2025-10-03 22:20

tags :

- Machine Learning
- Hands on ML - book

Chapter 8 - Hands on

- **Curse of Dimensionality**
  - Many features → training is slow + harder to find good solutions.
  - Example (MNIST): border pixels = useless, neighboring pixels = highly correlated.
- **Dimensionality Reduction**
  - Removes irrelevant/redundant features.
  - May lose some info (like JPEG compression).
  - Pros: faster training, sometimes noise reduction → better performance.
  - Cons: small accuracy drop, more complex pipelines.
  - Advice: train on original data first, then reduce if needed.
- **Uses**
  - **Speeding up training**.
  - **Visualization**: reduce to 2D/3D → see clusters/patterns, communicate results to non-technical people.
- **Approaches**
  1. **Projection**
  2. **Manifold Learning**
- **Techniques**
  - PCA (Principal Component Analysis)
  - Random Projection
  - LLE (Locally Linear Embedding)

The Curse of Dimensionality

- **Definition**:
  Refers to problems that arise when working with high-dimensional data. As dimensions grow, the feature space expands exponentially, data becomes sparse, and models struggle to learn meaningful patterns.

Key Effects

1. **Sparsity of Data**
   - In high dimensions, most points lie near the borders of the space.
   - Distances between random points grow much larger as dimensions increase (e.g., ~0.52 in 2D vs ~408 in 1,000,000D).
   - Result: data points are far apart → harder for models (like k-NN, clustering) that rely on proximity.
2. **Loss of Predictive Power**
   - New points are usually far from training points → predictions rely on large, unreliable extrapolations.
   - **Overfitting risk**: models may memorize noise because there are too many irrelevant features.
   - **Underfitting risk**: when data is too sparse, models fail to capture useful patterns.
3. **Exponential Data Requirements**
   - To maintain density in high dimensions, dataset size must grow exponentially.
   - Example: with 100 features (values in [0,1]), covering space well would require more samples than atoms in the observable universe.
4. **Increased Computational Complexity**
   - Training gets slower as algorithms must process more features.
   - Storage and memory costs increase significantly.

Takeaway

- More dimensions = more sparsity, less reliable predictions, risk of overfitting, and huge data/computation requirements. This is why **dimensionality reduction** is critical in ML.

Main Approaches for Dimensionality Reduction

- There are two primary approaches for reducing the number of features in a dataset:
  1. **Projection**
  2. **Manifold Learning**

projection

- **Core Idea:** In real-world data, training instances often exist within a much lower-dimensional **subspace** of the high-dimensional space.
- **Why it works:** Many features are constant or highly correlated, meaning the data doesn't use the full volume of the high-dimensional space.
- **The Process:**
  - Identify the lower-dimensional subspace (e.g., a plane in 3D space).
  - **Project** every data point perpendicularly onto this subspace.

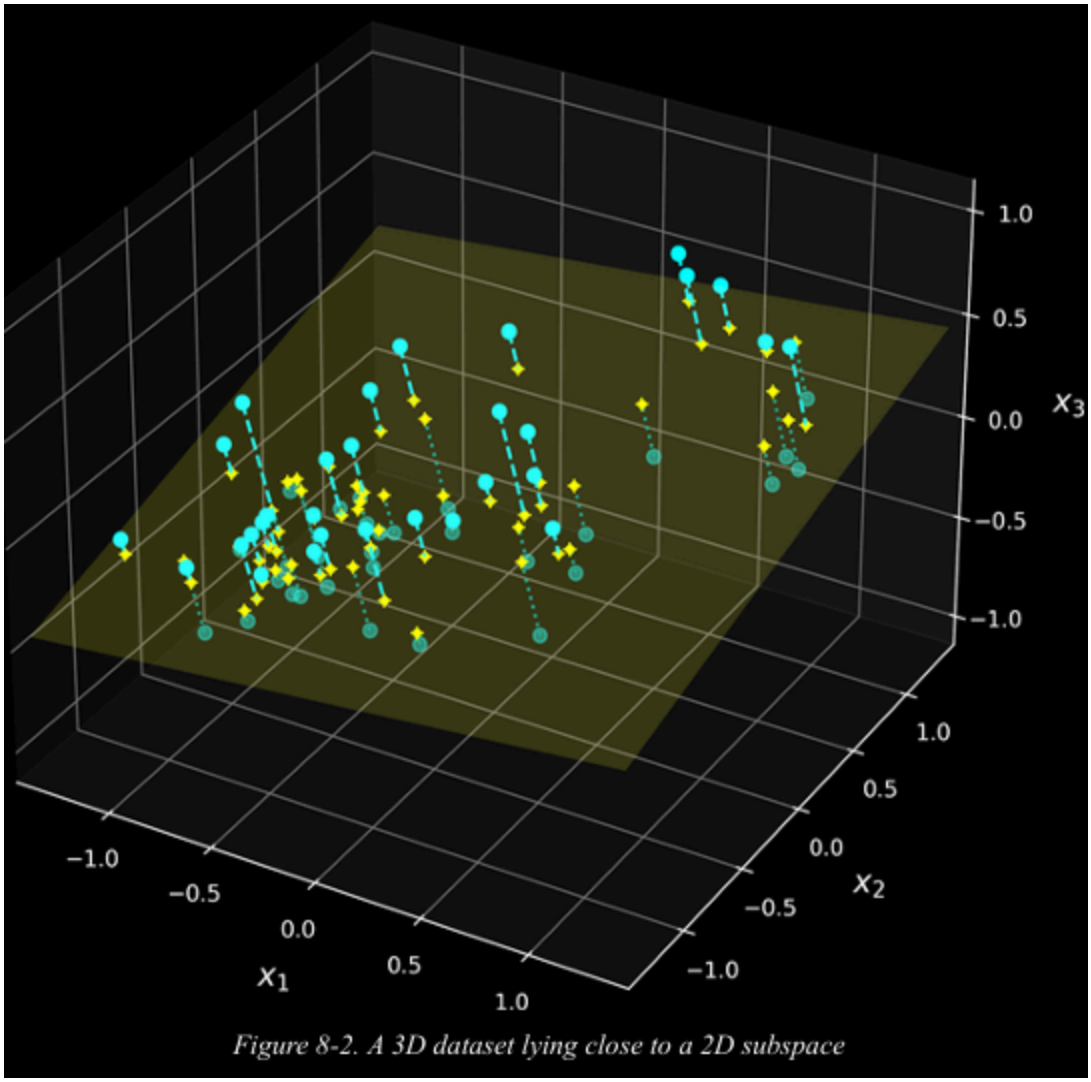
- **Result:** A new, lower-dimensional dataset where the axes are new features (the coordinates on the subspace).
- **Example (from the text):**
  - **Figure 8-2:** A 3D dataset where all points lie close to a 2D plane (a subspace).
  - 

    Figure 8-2. A 3D dataset lying close to a 2D subspace
  - **Figure 8-3:** The new 2D dataset created by projecting all points onto that plane. The dimensions are reduced from 3D to 2D.

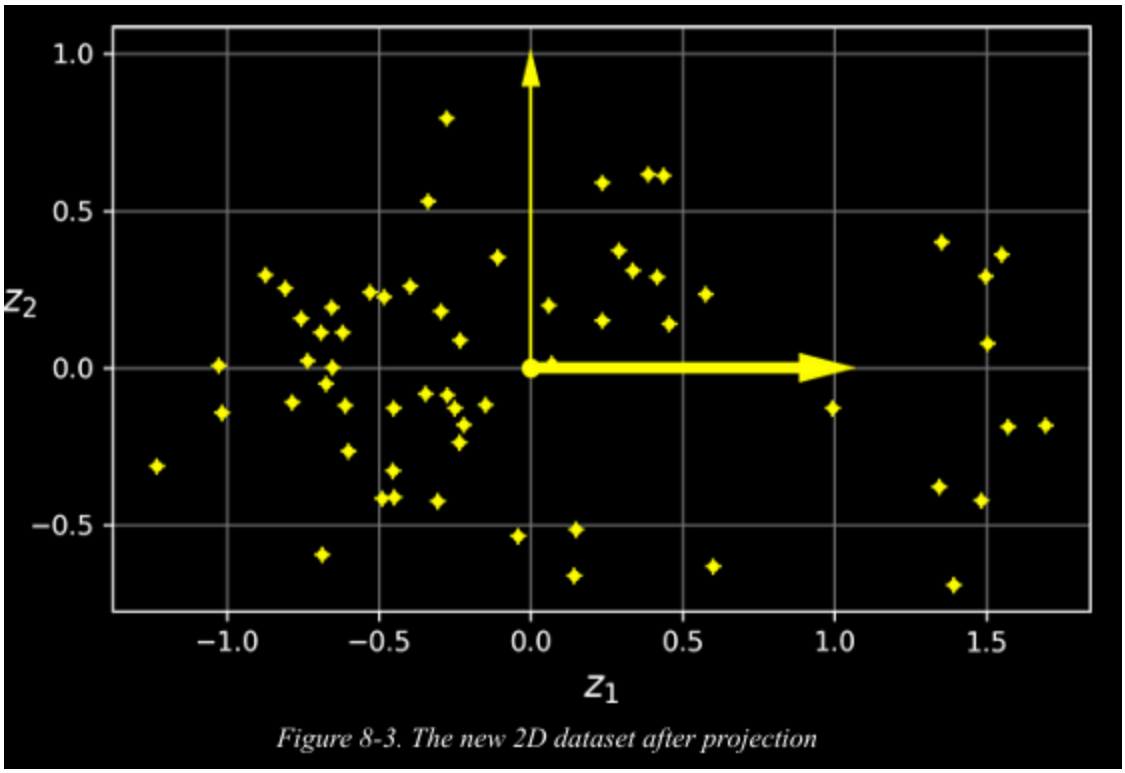


Figure 8-3. The new 2D dataset after projection

Manifold learning

- **Core Concept:** Data often lies on a lower-dimensional manifold (a bent/curved surface) within a high-dimensional space. Manifold learning aims to "unroll" this structure to reduce dimensions while preserving the true relationships in the data.

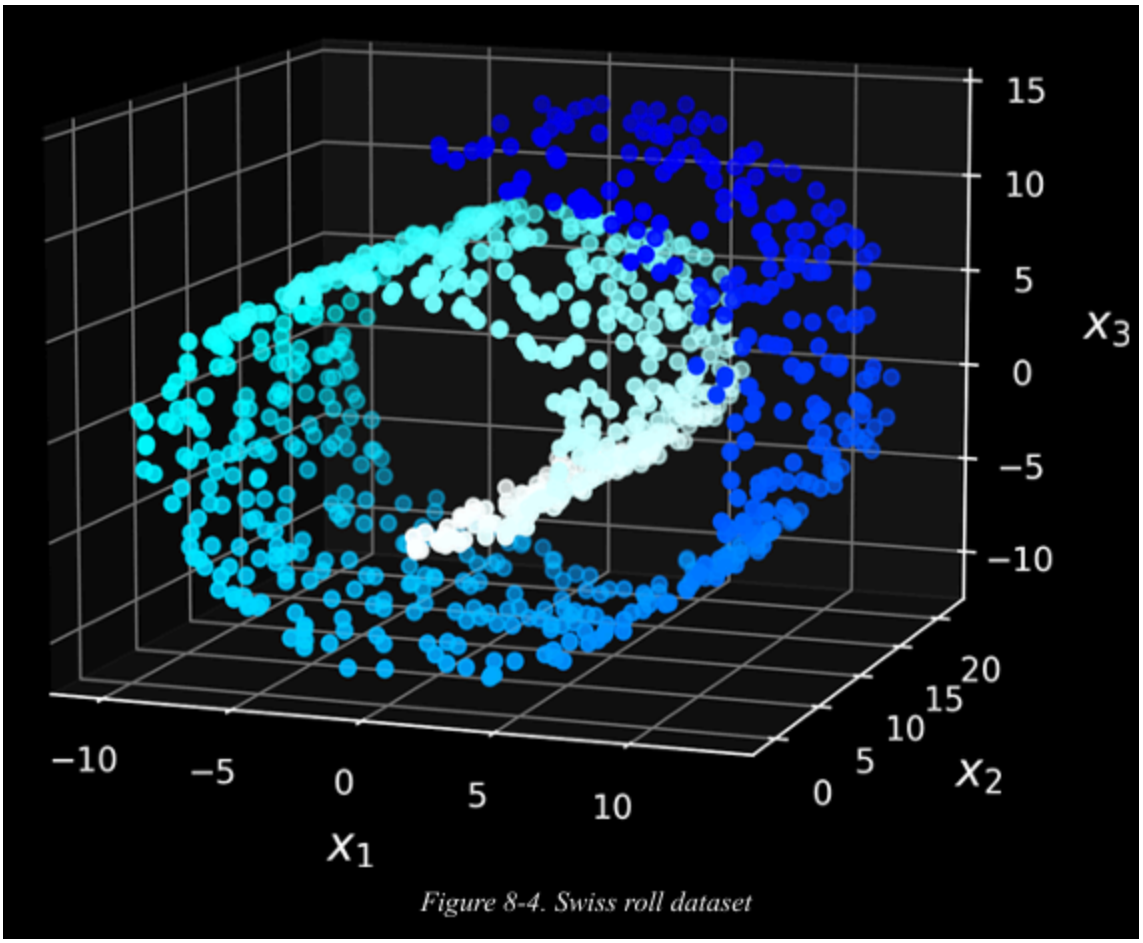


Figure 8-4. Swiss roll dataset

- **The Limitation of Projection (Illustrated by Figure 8-5 - Left Side)**
  - **Problem:** Simply projecting complex data (like the **Swiss Roll in Figure 8-4**) onto a plane **squashes** its layers together.
  - **Result:** The underlying structure is destroyed, and data points that are far apart on the manifold become neighbors.

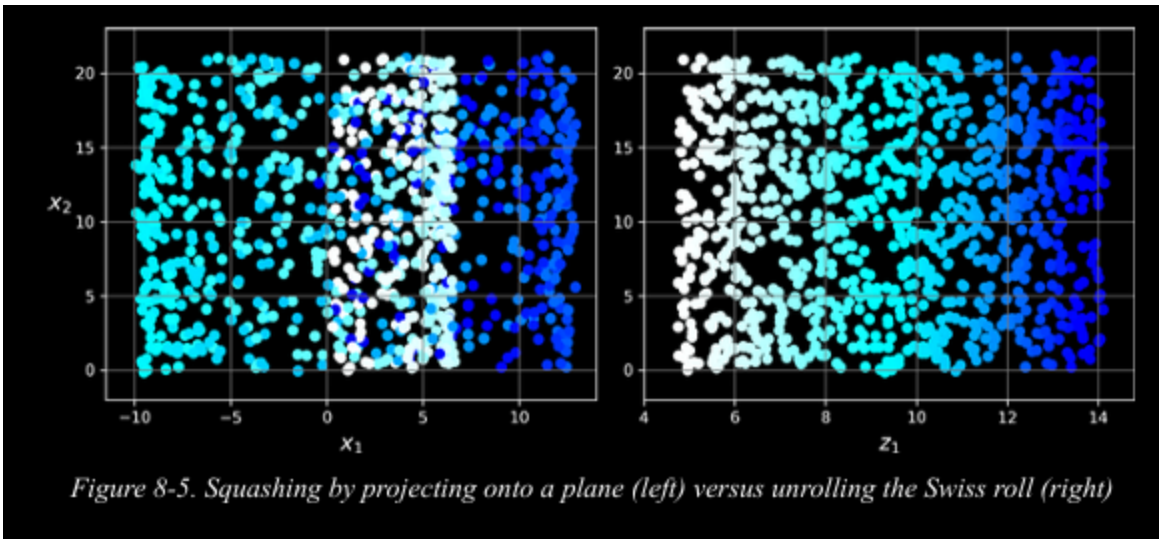


Figure 8-5. Squashing by projecting onto a plane (left) versus unrolling the Swiss roll (right)

- **The Goal: Manifold Unrolling (Illustrated by Figure 8-5 - Right Side)**
  - **Solution:** Instead of squashing, the goal is to **unroll the manifold**.
  - **Result:** This reveals the data's true 2D structure, where distances and relationships

are meaningful.

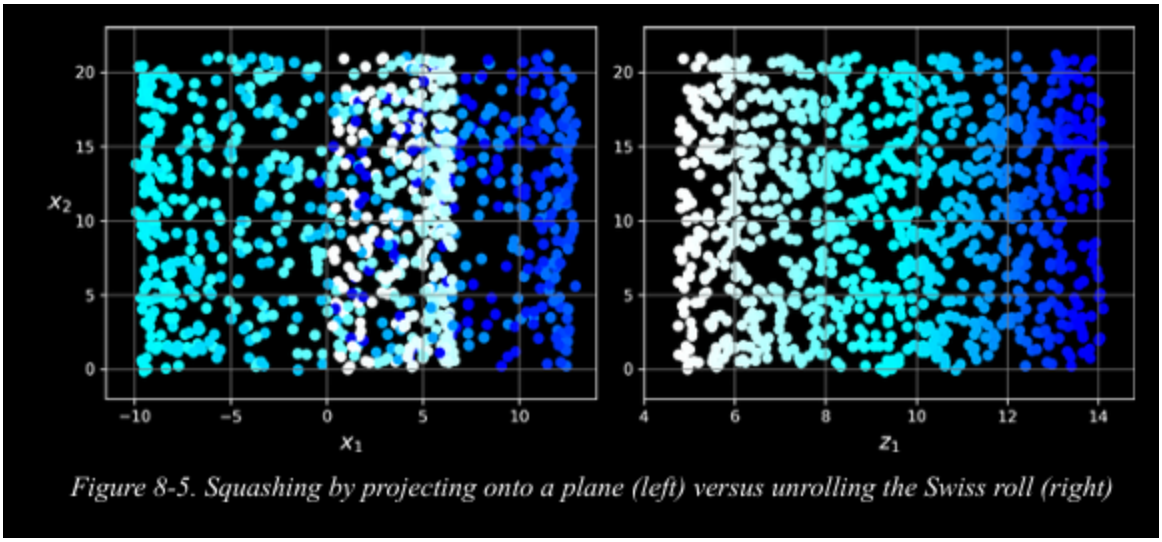


Figure 8-5. Squashing by projecting onto a plane (left) versus unrolling the Swiss roll (right)

- **The Manifold Assumption**
  - This approach works based on the assumption that most real-world high-dimensional data lies on or near a much lower-dimensional manifold.
- **A Crucial Caveat (Illustrated by Figure 8-6)**
  - **Implicit Hope:** A lower-dimensional representation will simplify tasks like classification.
  - **Reality Check (Figure 8-6):** This is **not always true**.
    - A complex 3D boundary can become a simple line in 2D.
    - Conversely, a simple 3D boundary can become a complex, fragmented line in 2D.

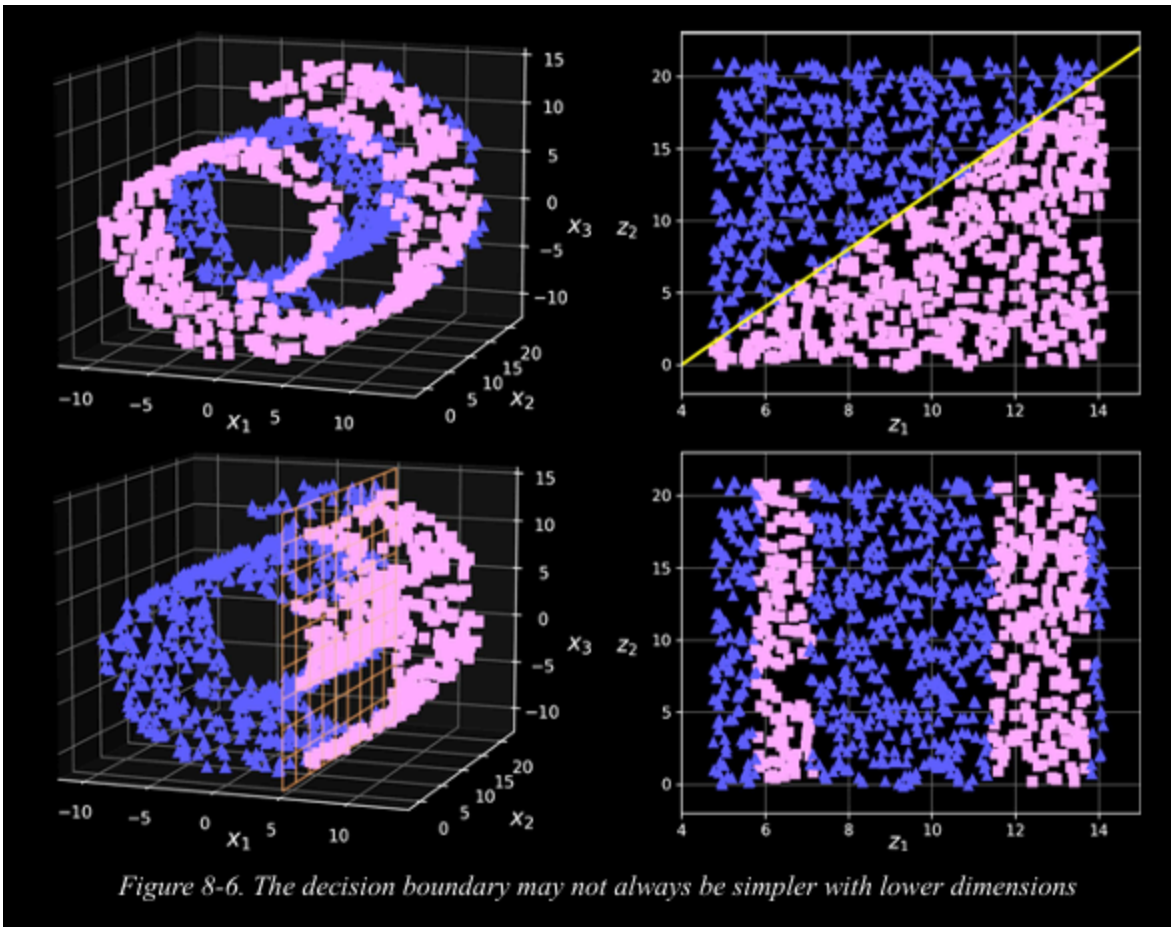


Figure 8-6. The decision boundary may not always be simpler with lower dimensions

- **Conclusion:** Dimensionality reduction speeds up training, but doesn't automatically yield a better model; the outcome is dataset-dependent.

PCA

- The most popular dimensionality reduction algorithm based on projection

Preserving the Variance

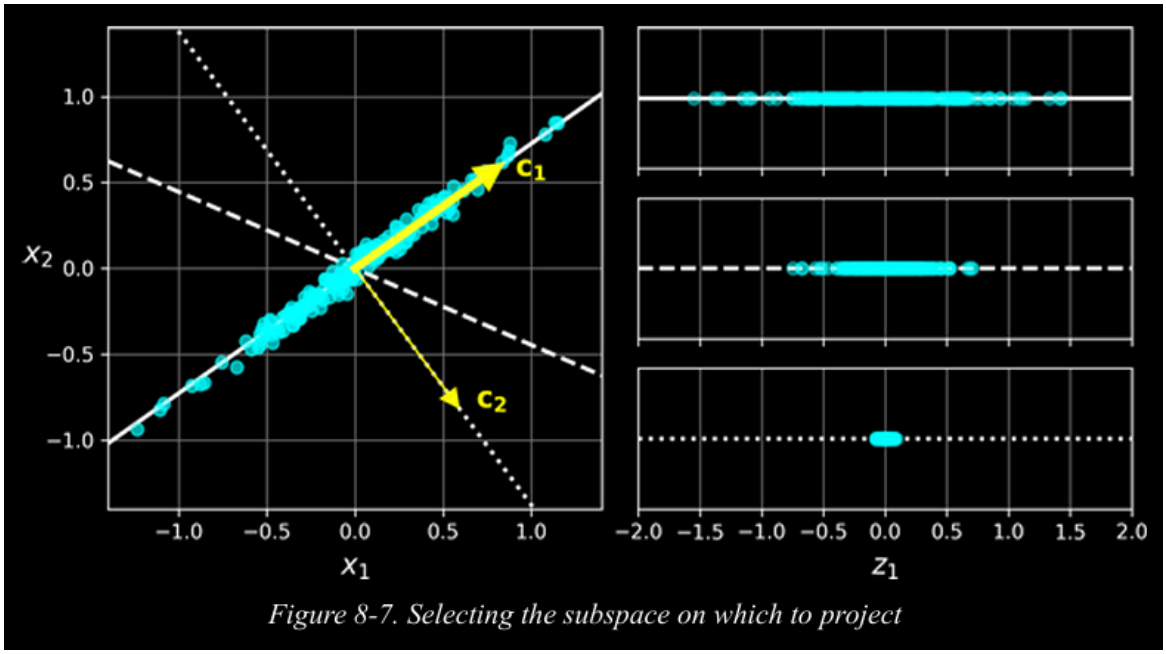
- **The Core Problem:** Before projecting data onto a lower-dimensional hyperplane (like a line or plane), you must first choose the **right hyperplane** to project onto.
- **The Solution: Maximize Variance**
  - When you project a dataset onto different axes, the results look very different.
  - As shown in **Figure 8-7**:
    - Projecting onto one axis preserves a **large amount of the data's spread (variance)**.
    - Projecting onto another axis compresses the data, preserving **very little variance**.
  - 

Figure 8-7. Selecting the subspace on which to project
- **The Rationale:**
  - It is reasonable to select the axis that **preserves the maximum amount of variance**.
  - **Why?** This choice is likely to lose the least information. A projection with high variance retains more of the dataset's essential structure.
  - An equivalent justification: this axis **minimizes the mean squared distance** between the original data points and their projected versions.

Principal Components

- **Core Concept:** PCA identifies a new set of axes (principal components) for the data, ordered by how much variance they capture.
  - **1st PC:** The axis that accounts for the **largest amount of variance**.
  - **2nd PC:** The axis **orthogonal** (at a 90-degree angle) to the first that accounts for the largest amount of *remaining* variance.
  - **Subsequent PCs:** This process repeats, finding each new axis orthogonal to all previous ones. The `i-th` axis is the `i-th` Principal Component.
- **Key Property:** Each PC is a **zero-centered unit vector** (a direction). The sign (positive/negative) of this vector can flip if the data is perturbed, but the axis it defines remains the same.

How to Find the Principal Components

There are two main mathematical approaches to find these components.

- **Method 1: Eigen Decomposition of Covariance Matrix**
  This is the classic theoretical approach.
  1. **Center the Data:** Subtract the mean from each feature so the dataset is centered around the origin.
  2. **Calculate Covariance Matrix:** Compute the matrix that describes how features vary together.
  3. **Find Eigenvectors & Eigenvalues:** Perform eigen decomposition on the covariance matrix.
     - The **eigenvectors** are the Principal Components (the directions).
     - The **eigenvalues** represent the amount of variance each PC captures.
  4. **Sort & Select:** Sort the eigenvectors by their eigenvalues in descending order and choose the top `N` eigenvectors.
  5. **Project Data:** Transform the original data by projecting it onto the selected eigenvectors (the new axes).
- **Method 2: Singular Value Decomposition (SVD)**
  This is the preferred, more efficient, and numerically stable method used in practice.
  - The SVD algorithm directly decomposes the centered data matrix `X` into three matrices: $X = U \Sigma V^T$
  - The crucial matrix for PCA is `V` (or V^T). The columns of the `V` matrix are the **unit vectors that define all the principal components**.
  - The first column of `V` is the first PC, the second column is the second PC, and so on.
  - **Python SVD Example:**

```
X_centered = X - X.mean(axis=0)  # Step 1: Center the data
U, s, Vt = np.linalg.svd(X_centered) # Step 2: Perform SVD
```

```
c1 = Vt[0]  # First Principal Component
c2 = Vt[1]  # Second Principal Component
```

Important Warning: PCA assumes the dataset is centered around the origin. Always center your data first if implementing it yourself like this case!

Projecting Down to d Dimensions

Goal: Reduce the dataset's dimensionality to `d` dimensions while preserving the maximum possible variance.
Method: Project the data onto the hyperplane defined by the first `d` principal components.

The Projection Process

1. **Identify Principal Components:** First, perform PCA to get all principal components. The matrix `V` contains these components as its columns, sorted by variance (PC1 is first column, PC2 is second, etc.).
2. **Create the Projection Matrix (`W_d`):**
   - This matrix is formed by taking the **first `d` columns** of the `V` matrix.
   - `W_d` = [c_1, c_2, ..., c_d] where `c_i` is the i-th principal component.
3. **Project the Data:** To get the lower-dimensional dataset, perform a matrix multiplication between the **centered** original data matrix `X` and the projection matrix `W_d` .
   Equation: X_{d-proj} = X · W_d
   - `X` : Centered training set matrix (n instances × m features)
   - `W_d` : Projection matrix (m features × d components)
   - `X_{d-proj}` : Reduced dataset (n instances × d features)

Python Code Example

```
# X_centered is the original data after mean normalization
# Vt is the matrix of principal components from SVD

# 1. Create projection matrix using first 2 PCs
W2 = Vt[:2].T  # Transpose to get correct dimensions

# 2. Project the data to 2D
X2D = X_centered @ W2  # Matrix multiplication
# X2D is now the 2D version of your dataset
```

Result: You now have a dataset `X2D` with only `d` dimensions that retains as much of the original variance as possible.

Using Scikit-Learn

```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
X2D = pca.fit_transform(X)  # Does everything automatically!
```

Key Points:

- Automatically centers data
- Uses SVD internally
- `pca.components_` contains the PC directions
- First row = most important PC

Explained Variance Ratio

- **What it is:** Shows how much variance each PC captures
- **How to get it:** `pca.explained_variance_ratio_`
- **Example:** `[0.76, 0.15]` means:
  - 1st PC: 76% of total variance
  - 2nd PC: 15% of total variance
  - 3rd PC: ~9% (the rest) - probably not important
  Use: Helps decide how many components to keep!

Choosing the Right Number of Dimensions

- **Special Case: Data Visualization**
  - If reducing dimensions for plotting, just use 2 or 3 dimensions regardless of variance explained
- Instead of guessing how many dimensions to keep, use these systematic approaches:

Method 1: Target Variance Threshold (Most Common)

- **Idea:** Keep enough dimensions to preserve a target percentage of total variance (e.g., 95%)
- **Implementation:**

```
# Automatic - Scikit-Learn finds the right number
pca = PCA(n_components=0.95)  # Preserve 95% of variance
X_reduced = pca.fit_transform(X_train)

# Check how many components were kept
print(pca.n_components_)  # e.g., 154 for MNIST
```

- **Manual way to find the number:**

```
pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1  # Find where we hit 95%
```

Method 2: The Elbow Method (Visual Approach)

- **Idea:** Plot cumulative explained variance and look for the "elbow"
- **Process:**
  1. Calculate cumulative sum of explained variance ratios
  2. Plot number of dimensions vs. cumulative variance
  3. Identify point where adding more dimensions gives diminishing returns
- **As shown in Figure 8-8:** The curve flattens around 100 dimensions for MNIST

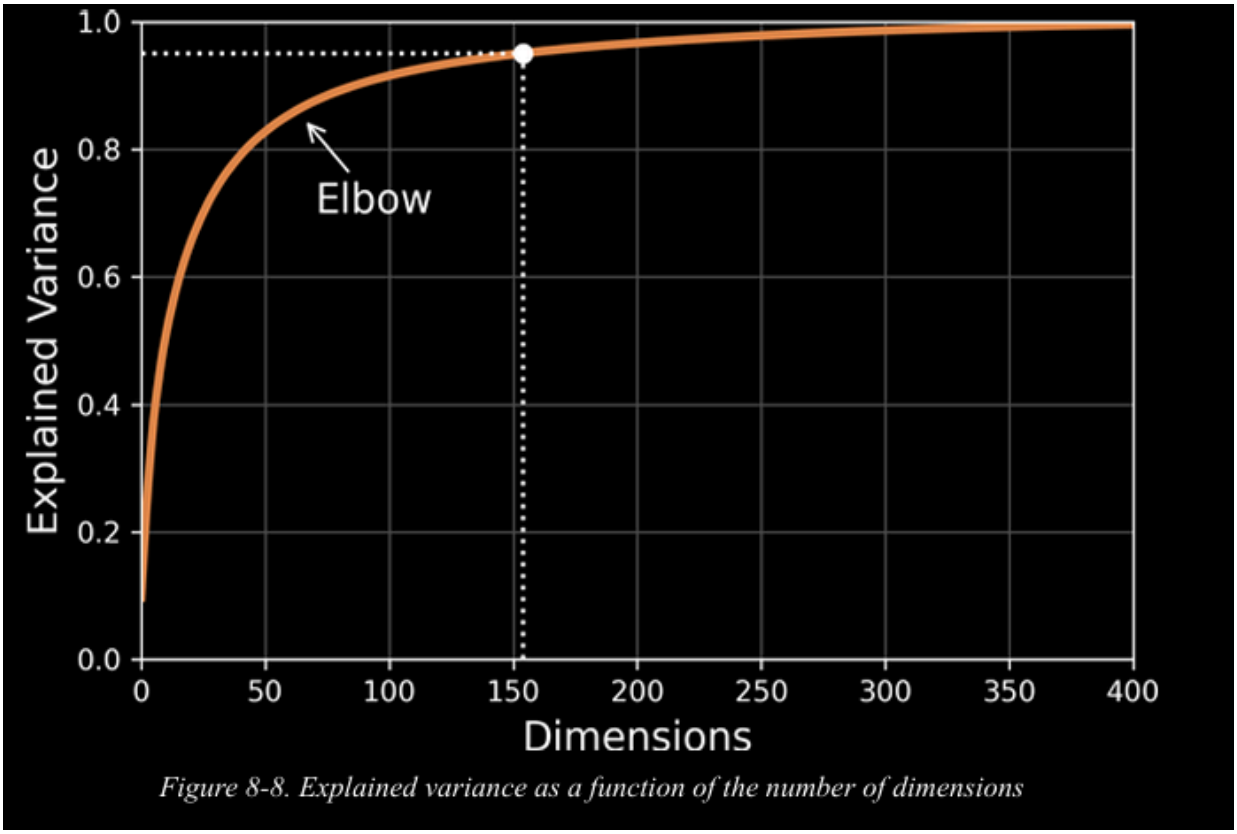


Figure 8-8. Explained variance as a function of the number of dimensions

Method 3: Hyperparameter Tuning (For ML Pipelines)

- **Use case:** When using PCA as preprocessing for supervised learning (classification/regression)
- **Approach:** Treat `n_components` as a hyperparameter and optimize it
- **Example with Random Forest:**

```python
from sklearn.pipeline import make_pipeline
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV

# Create pipeline
clf = make_pipeline(
    PCA(random_state=42),
    RandomForestClassifier(random_state=42)
)

# Search for best parameters
param_distrib = {
    "pca__n_components": np.arange(10, 80),
    "randomforestclassifier__n_estimators": np.arange(50, 500)
}

rnd_search = RandomizedSearchCV(clf, param_distrib, n_iter=10, cv=3)
rnd_search.fit(X_train[:1000], y_train[:1000])
```

- **Result Example:** Might find that only 23 dimensions are optimal for Random Forest on MNIST
- **Key Insight:** The optimal number depends on your model - powerful models (like Random Forest) need fewer dimensions than linear models.

PCA for Compression

- **Compression Benefits**
  - **Massive Size Reduction:** MNIST goes from 784 features → 154 features
  - **High Efficiency:** Less than 20% of original size while keeping 95% of variance
  - **Performance Boost:** Much faster classification algorithms
- **How Compression Works**
  1. **Project:** `X_reduced = X · W_d` (reduce to d dimensions)
  2. **Store:** Keep only the reduced data and projection matrix W_d
- **Decompression & Reconstruction**
  - **Reverse Process:** $X_{recovered} = X_{reduced} \cdot W_d^T$
  - **Scikit-Learn Method:** `pca.inverse_transform(X_reduced)`
  - **Reconstruction Error:** Measures quality loss (MSE between original and reconstructed)
- **Quality Trade-off**
  - **As shown in Figure 8-9:**
    - Left: Original MNIST digits
    - Right: Reconstructed after 95% variance compression

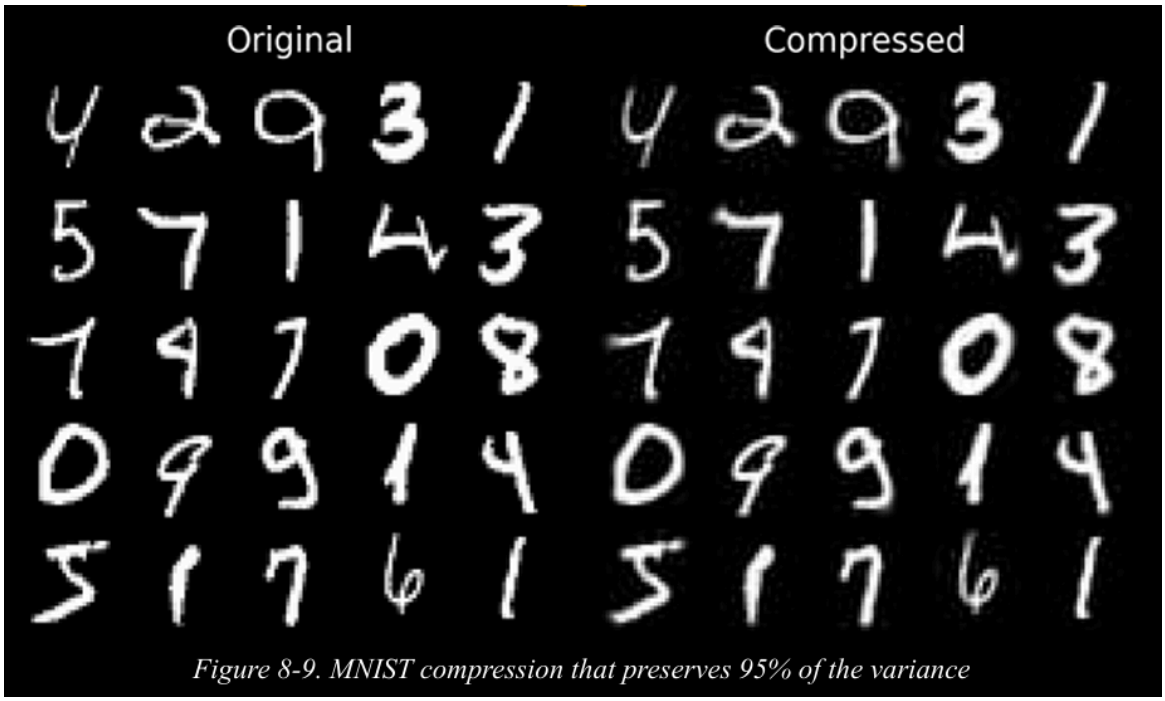


Figure 8-9. MNIST compression that preserves 95% of the variance

  - **Observation:** Slight quality loss, but digits remain clearly recognizable
  - **Practical:** Good balance between compression and information preservation
- **Key Applications**
  1. **Storage Optimization** - Save disk space
  2. **Fast Processing** - Speed up ML algorithms
  3. **Noise Reduction** - Reconstruction often removes minor variations

  Bottom Line: PCA = Smart lossy compression that keeps what matters most!

Randomized PCA

- **What is it?**
  - **Stochastic algorithm** that quickly finds approximate principal components
  - **Much faster** than full SVD when `d << n` (much fewer components than features)
- **Performance Advantage**
  - **Computational Complexity:** $O(m \times d^2) + O(d^3)$ vs Full SVD: $O(m \times n^2) + O(n^3)$
  - **Dramatic speedup** when keeping few components from high-dimensional data
- **Usage in Scikit-Learn**

```
rnd_pca = PCA(n_components=154, svd_solver="randomized", random_state=42)
X_reduced = rnd_pca.fit_transform(X_train)
```

- **Automatic Selection**
  - **Default:** `svd_solver="auto"`
  - **Uses Randomized PCA when:**
    - `max(m, n) > 500` AND
    - `n_components < 80% of min(m, n)`

    - **Example:** MNIST (m=60,000, n=784) with 154 components → uses randomized PCA automatically
- **When to Use Full SVD**
    - For slightly more precise results
    - Set `svd_solver="full"`
    - Only needed when precision is critical

Incremental PCA

- **Problem with Standard PCA**
    - Requires **entire dataset in memory**
    - Fails with **very large datasets**
- **Solution: Incremental PCA**
    - Processes data in **mini-batches**
    - Useful for:
        - Large training sets
        - Online learning (new instances arriving continuously)

Two Implementation Methods

- **Method 1: Manual Mini-Batches**

```
from sklearn.decomposition import IncrementalPCA

n_batches = 100
inc_pca = IncrementalPCA(n_components=154)

# Manual loop - you control batching
for X_batch in np.array_split(X_train, n_batches):
    inc_pca.partial_fit(X_batch)  # Must use partial_fit

X_reduced = inc_pca.transform(X_train)
```

- **Method 2: Memory-Mapping with Automatic Batching**
    - **Step 1: Create Memory-Mapped File**

```
filename = "my_mnist.mmap"
X_mmap = np.memmap(filename, dtype='float32', mode='write',
shape=X_train.shape)
X_mmap[:] = X_train  # Copy data to disk file
X_mmap.flush()  # Force save to disk
```

    - **Step 2: Use with Automatic Batching**

```
# Re-open for reading (data stays on disk)
X_mmap = np.memmap(filename, dtype="float32",
mode="readonly").reshape(-1, 784)

batch_size = X_mmap.shape[0] // n_batches
inc_pca = IncrementalPCA(n_components=154, batch_size=batch_size)
inc_pca.fit(X_mmap)  # Automatic batching - just call fit once!
```

    - Only the raw binary data is saved to disk, so you need to specify the data type and shape of the array when you load it. If you omit the shape, np.memmap() returns a 1D array.

- **Key Advantage of Method 2:**
    - **Simpler:** No manual loops
    - **Automatic:** IPCA handles batching internally
    - **Memory efficient:** Only one batch loaded at a time
- **When to Use Each Method**
    - **Manual (Method 1):** When you have control over data loading
    - **Automatic (Method 2):** When data is already in a memory-mapped file
    - **Both:** Give identical results!

The 3 Components:

1. `IncrementalPCA` - The algorithm itself
2. `np.memmap` - How you store/access data
3. `batch_size` - How you control processing

- **All Possible Combinations:**
    1. **Combination 1: IncrementalPCA Only**
    2. **Combination 2: IncrementalPCA + memmap Only**
    3. **Combination 3: IncrementalPCA + batch_size Only**
    4. **Combination 4: IncrementalPCA + memmap + batch_size**

Random Projection

- **Core Concept**
    - Projects data to lower dimensions using a **random linear projection**
    - Surprisingly preserves distances well (Johnson-Lindenstrauss lemma)
    - **Trade-off:** More dimension reduction = more distance distortion + information loss
- Choosing the Right Dimensions

- **Johnson-Lindenstrauss Equation:** Determines minimum dimensions to preserve distances between points within tolerance `ε`
- Usage:
  `python from sklearn.random_projection import johnson_lindenstrauss_min_dim m, ε = 5000, 0.1 d = johnson_lindenstrauss_min_dim(m, eps=ε) # e.g., 7300`
- **Key insight:** Depends only on number of instances `m`, not features `n`

- **Two Implementation Methods**
  1. Gaussian Random Projection
     - **Matrix:** Dense, Gaussian distribution (mean=0, variance=1/d)
     - **Usage:**

```
from sklearn.random_projection import GaussianRandomProjection
gaussian_rnd_proj = GaussianRandomProjection(eps=0.1,
random_state=42)
X_reduced = gaussian_rnd_proj.fit_transform(X)
```

     - **Pros:** Simple, good distance preservation
     - **Cons:** Memory intensive for large `n`
  2. Sparse Random Projection **(Usually Better)**
     - **Matrix:** Sparse (mostly zeros), faster and more memory-efficient
     - **Default density:** `1/√n` (very sparse for large n)
     - **Usage:**

```
from sklearn.random_projection import SparseRandomProjection
sparse_rnd_proj = SparseRandomProjection(random_state=42)
X_reduced = sparse_rnd_proj.fit_transform(X)
```

     - **Advantages:**
       - Much less memory
       - Faster computation
       - Preserves sparsity in input data
       - Comparable quality to Gaussian
- **Key Properties**
  - **No Training:** Random matrix generation depends only on data shape, not content
  - **Efficient:** Simple matrix multiplication
  - **Memory:** Sparse version uses significantly less memory
- Inverse Transformation (Approximate)

```
components_pinv = np.linalg.pinv(gaussian_rnd_proj.components_)
X_recovered = X_reduced @ components_pinv.T
```

  - **Warning:** Pseudo-inverse computation can be slow for large matrices

    - **Result:** Approximate reconstruction, not exact
- When to Use
  - **High-dimensional data** where PCA is too slow
  - **Large or sparse datasets** (use SparseRandomProjection)
  - **Quick dimensionality reduction** without training
  - **Memory-constrained** environments

  Bottom Line: Fast, simple, memory-efficient alternative to PCA for very high dimensions!

LLE

- Core Concept
  - **Nonlinear dimensionality reduction** technique
  - **Manifold learning** that doesn't rely on projections
  - **Key idea:** Preserves local linear relationships between neighbors
  - **Great for** unrolling twisted manifolds like the Swiss roll
- **How LLE Works - 2 Steps**
  - Step 1: Find Local Linear Relationships
    - For each point, find its **k-nearest neighbors**
    - Reconstruct each point as a **linear combination** of its neighbors
    - Find weights `w_i,j` that minimize reconstruction error
    - **Constraints:**
      - Weights sum to 1 for each point
      - Zero weight for non-neighbors
  - Step 2: Map to Lower Dimensions
    - Find low-dimensional representation `Z` that preserves the local relationships
    - Keep the weights fixed, optimize positions in low-dimensional space
    - Minimize difference between actual positions and weighted neighbor positions
- **Implementation**

```
from sklearn.manifold import LocallyLinearEmbedding

lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10,
random_state=42)
X_unrolled = lle.fit_transform(X_swiss)
```

- **Results (Figure 8-10)**

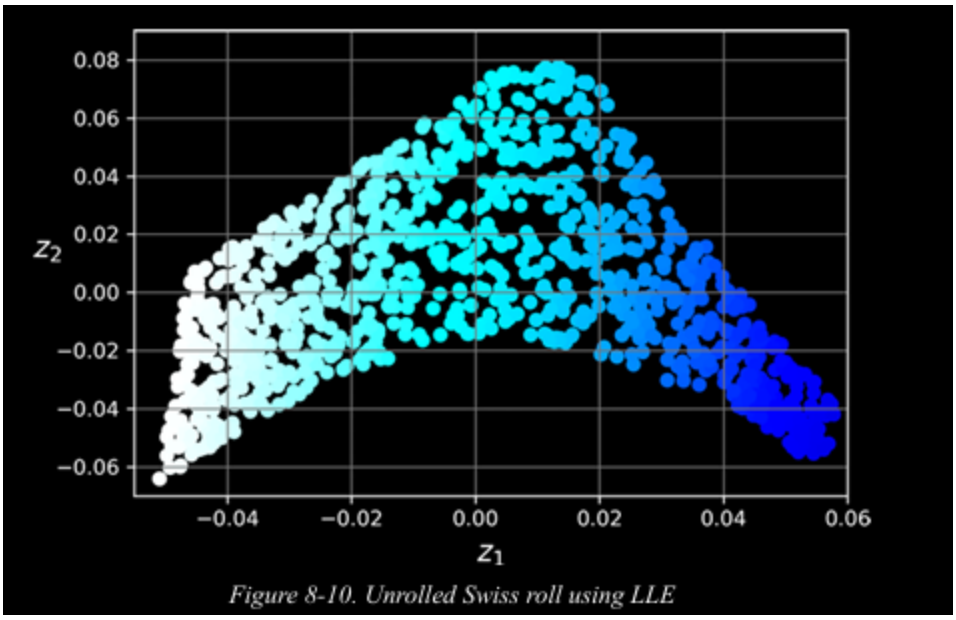
- 

Figure 8-10. Unrolled Swiss roll using LLE
  - **Locally preserves distances** well
  - **Globally may distort** (Swiss roll becomes twisted band, not perfect rectangle)
  - **Good at modeling** the underlying manifold structure
- **Pros & Cons**
  - **Advantages:**
    - Excellent for nonlinear manifolds
    - Preserves local structure well
    - Better than projection methods for twisted data
  - **Disadvantages:**
    - **Poor scalability:** $O(m^2)$ complexity for large datasets
    - Sensitive to noise
    - Global geometry may be distorted
- **When to Use**
  - **Small to medium datasets** (due to scalability)
  - **Nonlinear manifolds** (Swiss roll, S-curves)
  - **When local relationships** are more important than global structure
  - **Data visualization** of manifold structures

  Bottom Line: Powerful for nonlinear data, but watch the dataset size!

Other Dimensionality Reduction Techniques

1. MDS (Multidimensional Scaling)
   - **Goal:** Preserve distances between instances
   - **Best for:** Low-dimensional data (unlike random projection)
   - **Result in Figure 8-11:** Flattens Swiss roll but preserves global curvature
   - **Use case:** When maintaining global distances is important

2. Isomap
    - **Approach:** Creates graph connecting nearest neighbors, preserves **geodesic distances** (shortest path distances)
    - **Result in Figure 8-11:** Drops the roll structure entirely
    - **Use case:** When you care about the intrinsic manifold distances
3. t-SNE (t-distributed Stochastic Neighbor Embedding)
    - **Goal:** Keep similar instances close, dissimilar instances apart
    - **Main use: Visualization** (especially cluster visualization)
    - **Result in Figure 8-11:** Flattens roll, preserves some curvature, amplifies clusters
    - **Characteristics:** Can tear data apart to separate clusters
    - **Common use:** Visualizing MNIST digits, high-dimensional clusters
4. LDA (Linear Discriminant Analysis)
    - **Unique aspect: Supervised** technique (uses class labels)
    - **Goal:** Find axes that best separate classes
    - **Use case:** Dimensionality reduction before classification
    - **Benefit:** Projects data to keep classes maximally separated

Comparison from Figure 8-11:

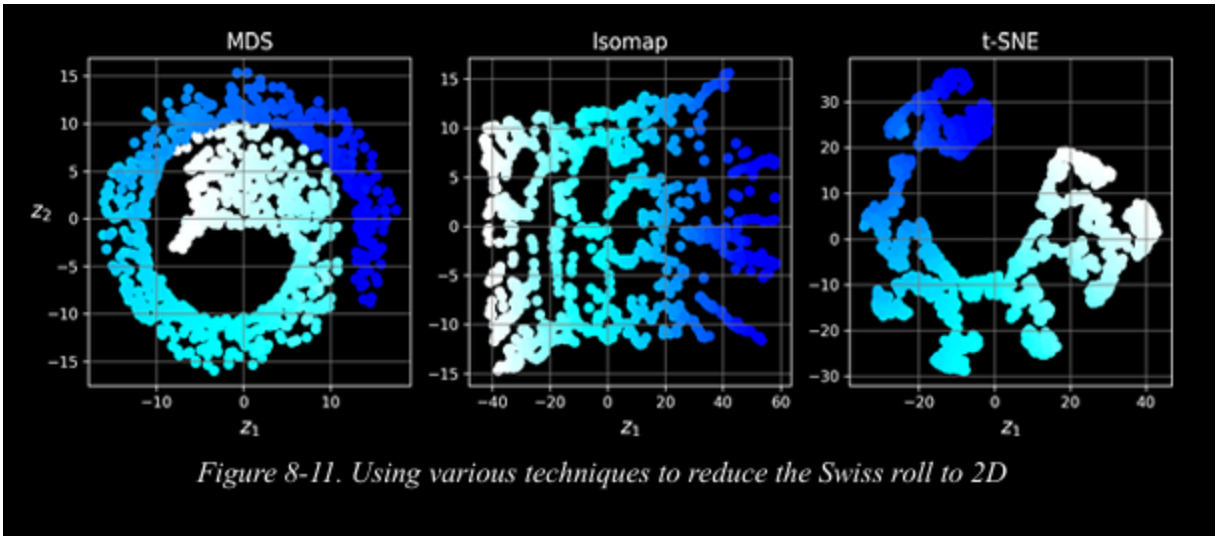


Figure 8-11. Using various techniques to reduce the Swiss roll to 2D

Technique	Swiss Roll Result	Key Characteristic
MDS	Preserves curvature	Maintains global structure
Isomap	Drops roll entirely	Uses graph distances
t-SNE	Flattens + amplifies clusters	Great for visualization
LLE	Unrolls but twists	Preserves local relationships

When to Use Each:

- **MDS:** Preserving global distance relationships
- **Isomap:** Manifold learning with graph-based distances
- **t-SNE: Data visualization** and cluster exploration
- **LDA: Supervised** dimensionality reduction for classification
- **LLE:** Nonlinear manifolds with local structure preservation

---

Resources :

-

Related notes :

-

References :

- **Internal :**
  -
  -
  -
- **External :**
  - hegab videos
  - the book
  - the notebook
  -