Latent Variables and Dimensionality Reduction Techniques in Python

Kiara Gholizad[1]

[1]Department of Physics, Sharif University of Technology, Tehran, Iran

March 15, 2025

Contents

1 Introduction

Latent variables are variables that are not directly observed but are inferred from observed data. In many datasets, high-dimensional observations (with dozens or hundreds of features) are driven by a smaller number of underlying latent factors. **Dimensionality reduction** techniques aim to discover these lower-dimensional representations (latent variables) that capture the essential structure in the data while discarding noise or redundant information. Reducing dimensionality can help in data visualization, speed up machine learning algorithms, alleviate the "curse of dimensionality," and reveal meaningful patterns. Below we explore several common dimensionality reduction methods, their concepts, applications, and how to use them in Python. We cover both **linear** methods (like PCA, LDA, NMF, ICA) and **nonlinear** methods (like t-SNE, UMAP, Autoencoders), as well as **unsupervised** vs **supervised** approaches.

2 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is an unsupervised linear technique that finds a new set of orthogonal axes (principal components) that successively maximize the variance of the data. In other words, PCA identifies the directions in feature space along which the data varies the most, and it projects the data onto those directions. The first principal component captures the largest possible variance, the second (orthogonal to the first) captures the next largest variance, and so on. By keeping only the top k components, we obtain a k-dimensional representation that retains most of the original variance. These principal components can be considered latent variables that summarize the original features.

Applications:

- **Feature Extraction & Compression:** Reduce dataset dimensionality while preserving most information (e.g., compress image or signal data).
- **Noise Reduction:** By dropping low-variance components, we remove noise and redundant features.
- **Data Visualization:** Plot high-dimensional data in 2D or 3D using the top components to reveal clustering or patterns (e.g., visualize 4D Iris data in 2D).

- **Preprocessing for ML:** Decorrelate features and speed up training (many algorithms work better on PCA-transformed data).

Python Example (PCA on the Iris dataset)

Below we use `sklearn.decomposition.PCA` to reduce the 4-dimensional Iris flower dataset to 2 dimensions. We then examine the explained variance ratio of the components (how much of the original variance each principal component accounts for).

```python
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA

# Load iris dataset (4 features: sepal length, sepal width, petal
    length, petal width)
iris = load_iris()
X = iris.data  # shape (150, 4)
y = iris.target

# Apply PCA to reduce to 2 dimensions
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

print("Explained variance ratio:", pca.explained_variance_ratio_)
```

Output:

```
Explained variance ratio: [0.92461872 0.05306648]
```

The two-component PCA in this case captures about 92.46% and 5.3% of the variance respectively (together ∼97.8% of total variance). This means we can represent the 4D iris data in 2D with minimal information loss. To visualize the result, we can plot the data points by their two principal components and color them by species:

PCA is unsupervised, meaning it ignores class labels. In the Iris example, PCA wasn't told about species, yet the first principal component happens to separate setosa well because that direction has the most overall variance. PCA is widely used as a general-purpose dimensionality reduction technique for many data types (images, gene expression, economic data, etc.) and is often a first step in exploratory data analysis.

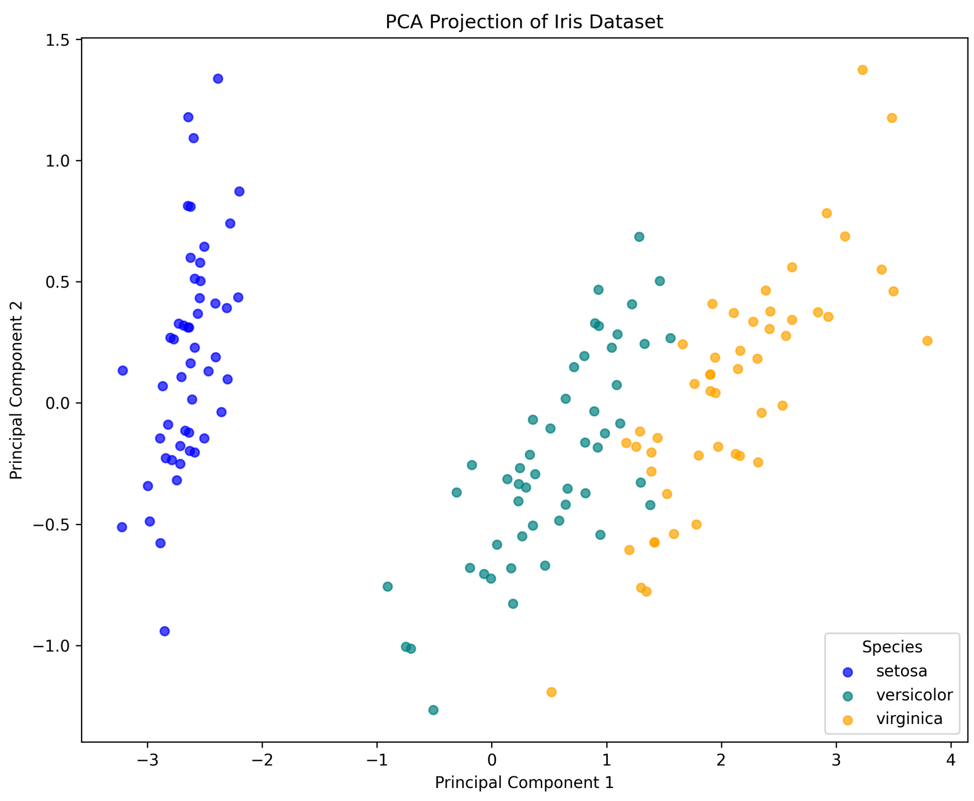

Figure 1: PCA projection of the Iris dataset onto its first two principal components. Each point is an iris flower, colored by species (setosa = blue, versicolor = teal, virginica = orange). The PCA has separated the setosa species (blue cluster on left) from the other two. Versicolor and virginica overlap more, but still show some separation along the second component.

3 t-Distributed Stochastic Neighbor Embedding (t-SNE)

t-SNE (t-Distributed Stochastic Neighbor Embedding) is a non-linear dimensionality reduction technique, primarily used for visualization of high-dimensional datasets in 2 or 3 dimensions. Simply put, t-SNE maps high-dimensional points to a lower-dimensional space in a way that preserves local structure: points that are close in the original space stay close together in the embedding, and distant points stay far apart. It does this by converting distances between points into probabilistic similarities and then finding a lower-dimensional layout that minimizes the divergence (Kullback–Leibler divergence) between those similarity distributions. The result is often a clustering of the data where similar instances form tight groups in the 2D/3D plot.

Applications:

- **Visualizing Complex Data:** t-SNE is popular for visualizing image datasets (e.g., MNIST digits), word vectors, genomics data, etc., to reveal clusters or grouping patterns that correspond to categories or other structure.
- **Non-linear Manifold Learning:** Useful when data lies on a complicated manifold; t-SNE can unravel some of that structure where linear methods like PCA fail.
- **Insight into Clusters:** Analysts often use t-SNE plots to identify potential natural groupings in high-dimensional data (e.g., customer segments, cell types in single-cell RNA-seq, etc.).

Python Example (t-SNE on MNIST digits):

We'll use `sklearn.manifold.TSNE` to reduce a dataset of handwritten digits to 2D. (For brevity, suppose we have features `X` of shape `(n_samples, n_features)` and labels `y`.)

```python
from sklearn.manifold import TSNE
from sklearn.datasets import load_digits

digits = load_digits()  # 8x8 images of handwritten digits (0-9),
 64 features
X = digits.data
y = digits.target

tsne = TSNE(n_components=2, random_state=42, perplexity=30)
X_tsne = tsne.fit_transform(X)

print(X_tsne.shape)
```

Output:

```
(1797, 2)
```

The result `X_tsne` contains 2D coordinates for each digit image. We can plot these points, using different colors or markers for each digit label:

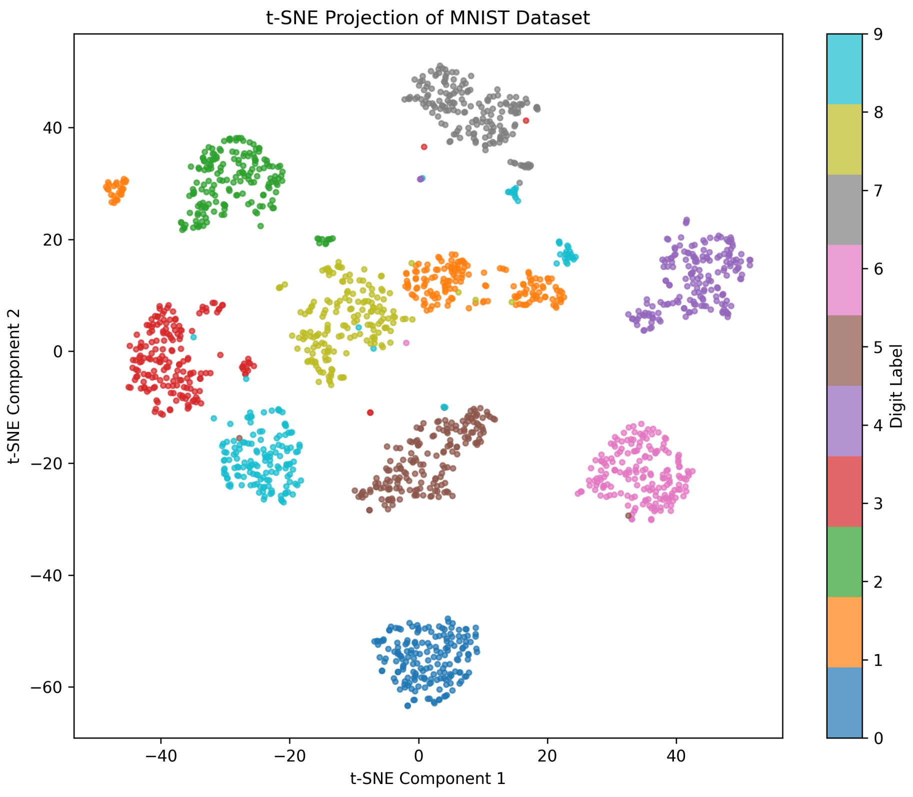

Figure 2: t-SNE projection of the MNIST dataset onto its first two components. Each point represents a digit image, colored by its label.

t-SNE embedding of 500 handwritten digits (0–9) into 2D. Each point represents an image, colored by the digit it represents. Notice that t-SNE has formed clear clusters for most digits (e.g., blue points for digit "0" cluster together, brown points for "5" cluster together, etc.), indicating it has separated the classes based on the images' pixel patterns.

t-SNE is great for visualization, but note that it is computationally intensive for large datasets and the embedding can vary between runs (the cost function is non-convex, so different random initializations may give slightly different layouts. It also doesn't preserve global distances well (clusters may be arbitrarily placed far apart or close together). Thus, t-SNE is mainly used as an exploratory tool. It's not typically used to generate features for another algorithm (since the absolute distances in t-SNE space are not meaningful beyond local neighborhood structure). Hyperparameters like perplexity can significantly affect the output, so some experimentation is often needed.

4 Autoencoders (Neural Network for Dimensionality Reduction)

An **autoencoder** is a type of neural network designed to learn an efficient encoding of input data. It consists of an **encoder** function that compresses the input into a low-dimensional latent space (bottleneck), and a **decoder** that reconstructs the original data from this compressed representation. Autoencoders are trained in an unsupervised manner: the training objective is to minimize the reconstruction error between the input and its reconstruction. By forcing the network to go through a small hidden layer, the autoencoder is compelled to learn the most salient features of the data – essentially the latent variables that capture the data's structure. Unlike PCA which is linear, autoencoders can learn **non-linear** encodings (through non-linear activation functions or deeper architectures), enabling them to capture complex patterns.

Applications:

- **Non-linear Dimensionality Reduction:** Use the encoder part to reduce data to a latent vector (which can be used as features for other tasks).
- **Data Denoising:** Denoising autoencoders learn to reconstruct clean data from noisy input, thus isolating essential structure.
- **Anomaly Detection:** If an autoencoder is trained on "normal" data, it may reconstruct familiar patterns well but fail on anomalies, so reconstruction error can flag outliers.
- **Generative Modeling:** Variational Autoencoders (VAEs) are a variant that learn latent spaces suitable for generating new data (images, etc.).

Python Example (Autoencoder for Dimensionality Reduction)

We'll build a simple autoencoder using Keras to compress the 64-dimensional digits data down to 2 dimensions.

```python
from tensorflow.keras import layers, models
from sklearn.datasets import load_digits

digits = load_digits()  # 8x8 images of handwritten digits (0-9), 64 features
X = digits.data

# Define dimensions
input_dim = X.shape[1]
encoding_dim = 2    # size of latent space

# Encoder network
input_img = layers.Input(shape=(input_dim,))
encoded = layers.Dense(encoding_dim, activation='relu')(input_img)

# Decoder network
decoded = layers.Dense(input_dim, activation='sigmoid')(encoded)

# Autoencoder model maps input->reconstruction
autoencoder = models.Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='mse')

# Train the autoencoder (use X as both input and target)
autoencoder.fit(X, X, epochs=50, batch_size=256, shuffle=True, verbose=0)

# After training, get encoded 2D representation
encoder = models.Model(input_img, encoded)
X_encoded = encoder.predict(X)
print(X_encoded[:5])
```

Output:

```
[[40.28191  43.1432 ]
[42.25422  36.343777]
[43.712975 39.806633]
[29.008682 34.154594]
[31.0693   30.586098]]
```

After training, `X_encoded` contains the 2D latent variables for each data point. We can visualize these just like we did for PCA or t-SNE. Autoencoders often achieve qualitatively similar results to other non-linear methods (e.g., clustering digits in latent space), though the latent space might not be as cleanly separated unless the network capacity and training are well-tuned. The strength of autoencoders is their ability to reconstruct data and potentially learn more complex features. For example, below is a visualization of an autoencoder's reconstruction ability:

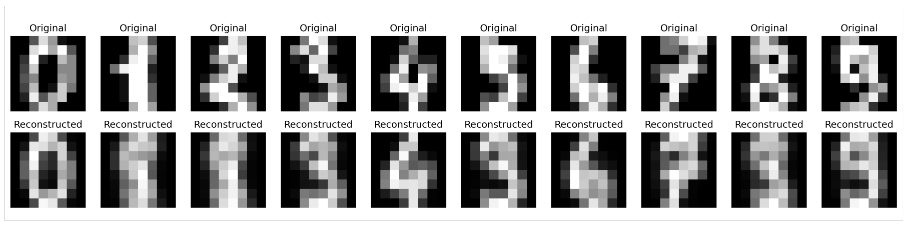

Figure 3: Example of an autoencoder's output on the MNIST dataset. The top row shows original digit images, and the bottom row shows the autoencoder's reconstructions. The autoencoder has learned to capture the essential features of each digit well enough that the reconstructed images are recognizable, despite the data being compressed into a lower-dimensional latent space.

In practice, autoencoders can be made much more powerful with deep convolutional layers for image data, or recurrent layers for sequence data. However, they require more data and computational effort to train compared to simpler techniques like PCA. Autoencoders don't inherently provide *interpretable* components (each dimension of the latent space is a learned combination of features), but they can model non-linear relationships. Techniques like **Variational Autoencoders (VAE)** impose structure on the latent space (e.g., roughly Gaussian distributed) to make it more continuous and interpretable, at the cost of a more complex training process.

5 Linear Discriminant Analysis (LDA)

Linear Discriminant Analysis (LDA) in the context of dimensionality reduction is a supervised linear method that finds axes which maximize class separability. (This is not to be confused with *Latent Dirichlet Allocation*, which is a topic modeling technique – here we mean Fisher's LDA for dimensionality reduction.) LDA searches for a linear combination of features that best separates two or more classes. It does so by maximizing the ratio of *between-class variance* to *within-class variance* in the projected space. In essence, LDA produces latent variables (discriminant components) that emphasize class differences in the data. If you have C classes, LDA can yield at most $C - 1$ meaningful components. These components can be used to visualize class clusters or as features for classification.

Applications:

- **Feature Reduction for Classification:** Project data into a lower-dimensional space that emphasizes class separation, then run a classifier in that space. (LDA itself can be used as a classifier too, assuming Gaussian class distributions.)
- **Visualization of Labeled Data:** Plot high-dimensional labeled data in 2D by LDA to see how well classes separate (often used in pattern recognition, e.g., visualize how different species, document categories, or customer segments separate).
- **Face Recognition (Fisherfaces):** In image recognition, LDA has been used on face image data to find components that best distinguish individuals, as a complement to PCA "eigenfaces".

Python Example (LDA on Iris dataset)

We use `sklearn.discriminant_analysis.LinearDiscriminantAnalysis` to reduce the Iris data (3 classes) to 2 dimensions (since for 3 classes, $C - 1 = 2$).

```python
from sklearn.discriminant_analysis import
 LinearDiscriminantAnalysis
from sklearn.datasets import load_iris

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

lda = LinearDiscriminantAnalysis(n_components=2)
X_lda = lda.fit_transform(X, y)

print(X_lda.shape)    # (150, 2)
print(lda.explained_variance_ratio_)   # proportion of class-
 separability variance
```

Output:

```
(150, 2)
[0.9912126 0.0087874]
```

If we plot `X_lda` for the Iris dataset, we get a projection that is explicitly optimized for class separation:

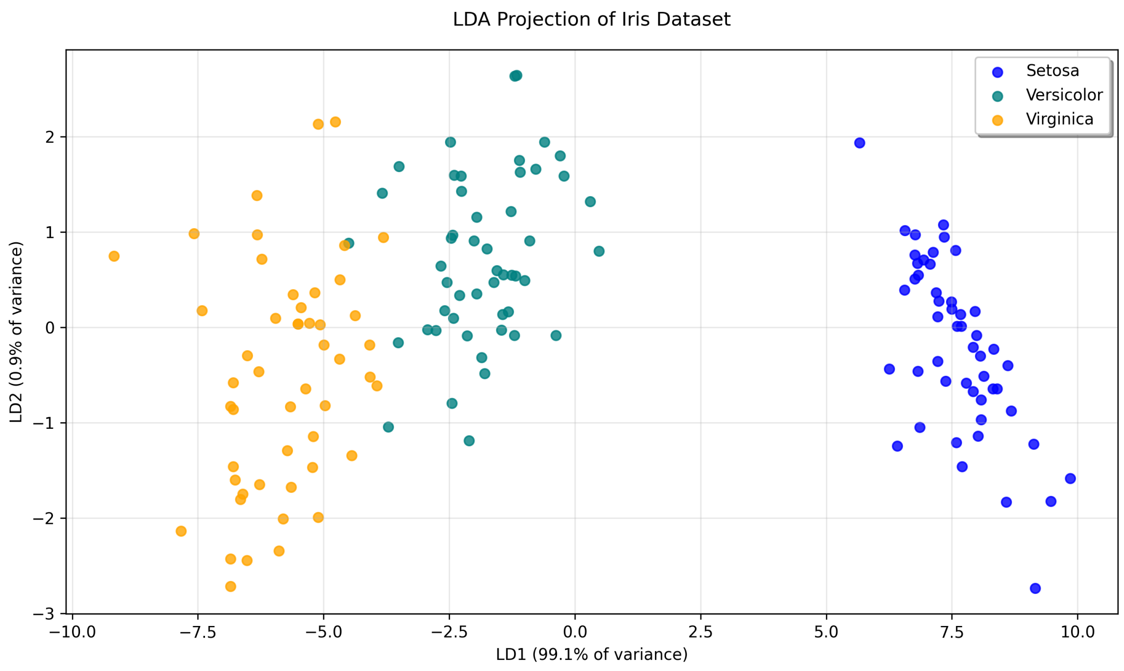

Figure 4: LDA projection of the Iris dataset onto 2 linear discriminant components. Each point is an iris flower, colored by species. Compared to PCA's unsupervised projection, LDA (supervised) achieves a near-perfect separation of the three species in this 2D space. Setosa (blue) is well-separated on the first LD axis, while the versicolor (teal) and virginica (orange) clusters are separated along the second axis.

Because LDA used the class labels, it found directions that discriminate the species (in this case, setosa vs. non-setosa, and versicolor vs. virginica). LDA is commonly used in classification tasks when we want to reduce feature dimension while preserving class information. It assumes normal distribution of features within each class and equal class covariances; when those assumptions hold, LDA provides optimal dimensionality reduction for class separation. Even if the assumptions are not perfectly met, LDA often yields useful low-dimensional views of labeled data.

6 Non-Negative Matrix Factorization (NMF)

Non-Negative Matrix Factorization (NMF) is an unsupervised linear technique that factorizes a data matrix into two lower-dimensional matrices, under the constraint that all values are non-negative. Typically, given a data matrix X of shape $(n\_samples, n\_features)$ with non-negative entries, NMF finds an approximation $X \approx W \cdot H$, where W is $(n\_samples, k)$ and H is $(k, n\_features)$, with $k \ll n\_features$. The columns of H can be thought of as **basis vectors** (latent components) and the rows of W as coefficients or *activations* of those components for each sample. The non-negativity constraint often leads to a **parts-based representation**. Since you can only add components, not subtract, NMF tends to learn components that correspond to interpretable parts of the data.

For example, when NMF is applied to images of faces, the basis components in H often resemble parts of faces (like eyes, nose, mouth) rather than holistic faces. Each face image (row of X) is then represented as an additive combination of these parts. This is in contrast to PCA, where components can be mixtures of positive and negative values and typically look like global eigen-features that are harder to interpret.

Applications:

- **Topic Extraction in Text:** NMF on a document-term matrix can uncover topics. Each component (latent factor) corresponds to a topic, with large weights for words related to that topic, and each document is a combination of topics.
- **Audio Source Separation:** If we have magnitude spectrograms of mixed sounds (all non-negative), NMF can separate sources (e.g., separate different instruments or speakers) by learning parts of spectrograms.
- **Recommender Systems:** NMF on a user-item rating matrix can find latent factors that describe user preferences and item attributes, with only positive influence factors.
- **Image Processing:** Decompose images into parts (e.g., parts of faces, objects) which can be useful for object recognition or compression.

Python Example (NMF on a toy dataset)

We'll use `sklearn.decomposition.NMF` to factorize the Iris data matrix (since all features are positive lengths) into, say, 2 components.

```python
from sklearn.decomposition import NMF
from sklearn.datasets import load_iris

iris = load_iris()
X = iris.data

nmf = NMF(n_components=2, random_state=42)
W = nmf.fit_transform(X)    # shape (150, 2)
H = nmf.components_    # shape (2, 4)

print(W[:5])    # first 5 samples in 2D latent space
print(H[:, :4])    # the 2 basis vectors (for 4 features)
```

Output:

```
[[0.41356229 0.10457618]
[0.36548759 0.14091832]
[0.37785481 0.10179942]
[0.35001275 0.14891411]
[0.41596217 0.09520372]]
[[11.05547983  7.96828449  2.22911407  0.07051629]
[ 5.0075179   2.01368146  4.58653481  1.64561552]]
```

Here `W` is the 2-dimensional representation of each iris sample (each row is an iris in latent space), and `H` contains 2 latent feature vectors of length 4. In practice, the meaning of these NMF components can be interpreted from `H`: for instance, we might find one component heavily weights petal length/width (capturing "petal size"), and another weights sepal dimensions, indicating the data's variation can be explained as a combination of "overall size" factors.

NMF is especially powerful and interpretable when features are non-negative and additive in nature (e.g., counts, intensities). It has been used for discovering additive features in images and text. Because NMF has to be solved with iterative algorithms (usually minimizing reconstruction error), results can depend on initialization and might converge to a local optimum. It's often useful to try multiple runs or use random seed for reproducibility. The number of components k is a parameter that may require tuning (e.g., via cross-validation or based on interpretability).

7 Independent Component Analysis (ICA)

Independent Component Analysis (ICA) is an unsupervised technique that seeks a linear transformation of the data into components that are **statistically independent** from each other. Unlike PCA which focuses on maximizing variance and decorrelation, ICA goes further to remove higher-order statistical dependencies, aiming to find the underlying independent *sources* that mixed together to produce the observed data. A classic example is the "cocktail party problem": given recordings from multiple microphones in a room (each microphone captures a mix of all voices/music in the room), ICA can separate the mixed signals to recover the original independent sound sources (each person's speech, music, etc.). In this scenario, the recorded signals are observed mixtures, and the independent source signals are latent variables that ICA tries to infer.

ICA assumes that the source signals are non-Gaussian and mixed linearly. It finds an unmixing matrix that when applied to the observed data yields components with maximal statistical independence (often measured by non-Gaussianity).

Applications:

- **Blind Source Separation:** Separate mixed signals into original independent sources (audio signal separation, EEG/MEG brain signal separation into independent brain activities or artifacts).
- **Feature Extraction:** Find underlying factors in financial data, telecommunication signals, or any scenario where observed data are mixtures of latent factors.
- **Image Analysis:** Separate independent textures or patterns in images (though in images, "independence" assumptions may or may not hold well).
- **Data Preprocessing:** Sometimes used to remove artifacts (e.g., separate eyeblink artifacts from EEG signals, then remove those components).

Python Example (ICA for signal separation)

To illustrate ICA, let's simulate a simple scenario: two independent signals and two sensors that record mixtures of these signals. We'll use `sklearn.decomposition.FastICA`.

```
import numpy as np
from sklearn.decomposition import FastICA

```

```python
# Simulate two independent source signals
t = np.linspace(0, 8, 1000)
s1 = np.sin(2 * np.pi * 1 * t)    # Source 1: sine wave
s2 = np.sign(np.sin(2 * np.pi * 4 * t))   # Source 2: square wave
S = np.vstack([s1, s2]).T   # shape (1000, 2), each column is a
  source

# Mix the sources with a random mixing matrix A
A = np.array([[0.6, 0.4],
[0.4, 0.81]])    # 2x2 mixing matrix
X_mixed = S.dot(A.T)    # Observations (1000 x 2)

# Apply FastICA to recover independent components
ica = FastICA(n_components=2, random_state=0)
S_ica = ica.fit_transform(X_mixed)    # ICA estimated sources
A_ica = ica.mixing_    # Estimated mixing matrix

print(np.round(ica.mixing_, 2))
```

After ICA, `S_ica` should contain signals that resemble our original `s1` and `s2` (up to scaling or ordering). We could measure the correlation between `S_ica` and true `S` to verify separation. In a real use-case, we often don't know the true sources, but ICA can still be useful (for example, separating independent factors in data for inspection).

In summary, ICA is valuable when your data is assumed to be a linear combination of hidden independent factors. It does not rank components by variance or "importance" – all independent components are treated equally, and one may need domain knowledge to interpret them. Additionally, when the independent components assumption is violated or sources are only mildly non-Gaussian, ICA might not yield meaningful results. Common algorithms for ICA (like FastICA) use measures of non-Gaussianity (e.g., kurtosis or negentropy) to perform the separation.

8 Uniform Manifold Approximation and Projection (UMAP)

UMAP is a newer non-linear dimensionality reduction technique that, like t-SNE, is often used for visualizing high-dimensional data in 2D or 3D. UMAP is founded on manifold learning and topological data analysis. It constructs a graph of the high-dimensional data (connecting data points that are nearest neighbors), and then optimizes a low-dimensional embedding such that the structure of this neighborhood graph is preserved. Intuitively, think of each data point as a node in a graph; UMAP

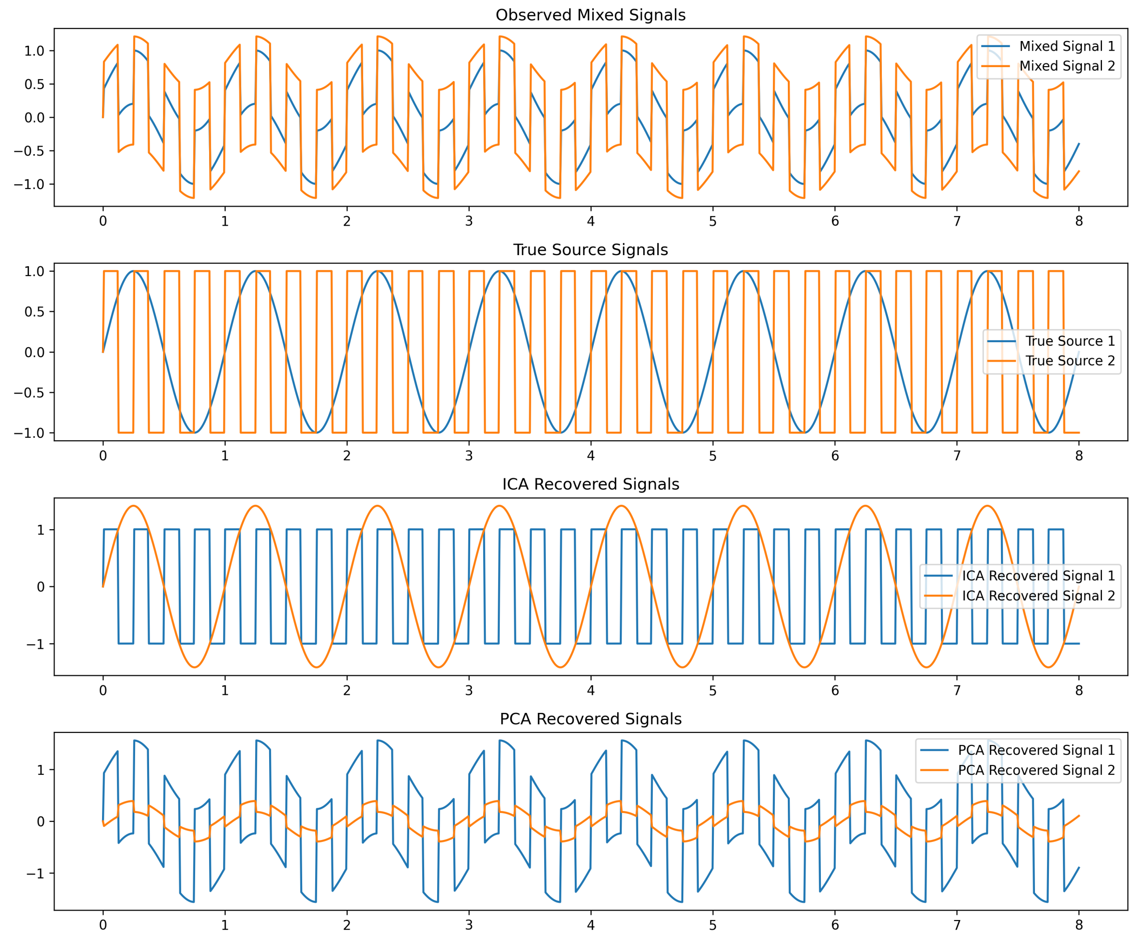

Figure 5: Illustration of ICA on a signal separation problem. The top plot shows the observed mixed signals from two sensors (each a mix of two source signals). The second plot shows the true underlying source signals (which are independent of each other). The third plot shows the signals recovered by ICA, which align almost perfectly with the true sources. The bottom plot shows what PCA would recover: PCA finds uncorrelated components, but they are not the original sources.

connects points that are close in the original space, then tries to lay out this graph in a low-dimensional space so that connected points remain close and the overall shape of the graph is retained. The result is often similar to t-SNE (clusters of similar points), but UMAP tends to better maintain some global relationships and scales well to large datasets.

Applications:

- **Visualization of High-D Data:** UMAP is widely used in bioinformatics (e.g., single-cell RNA-seq data with thousands of genes) and any domain with high-dimensional data to visualize clusters and continuous manifolds.
- **General Purpose Non-linear Reducer:** UMAP can be used as a preprocessing step to reduce dimensionality before clustering or other machine learning tasks, as it can preserve meaningful structure in fewer dimensions (and is faster than t-SNE on large data).
- **Recommender/Embedding Visualization:** Like t-SNE, used for visualizing word embeddings, graph embeddings, etc., with often better runtime performance.

UMAP has a couple of parameters like *n_neighbors* (controls local vs global blend) and *min_dist* (controls how tightly points can clump) which allow tuning the embedding. It's also deterministic given a fixed random seed (no random initialization needed by default, as it often initializes using a preliminary spectral embedding).

Python Example (UMAP on digits dataset)

UMAP is not in scikit-learn by default; we install the `umap-learn` package. Here we reduce the digits data to 2D:

```python
!pip install umap-learn
import umap
from sklearn.datasets import load_digits

# Load and preprocess data
digits = load_digits()
X = digits.data
y = digits.target

reducer = umap.UMAP(n_components=2, random_state=42)
X_umap = reducer.fit_transform(X)
print(X_umap.shape)   # (1797, 2)
```

Now `X_umap` has 2D coordinates. We can visualize it similarly with a scatter plot:

```python
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 8), dpi=300)
scatter = plt.scatter(X_umap[:, 0], X_umap[:, 1], c=y, cmap="
 Spectral", s=10, alpha=0.7)
```

```python
plt.colorbar(scatter, label="Digit Label")
plt.title("Digits Data via UMAP", fontsize=14, pad=15)
plt.xlabel("UMAP Component 1", fontsize=12)
plt.ylabel("UMAP Component 2", fontsize=12)
plt.grid(alpha=0.3)
plt.tight_layout()
plt.show()
```

If you plot the result, you should see clusters for each digit (0–9). In fact, applying UMAP to the MNIST digits dataset (a larger set of handwritten digits) typically reveals ten distinct clusters corresponding to the ten digit classes. UMAP tends to preserve some global structure too – for example, it may place similar digits (like 3, 5, and 8, which sometimes resemble each other) closer together in the plot, while keeping very different digits (like 0 vs 1) far apart.

UMAP has been shown to preserve both local and much of the global structure of data and can handle larger datasets (with approximations for nearest neighbors) more efficiently as millions of single-cell data points). However, similar caution is warranted: like t-SNE, UMAP can sometimes produce "cluster artifacts" and its plots require interpretation – distances in the embedding are not strictly meaningful in absolute terms, and different parameter settings can yield slightly different structures. Always combine such analyses with domain knowledge and, if possible, quantitative evaluation.

9 Conclusion

Dimensionality reduction is a powerful toolset for both data exploration and preprocessing. Linear methods like PCA and LDA provide simple and fast ways to reduce dimensions and uncover latent structure (unsupervised variance in PCA, supervised class structure in LDA). Non-linear methods such as t-SNE and UMAP excel at producing intuitive visualizations of complex manifolds, often revealing clustering structure that might be hidden in raw high-dimensional space. Neural network approaches like autoencoders offer flexibility to learn custom low-dimensional representations suited to reconstruction or downstream tasks. Methods like NMF and ICA target specific structural assumptions (additive parts and statistical independence, respectively) and can yield interpretable latent factors in the right contexts.

When applying these techniques in Python, libraries like `scikit-learn` (`PCA`, `LinearDiscriminantAnalysis`, `FastICA`, `NMF`, `TSNE`) and **umap-learn** (`UMAP`), or deep learning frameworks for autoencoders, make implementation straightforward. Always remember to consider the assumptions and goals of each method: for instance,

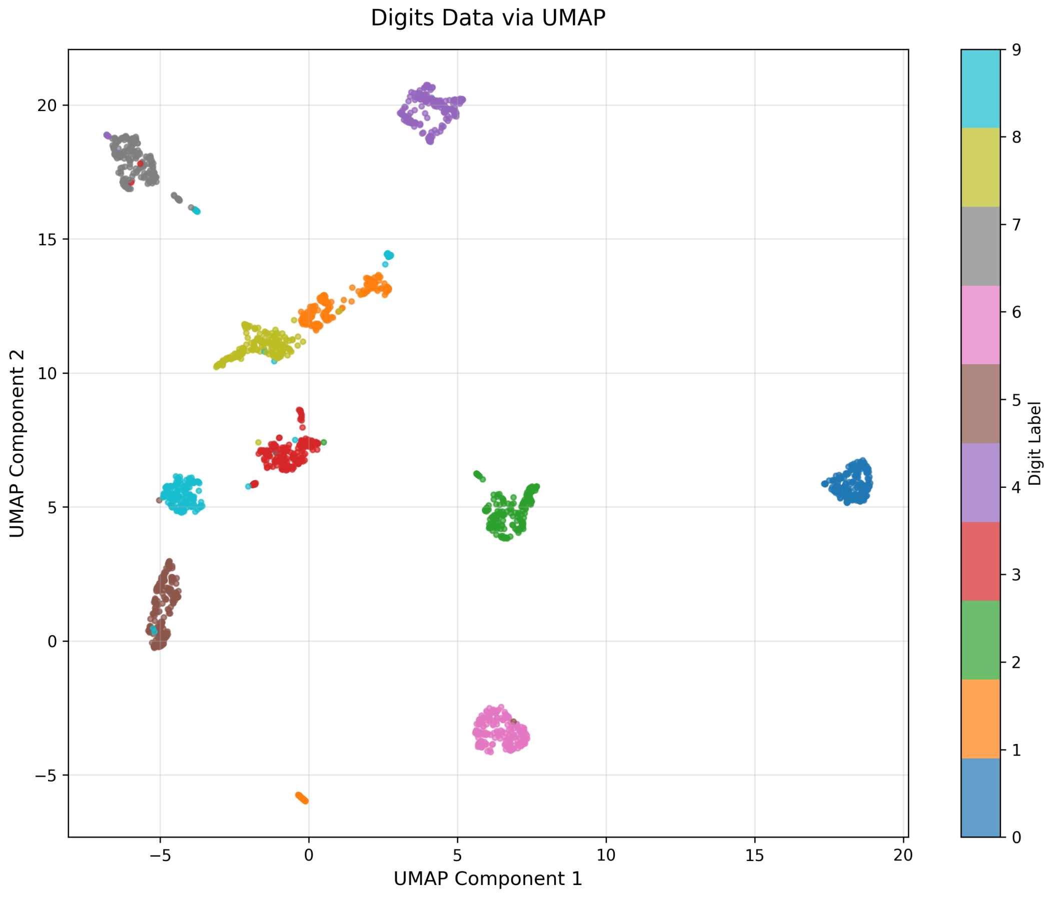

Figure 6: UMAP projection of the digits dataset onto 2D. Each point represents a digit image, colored by its label.

use PCA for speed and simplicity if linear relationships suffice, or t-SNE/UMAP if you need to untangle complex non-linear structures for visualization. By leveraging dimensionality reduction, you can distill high-dimensional data into insightful latent variables that not only make analysis and modeling easier but often provide a clearer understanding of the underlying phenomena in the data.

References

- `scikit-learn.org` - Documentation and examples for PCA, LDA, ICA, NMF, and t-SNE.

- `umap-learn.readthedocs.io` - Documentation and examples for UMAP.
- `cscolumbia.edu` - Explanation of Non-Negative Matrix Factorization (NMF) and its applications.
- `medium.com` - Explanation of Linear Discriminant Analysis (LDA) and its applications.
- `fiveable.me` - Explanation of Linear Discriminant Analysis (LDA) and its applications.
- `codesignal.com` - Explanation of t-SNE and its applications.
- `scintt-learn.org` - Explanation of Principal Component Analysis (PCA) and its applications.
- `quickonomics.com` - Explanation of latent variables and their role in dimensionality reduction.
- `distill.pub` - Explanation of t-SNE and its limitations.
- `ibm.com` - Explanation of autoencoders and their applications.
- `frcs.github.io` - Example of autoencoder reconstruction on the MNIST dataset.
- `smapplystatistics.org` - Example of UMAP on the MNIST dataset.
- `umap-learn.readthedocs.io` - Explanation of UMAP and its applications.
- `scintileandloro` - Example of ICA for signal separation.