to the geodesic distance matrix.
  - MDS finds a low-dimensional representation of the data by preserving the pairwise distances as much as possible in the new space.
  - The result is a low-dimensional embedding of the data points that respects the manifold structure.

Steps of Isomap

1. Data Preparation
   - Before applying Isomap, ensure your data is properly preprocessed. This includes:
     - Normalization: Scaling your data (if necessary) to have a mean of zero and a standard deviation of one can improve the performance of distance calculations.
     - Handling Missing Values: Address any missing values in your dataset, as they can affect distance computations.
2. Constructing the Neighborhood Graph

- Objective: Create a graph where each data point is connected to its nearest neighbors.
  - Select the Neighborhood Size:
    - Choose a value for k (the number of nearest neighbors) or a distance threshold ϵ to define the connections in the graph.

    - If using k-nearest neighbors (k-NN), for each point in the dataset, connect it to its k closest points based on the Euclidean distance.
    - Alternatively, if using a distance threshold ϵ, connect a point to all other points within that distance.

3. Graph Representation:

- Each data point is a node in the graph.
- Edges between nodes represent the Euclidean distances between connected points.
- For example, if you have points $x_1, x_2, \ldots, x_n$, the graph can be represented as an adjacency matrix A, where:

$$A_{ij} = \begin{cases} \text{dist}(x_i, x_j) & \text{if } x_j \text{ is a neighbor of } x_i \\ \infty & \text{otherwise} \end{cases}$$

3. Compute Geodesic Distances

- Objective: Determine the shortest paths (geodesic distances) between all pairs of points in the neighborhood graph.
  - Shortest Path Calculation:
    - Use a graph algorithm (like Dijkstra's algorithm or the Floyd-Warshall algorithm) to compute the shortest path distances between every pair of nodes in the graph.
    - The result is a distance matrix D, where D_{ij} is the geodesic distance between points x_i and x_j.
- Example: If your graph connects $x_1$to x_2 and x_2 to x_3, but not directly x_1 to x_3, the geodesic distance from x_1 to x_3 will be the sum of the distances from x_1 to x_2 and from x_2 to x_3.

4. Multidimensional Scaling (MDS)

- Objective: Reduce the dimensionality of the data based on the geodesic distance matrix.
  - Applying MDS:
    - MDS aims to find a configuration of points in a lower-dimensional space (e.g., 2D or 3D) such that the pairwise distances between these points match the geodesic distances as closely as possible.
    - Use techniques such as classical MDS or other MDS variants to obtain the low-dimensional representation.
  - Eigen Decomposition:
    - Center the distance matrix D using the double centering method to obtain a new matrix D_c:

$D_c = -(1/2)HDH$

- where H is the centering matrix defined as:

  $H = I - (1/n)\, 11^T$

  Here, I is the identity matrix, and 1 is a column vector of ones.

- Extracting Components:
  - Perform eigen decomposition on the centered distance matrix D_c to find the eigenvalues and eigenvectors.
  - Select the top d (desired dimensions) eigenvectors corresponding to the largest eigenvalues to construct the low-dimensional representation.

5. Embedding the Data

- Objective: Map the original high-dimensional data points to the lower-dimensional space.
  - Final Representation:
    - The final low-dimensional coordinates are obtained by projecting the data points using the selected eigenvectors.

$$Y = E_d\sqrt{\Lambda_d}$$

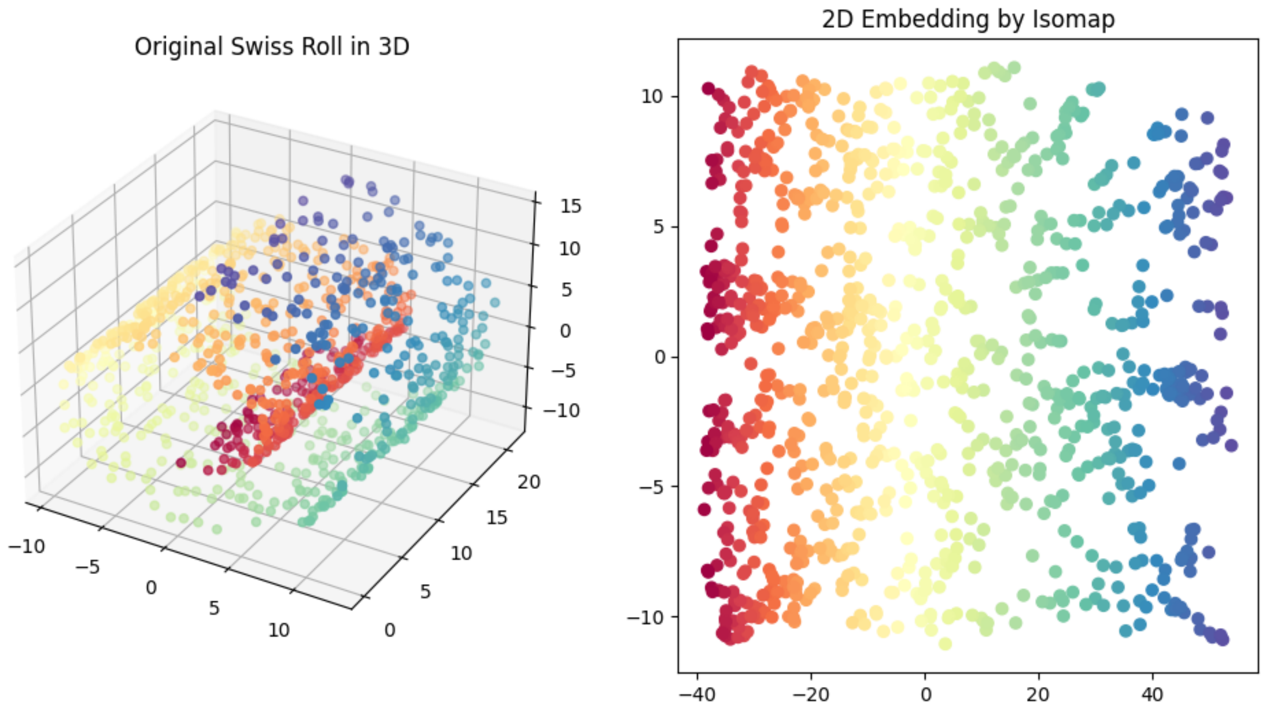
where E_d consists of the top d eigenvectors and Λ_d is a diagonal matrix containing the corresponding eigenvalues.

6. Visualization and Interpretation

- Objective: Visualize the reduced data and analyze the results.
  - Plotting:
    - If the reduced space is 2D or 3D, plot the resulting points to visualize the intrinsic structure of the data.
    - Each point's position in the low-dimensional space reflects its relationships with other points based on geodesic distances.
  - Interpretation:
    - Analyze the resulting plot to identify clusters, patterns, or trends in the data that may not have been apparent in the original high-dimensional representation.

Below is an example implementation of Isomap in Python using sklearn, demonstrating the steps above:

```
import numpy as np
from sklearn.datasets import make_swiss_roll
from sklearn.manifold import Isomap
import matplotlib.pyplot as plt

# Generate Swiss Roll dataset
X, color = make_swiss_roll(n_samples=1000, noise=0.1)

# Step 1: Initialize Isomap
n_neighbors = 10  # Set the number of nearest neighbors
n_components = 2  # Set the desired number of dimensions for embedding

# Step 2: Apply Isomap
isomap = Isomap(n_neighbors=n_neighbors, n_components=n_components)
X_isomap = isomap.fit_transform(X)

# Step 3: Visualization
# Original Swiss Roll dataset in 3D
fig = plt.figure(figsize=(12, 6))
ax = fig.add_subplot(121, projection='3d')
ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=color, cmap=plt.cm.Spectral)
ax.set_title('Original Swiss Roll in 3D')

# 2D embedding from Isomap
plt.subplot(122)
plt.scatter(X_isomap[:, 0], X_isomap[:, 1], c=color, cmap=plt.cm.Spectral)
plt.title('2D Embedding by Isomap')
plt.show()
```



Use Case: Image Classification for Handwritten Digit Recognition

Objective

- The primary objective of this use case is to enhance the process of image classification by utilizing dimensionality reduction techniques on the MNIST dataset, which consists of handwritten digit images.
- Handwritten digit recognition is a fundamental problem in computer vision and pattern recognition, offering valuable insights into the effectiveness of machine learning models in interpreting visual data.
- By applying various feature selection methods, such as Filter, Wrapper, and Embedded techniques, we aim to identify and retain the most informative features while discarding redundant or irrelevant ones.
- Furthermore, we will implement feature extraction methods, including Principal Component Analysis (PCA), Kernel PCA, Linear Discriminant Analysis (LDA), t-Distributed

Stochastic Neighbor Embedding (t-SNE), and Autoencoders, to create a more compact representation of the data that retains essential information for classification.
- Ultimately, this use case seeks to build an efficient classification model that accurately identifies handwritten digits while demonstrating the practical applications of these dimensionality reduction techniques in a real-world context.

Dataset Overview

- The MNIST dataset is a widely recognized benchmark in the field of machine learning and computer vision, consisting of 70,000 grayscale images of handwritten digits. Each image is represented as a 28x28 pixel matrix, resulting in a total of 784 pixels per image.
- The dataset includes:
  - Training Set: 60,000 images used to train the classification model.
  - Test Set: 10,000 images used to evaluate the model's performance.
- Each image is associated with a corresponding label indicating the digit it represents (0-9). The dataset is known for its simplicity and is often used for benchmarking image classification algorithms.
- The MNIST dataset provides a rich ground for understanding fundamental concepts in image processing, machine learning, and dimensionality reduction techniques, making it an ideal choice for professionals seeking to enhance their skills in these areas.
- The images in MNIST are centered and normalized, making it easier to work with and ideal for training various classification algorithms.

Data Loading and Exploration

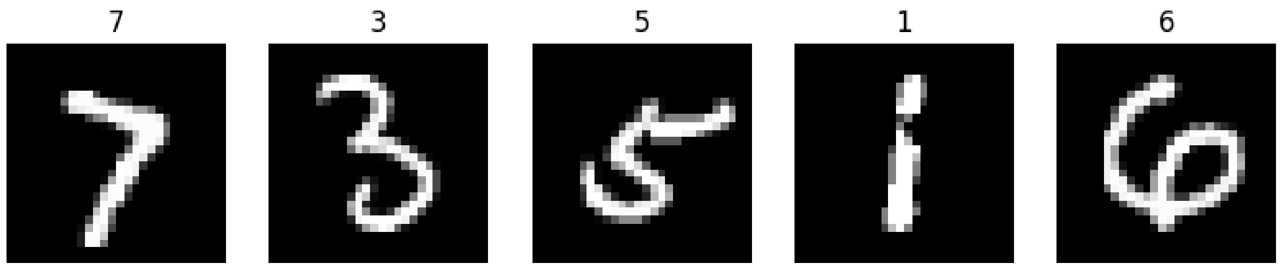
- **First, we will load the MNIST dataset and explore its characteristics.**
- **In this use case we are taking 800 traning data and 200 testing data**

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split

# Load the MNIST dataset
mnist = fetch_openml('mnist_784', version=1)
X, y = mnist.data, mnist.target

# Convert the labels to integers
y = y.astype(int)

# Sample 1,000 images
X_1000, _, y_1000, _ = train_test_split(X, y, train_size=1000, stratify=y, random

# Split into training (800) and testing (200) sets
X_train, X_test, y_train, y_test = train_test_split(X_1000, y_1000, train_size=0.

# Display the shape of the dataset
print(f"Shape of the training set: {X_train.shape}")
print(f"Shape of the testing set: {X_test.shape}")
print(f"Number of classes: {len(np.unique(y))}")
```

```
Shape of the training set: (800, 784)
Shape of the testing set: (200, 784)
Number of classes: 10
```

Creating DataFrame and Plotting sample images

```
def plot_sample_images(images, labels, n=5):
    plt.figure(figsize=(10, 2))
    for i in range(n):
        plt.subplot(1, n, i + 1)
        plt.imshow(images[i].reshape(28, 28), cmap='gray')
        plt.title(labels[i])
        plt.axis('off')
    plt.show()

# Plot sample images
plot_sample_images(X_train.to_numpy(), y_train.to_numpy(), n=5)
```



Feature Selection

For image data, feature selection often involves dimensionality reduction. We will apply various feature selection techniques to enhance our dataset.

Filter Method: Variance Threshold

This method selects features based on their variance. Features with low variance are less likely to be useful for classification.

```
from sklearn.feature_selection import VarianceThreshold

# Apply variance threshold to the MNIST dataset
var_thresh = VarianceThreshold(threshold=0.01)
X_train_var = var_thresh.fit_transform(X_train)  # Apply to training data
X_test_var = var_thresh.transform(X_test)        # Apply to test data

# Display the shape after applying variance threshold
print(f"Shape of training set after variance threshold: {X_train_var.shape}")
print(f"Shape of test set after variance threshold: {X_test_var.shape}")
```

```
Shape of training set after variance threshold: (800, 599)
Shape of test set after variance threshold: (200, 599)
```

- Originally, the MNIST dataset has 784 features (corresponding to the pixel values of 28x28 images). By applying the variance threshold, you have effectively reduced the number of features from 784 to 599, which indicates that 185 features had low variance and were removed from the dataset.
- Features with low variance contribute little to the model's predictive power. By removing these features, you can simplify your model and potentially improve its performance by reducing noise.

Wrapper Method: Recursive Feature Elimination (RFE)

Wrapper methods evaluate feature subsets by training and testing a model on them. Common algorithms include Recursive Feature Elimination (RFE).

```
from sklearn.svm import SVC
from sklearn.feature_selection import RFE

# Use a SVC as the estimator for RFE
svc = SVC(kernel="linear")

# RFE to select 100 features from the training data after variance threshold
rfe = RFE(estimator=svc, n_features_to_select=100)  # Keep 100 features
X_train_rfe = rfe.fit_transform(X_train_var, y_train)
X_test_rfe = rfe.transform(X_test_var)

# Display the shape after applying RFE
print(f"Shape of training set after RFE: {X_train_rfe.shape}")
print(f"Shape of test set after RFE: {X_test_rfe.shape}")
```

```
Shape of training set after RFE: (800, 100)
Shape of test set after RFE: (200, 100)
```

- After running this code, you can get the output, indicating that both datasets have been reduced to 100 features.
- Using RFE with a linear SVC is a robust method for feature selection, as it iteratively removes the least important features based on the model's performance.
- By retaining only the 100 most significant features, you are likely to improve the performance of subsequent models while reducing the risk of overfitting and improving computational efficiency.
- This approach can also enhance the interpretability of your model by highlighting the features that contribute most to the predictions.

Embedded Method: Feature Importance from Random Forest

```
from sklearn.ensemble import RandomForestClassifier

# Fit Random Forest model on the training set after applying variance threshold
rf = RandomForestClassifier(random_state=42)  # Set random_state for reproducibil
rf.fit(X_train_var, y_train)

# Get feature importances
importances = rf.feature_importances_
indices = np.argsort(importances)[::-1]

# Select top 100 features based on importance
X_train_embedded = X_train_var[:, indices[:100]]
X_test_embedded = X_test_var[:, indices[:100]]

# Display the shape after embedded feature selection
print(f"Shape of training set after embedded feature selection: {X_train_embedded
print(f"Shape of test set after embedded feature selection: {X_test_embedded.shap
```

```
Shape of training set after embedded feature selection: (800, 100)
Shape of test set after embedded feature selection: (200, 100)
```

- This approach effectively reduces the dimensionality of your datasets by retaining only the most important features according to the Random Forest model.
- By focusing on the top 100 features, you may improve the performance of subsequent models by reducing noise and computational complexity.
- Additionally, feature importance from the Random Forest can provide insights into which features are most influential for the predictions, aiding in model interpretability.

Feature Extraction

Next, we will apply dimensionality reduction techniques to extract features from the image data.

Principal Component Analysis (PCA)

```
from sklearn.decomposition import PCA

# Applying PCA to reduce dimensions to 50 components
pca = PCA(n_components=50)
X_train_pca = pca.fit_transform(X_train_embedded)  # Apply PCA to training set
X_test_pca = pca.transform(X_test_embedded)        # Apply PCA to test set

# Display the shape after applying PCA
print(f"Shape of training set after PCA: {X_train_pca.shape}")
print(f"Shape of test set after PCA: {X_test_pca.shape}")
```

```
Shape of training set after PCA: (800, 50)
Shape of test set after PCA: (200, 50)
```

- The code applies PCA to reduce the dimensionality of the embedded training and test datasets to 50 components. This transformation is particularly useful for:
  - Reducing Complexity: Making the data easier to visualize and analyze.
  - Improving Performance: Reducing the risk of overfitting in machine learning models by eliminating less informative features.
  - Maintaining Variance: Capturing the most significant patterns in the data while discarding noise.

Kernel Principal Component Analysis (Kernel PCA)

```
from sklearn.decomposition import KernelPCA

# Applying Kernel PCA to reduce dimensions to 50 components using RBF kernel
kpca = KernelPCA(n_components=50, kernel='rbf')
X_train_kpca = kpca.fit_transform(X_train_embedded)  # Apply Kernel PCA to traini
X_test_kpca = kpca.transform(X_test_embedded)        # Apply Kernel PCA to test s

# Display the shape after applying Kernel PCA
print(f"Shape of training set after Kernel PCA: {X_train_kpca.shape}")
print(f"Shape of test set after Kernel PCA: {X_test_kpca.shape}")
```

```
Shape of training set after Kernel PCA: (800, 42)
Shape of test set after Kernel PCA: (200, 42)
```

- The code applies Kernel PCA to the embedded training and test datasets to reduce their dimensionality to 42 components using the RBF kernel. The key features of this process include:
  - Non-Linear Dimensionality Reduction: Unlike standard PCA, Kernel PCA can capture non-linear relationships in the data, which is particularly useful for complex datasets like images.
  - Flexibility: By using a kernel function (in this case, RBF), Kernel PCA can project the data into a higher-dimensional feature space where linear separability may be achieved.
  - Reducing Complexity: Similar to standard PCA, it simplifies the dataset, making it more manageable for modeling and analysis.

Linear Discriminant Analysis (LDA)

LDA can be used to maximize the separation between classes.

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# Apply LDA to reduce dimensionality to 9 components (10 classes - 1)
lda = LinearDiscriminantAnalysis(n_components=9)  # 10 classes in MNIST (digits 0
X_train_lda = lda.fit_transform(X_train_embedded, y_train)  # Fit and transform t
X_test_lda = lda.transform(X_test_embedded)                 # Transform the test

# Display the shape after applying LDA
print(f"Shape of training set after LDA: {X_train_lda.shape}")
print(f"Shape of test set after LDA: {X_test_lda.shape}")
```

```
Shape of training set after LDA: (800, 9)
Shape of test set after LDA: (200, 9)
```

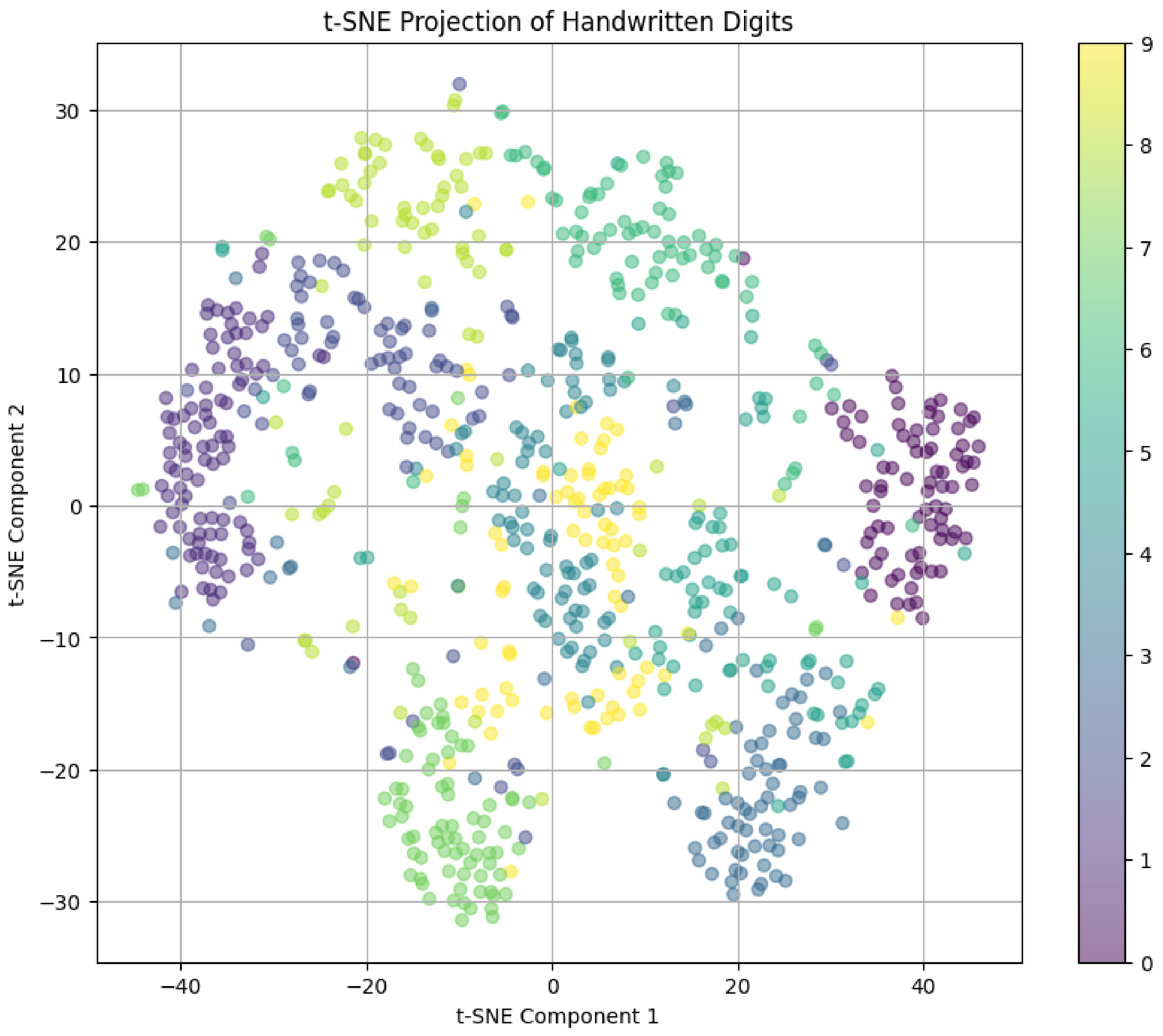
- LDA focuses on class separability, which is beneficial for a classification task, but the small sample size may limit its effectiveness.

- After running this code, you will have two new datasets: X_train_lda and X_test_lda, both of which have reduced dimensionality (9 features), allowing for potentially more efficient and effective machine learning modeling.

t-Distributed Stochastic Neighbor Embedding (t-SNE)

t-SNE is great for visualizing high-dimensional data in two or three dimensions.

```
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt

# Applying t-SNE to reduce to 2 dimensions
tsne = TSNE(n_components=2, random_state=42)
X_train_tsne = tsne.fit_transform(X_train_embedded)  # Apply t-SNE to the embedde

# Visualizing t-SNE results
plt.figure(figsize=(10, 8))  # Optional: set the figure size for better visibilit
plt.scatter(X_train_tsne[:, 0], X_train_tsne[:, 1], c=y_train, cmap='viridis', al
plt.title('t-SNE Projection of Handwritten Digits')
plt.colorbar()
plt.xlabel('t-SNE Component 1')  # Label for x-axis
plt.ylabel('t-SNE Component 2')  # Label for y-axis
plt.grid(True)  # Optional: add grid for better readability
plt.show()
```



- After running this code, you will get a scatter plot that visualizes the t-SNE projection of the embedded training dataset.
- The plot will show how different handwritten digits are represented in a two-dimensional space, with distinct clusters indicating different classes.
- This visualization aids in assessing the performance of the previous dimensionality reduction techniques (like PCA, LDA, etc.) and understanding the underlying structure of the data.

Autoencoders

Autoencoders can be used to reduce dimensionality while preserving significant features.

```
from keras.layers import Input, Dense
from keras.models import Model

# Define the autoencoder model using the shape of the embedded training set
input_layer = Input(shape=(X_train_embedded.shape[1],))  # Use the number of feat
encoded = Dense(32, activation='relu')(input_layer)  # Compression layer
decoded = Dense(X_train_embedded.shape[1], activation='sigmoid')(encoded)  # Reco

# Create the autoencoder model
autoencoder = Model(input_layer, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')  # Compile the

# Train the autoencoder on the embedded data
autoencoder.fit(X_train_embedded, X_train_embedded, epochs=50, batch_size=256, sh

# Use the encoder part of the autoencoder to reduce dimensionality
encoder = Model(input_layer, encoded)  # Create a model to output the encoded rep
X_train_autoencoded = encoder.predict(X_train_embedded)  # Encode the training se
X_test_autoencoded = encoder.predict(X_test_embedded)      # Encode the test set

# Display the shape after autoencoding
print(f"Shape of training set after Autoencoder: {X_train_autoencoded.shape}")
print(f"Shape of test set after Autoencoder: {X_test_autoencoded.shape}")
```

```
Epoch 1/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - loss: -1224.1215
Epoch 2/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: -3229.3276
Epoch 3/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: -5366.7769
Epoch 4/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: -7890.4526
Epoch 5/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: -11215.2256
Epoch 6/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: -15032.3301
Epoch 7/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: -20068.1367
```