IMEN574/IDSC574 Programing for Data Science

12. **Dimensionality Reduction**

Spring 2026

Instructor: **Sangwoo Mo**

Today's Agenda

01 Why DR?
Curse of dimensionality, why we reduce

02 PCA & SVD
PCA via SVD, PPCA & FA, Autoencoders

03 LDA
Supervised DR, class separation

04 t-SNE & UMAP
Nonlinear viz, hyperparams, pitfalls & comparison

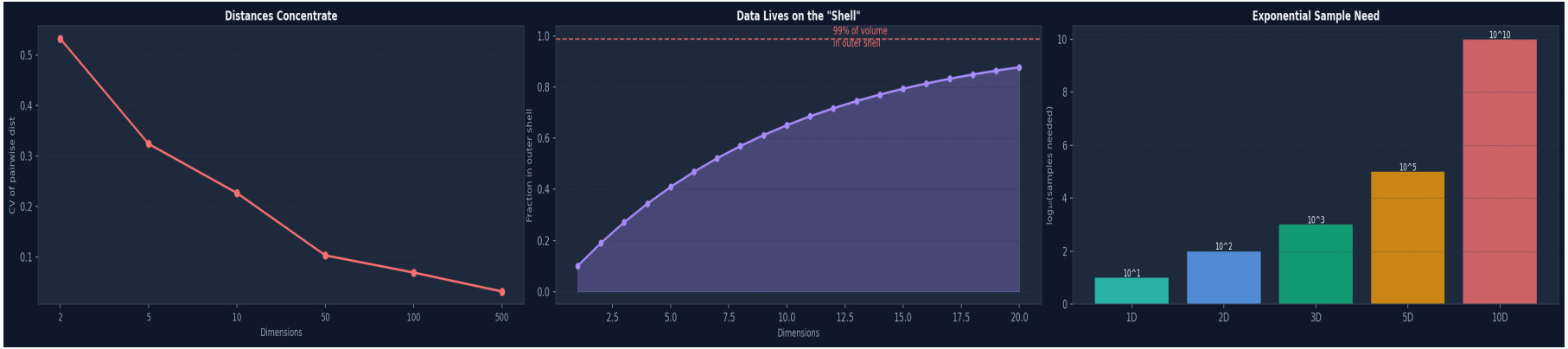
05 Feature Selection
Filter, Wrapper, Embedded methods & code

06 Practice
Algorithm comparison, end-to-end examples

The Curse of Dimensionality

Why do we need dimensionality reduction? Because high dimensions cause problems.

Distances Become Meaningless	In high dims, all points become equidistant from each other. Similarity metrics fail.
Sparsity	Data becomes extremely sparse. Most regions of space are empty. Hard to find patterns.
Overfitting Risk	More features = easier to fit noise instead of signal. Model generalizes poorly.



What Is Dimensionality Reduction?

Project high-dimensional data onto a lower-dimensional subspace while preserving important structure (variance, distances, or labels).

Feature Extraction

Create new features that are combinations of original ones:
- PCA: rotate to variance
- LDA: optimize class separation
- t-SNE/UMAP: preserve structure in 2D

Feature Selection

Choose subset of original features:
- Remove redundant/irrelevant
- Faster, more interpretable
- sklearn.feature_selection

Main libraries: sklearn.decomposition, sklearn.manifold, umap-learn

DR Taxonomy

DR methods vary by supervision level and linearity.

Unsupervised

No labels available. Preserve variance, distances, or structure.

Linear: PCA

Nonlinear: t-SNE, UMAP, Autoencoders

Supervised

Labels available. Maximize class separation or predictive power.

Linear: LDA

Nonlinear: Manifold learning with labels

Preprocessing Reduce noise, speed up training

Visualization 2D/3D plots of high-dim data

Interpretability Understand key patterns

Principal Component Analysis

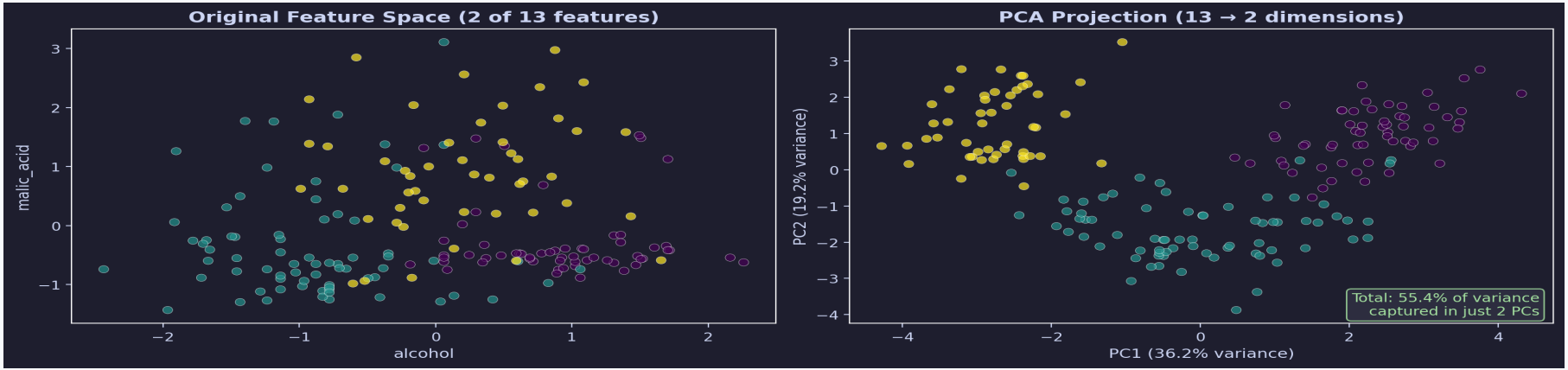
The foundation of linear dimensionality reduction

PCA — Core Idea

Find directions of maximum variance in the data. Rotate coordinate system to align with these directions.

Key Concepts:

- First PC captures most variance; Second PC captures next-most, orthogonal to first
- Continue for all dimensions; usually only need first 2-3



PCA — Covariance & Eigenvectors

Understanding the mathematics: what does covariance tell us? How do eigenvectors help?

Covariance Matrix $\Sigma = (1/n)\, X^T X$ (from centered data)

Σ[i,j] = covariance between feature i and feature j. Diagonal = variance of each feature. Off-diagonal = relationships.

High covariance = features move together. Low covariance = independent. We want to find directions of high variance = high covariance directions.

Eigendecomposition: $\Sigma v = \lambda v$

Eigenvector v = a direction in space.
Eigenvalue λ = how much variance the data has along direction v.
Principal Component = eigenvector with highest eigenvalue (most variance).
Second PC = next-highest eigenvalue, perpendicular to first.

Tip: Eigenvectors point to directions of high variance; eigenvalues tell us how much variance lies in each direction.

PCA — Step by Step

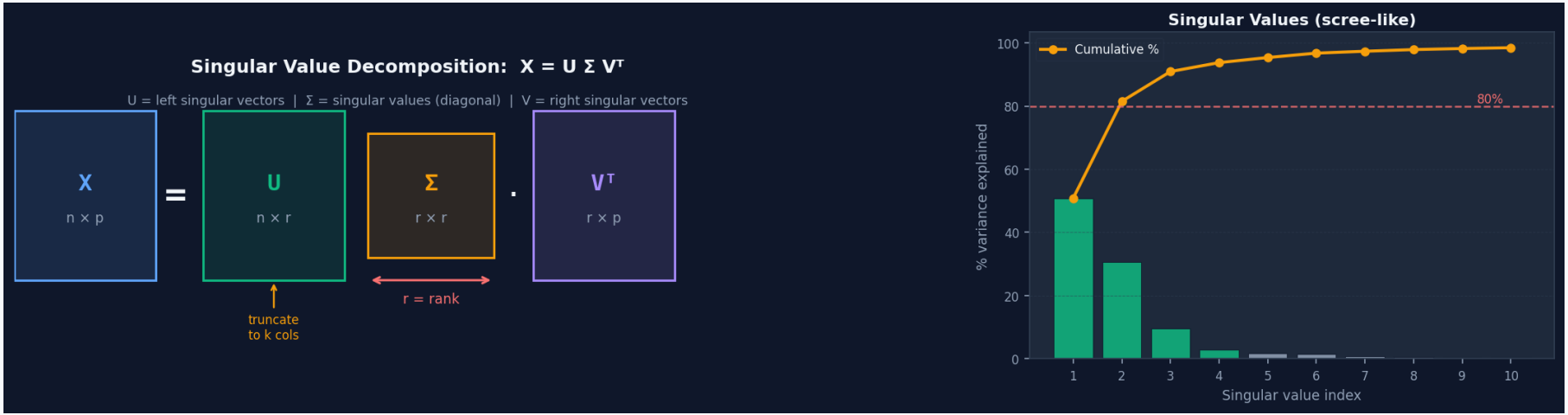
The algorithm: a clear pipeline from raw data to reduced data.

1	Center data	Subtract mean from each feature (X = X - mean(X))
2	Compute covariance	Σ = (1/n) X^TX from centered data
3	Eigendecomposition	Find eigenvectors v and eigenvalues λ from Σ
4	Sort & select	Sort by eigenvalue (descending). Keep top k PCs.
5	Project	Z = X · V_k (project onto k principal components)

Output: Z (n × k) with only the k most important components. Much smaller than original X (n × p)!

Solving PCA via SVD

PCA can be solved via eigendecomposition of Σ, or equivalently via SVD of X. SVD is numerically more stable and efficient.



Eigendecomposition vs SVD

Eigendecomposition: $\Sigma = V\Lambda V^T$ (compute covariance first). SVD: $X = U\Sigma V^T$ (directly on data). Both give the same principal components V. SVD avoids forming Σ explicitly.

Why SVD for PCA?

Numerically stable (avoids $\Sigma = X^TX$). Efficient for tall matrices (n >> p). Works on sparse matrices. sklearn PCA uses SVD internally!

PCA via SVD — In Practice

```python
import numpy as np
from sklearn.decomposition import TruncatedSVD, PCA

# --- TruncatedSVD for sparse/large data ---
svd = TruncatedSVD(n_components=50, random_state=42)
X_svd = svd.fit_transform(X_sparse)   # works on sparse matrices!

print(f"Singular values: {svd.singular_values_[:5]}")
print(f"Variance explained: {svd.explained_variance_ratio_.sum():.1%}")

# --- Compare with full SVD via numpy ---
U, sigma, Vt = np.linalg.svd(X_dense, full_matrices=False)
X_reconstructed = U[:, :50] @ np.diag(sigma[:50]) @ Vt[:50, :]

# --- PCA is SVD on centered X ---
# These two are equivalent:
pca = PCA(n_components=50); X_pca = pca.fit_transform(X_centered)
# vs: center X, then TruncatedSVD

# --- Reconstruction error (information loss) ---
err = np.mean((X_dense - X_reconstructed) ** 2)
print(f"Reconstruction MSE with 50 components: {err:.4f}")
```

PCA — Code

```python
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import numpy as np

# MUST scale first!
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Fit PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

print(f"Explained variance ratio: {pca.explained_variance_ratio_}")
print(f"Cumulative: {np.cumsum(pca.explained_variance_ratio_)}")
print(f"Loadings shape: {pca.components_.shape}")  # (n_comp, n_feat)
```

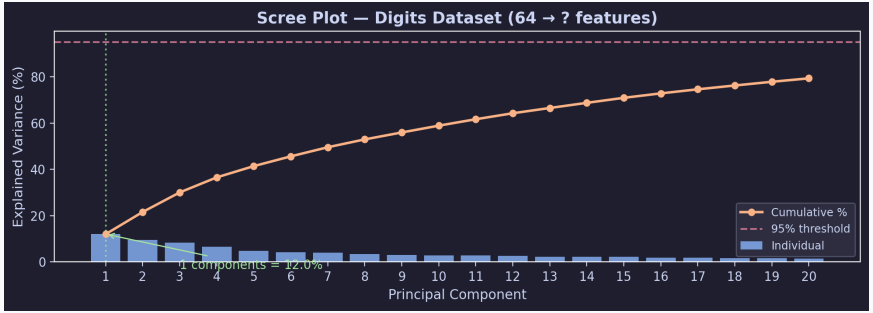
- explained_variance_ratio_: what % of total variance each PC captures. Sum to 1.0.
- components_: eigenvectors as rows. Use for interpretation (feature loadings).

Explained Variance & Choosing n_components

How many PCs do we need? Use cumulative explained variance.

Scree Plot

Plot explained_variance_ratio_ vs. component index. Look for the elbow (diminishing returns). Beyond elbow, each extra PC adds little info.

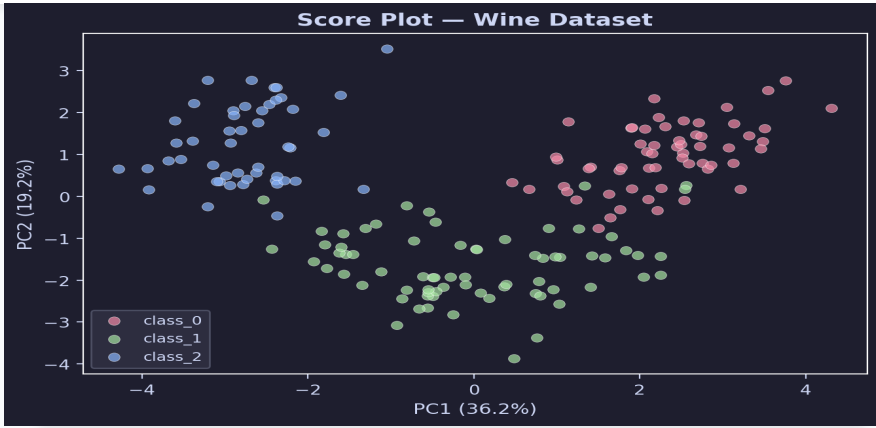


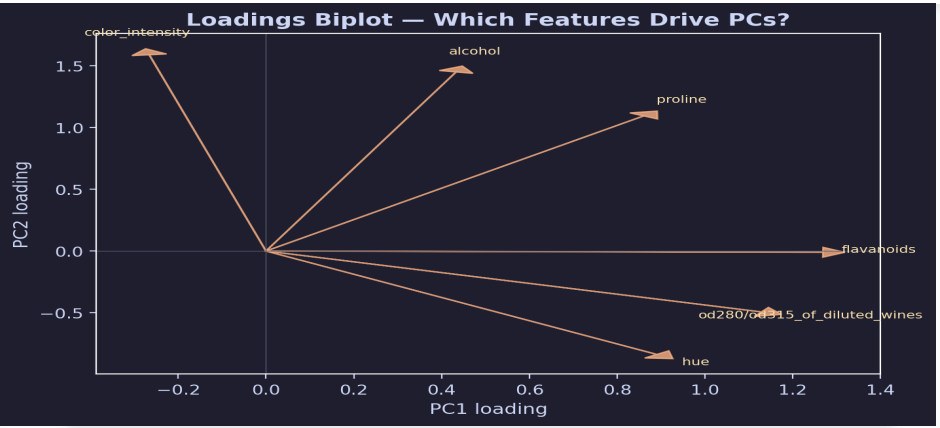
Threshold Approach: Choose k such that cumulative variance ≥ 95%. Common thresholds: 90%, 95%, 99%. Balances info loss vs. reduction.

```
# Auto: specify as fraction (0 < n < 1)
pca = PCA(n_components=0.95)  # 95%
X_reduced = pca.fit_transform(X_scaled)
print(f"{pca.n_components_}/{X.shape[1]} features")

# Manual threshold
pca = PCA().fit(X_scaled)
cumsum = np.cumsum(
    pca.explained_variance_ratio_)
n = np.argmax(cumsum >= 0.95) + 1
```

PCA — Visualization & Interpretation






```python
# Score plot (project data onto first 2 PCs)
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap='viridis')
plt.xlabel(f'PC1 ({pca.explained_variance_ratio_[0]:.1%})')
plt.ylabel(f'PC2 ({pca.explained_variance_ratio_[1]:.1%})')
plt.title('PCA Score Plot'); plt.show()

# Loadings (biplot): which features drive each PC?
for i, feat in enumerate(feature_names):
    plt.arrow(0, 0, pca.components_[0,i]*3,
              pca.components_[1,i]*3, head_width=0.1)
    plt.text(pca.components_[0,i]*3.2,
             pca.components_[1,i]*3.2, feat)
plt.title('PCA Loadings'); plt.show()
```

PCA — Practical Considerations

Scaling is Mandatory

Different feature scales = different variances. Always StandardScaler() before PCA.

Sensitive to Outliers

Outliers inflate variance. May need RobustScaler or outlier removal first.

Linear Assumption

Can't capture nonlinear structure (spiral, Swiss roll). For nonlinear data, use t-SNE, UMAP, or Autoencoders.

No Label Info

PCA is unsupervised — ignores y. If you have labels, LDA gives better class separation.

Interpretability

PCs are combinations of all features — hard to explain. Use loadings to understand.

PCA for Preprocessing & ML Pipelines

PCA as a preprocessing step before classification or regression reduces noise and speeds training.

Why Use PCA?

- Noise reduction: drops low-variance dims
- Speed: fewer features = faster training
- Decorrelation: PCs are orthogonal

When NOT to Use

If original features are already interpretable & important for domain (e.g., medical diagnosis). PCA mixes them into black-box components.

```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression

pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('pca', PCA(n_components=10)),
    ('clf', LogisticRegression(max_iter=1000))
])
pipe.fit(X_train, y_train)
print(f"Accuracy: {pipe.score(X_test, y_test):.3f}")

# GridSearchCV over n_components
from sklearn.model_selection import GridSearchCV
params = {'pca__n_components': [2, 5, 10, 20, 50]}
gs = GridSearchCV(pipe, params, cv=5)
gs.fit(X_train, y_train)
print(f"Best: {gs.best_params_}")
```

PCA: Two Ways to Think About It

The same algorithm, but a completely different story about what's happening.

Mechanistic View

- Find max-variance directions
- Eigendecompose $\Sigma = X^T X/n$
- Project: $Z = XV$
- Just geometry. No model of how data was generated.

Probabilistic View

- Data was generated: $y = Cx + \text{noise}$
- Latent factors $x \sim N(0, I)$
- Fit C via maximum likelihood
- A story about WHERE data comes from.

The Payoff:

Probabilistic view gives us a generative model. This unlocks: missing data handling, model comparison via likelihood, uncertainty quantification, and natural extensions to PPCA and Factor Analysis.

One Model, Different Noise

y = Cx + noise. Every DR method is THIS model with a different noise assumption about R.

PCA: $R \to 0$	No noise assumed. All variation = latent factors. Solution: eigenvectors of sample covariance. Simple but unrealistic — sensors always have noise.
Probabilistic PCA: $R = \sigma^2 I$	Every sensor has same noise σ^2. Now we have a likelihood! Enables model comparison, handles missing data via EM. Posterior shrinks toward zero.
Factor Analysis: R = diag(ψ_1..ψ_p)	Each sensor has its own noise ψ_i. Model learns which sensors to trust. Precision-weighted fusion: noisy sensors down-weighted. Most flexible.

Why Go Probabilistic?

Likelihood → Model Selection

$P(Y|\theta)$ lets you compare models. How many latent dimensions k? Compare log-likelihoods for different k. No more subjective scree plots.

EM → Missing Data

Sensor 7 failed today? EM marginalizes over missing values automatically. PCA has no mechanism for incomplete data.

Posterior → Uncertainty

$P(x|y)$ gives you not just a point estimate but a full distribution over latent factors. You know how confident you are.

Bayesian → Auto Complexity

Put priors on C and R → Bayesian PCA. Irrelevant dimensions get pruned automatically. No need to choose k by hand!

sklearn Implementation

sklearn.decomposition.FactorAnalysis(n_components=k) for FA. PCA internally uses SVD, but FA uses EM — fits the generative model directly.

PPCA & Factor Analysis — Code

```python
from sklearn.decomposition import PCA, FactorAnalysis
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import StandardScaler

X_scaled = StandardScaler().fit_transform(X)

# --- Probabilistic PCA (same API as PCA!) ---
ppca = PCA(n_components=5)  # PCA uses SVD internally
X_ppca = ppca.fit_transform(X_scaled)
ll_ppca = ppca.score(X_scaled)  # log-likelihood!
print(f"PPCA log-likelihood: {ll_ppca:.2f}")

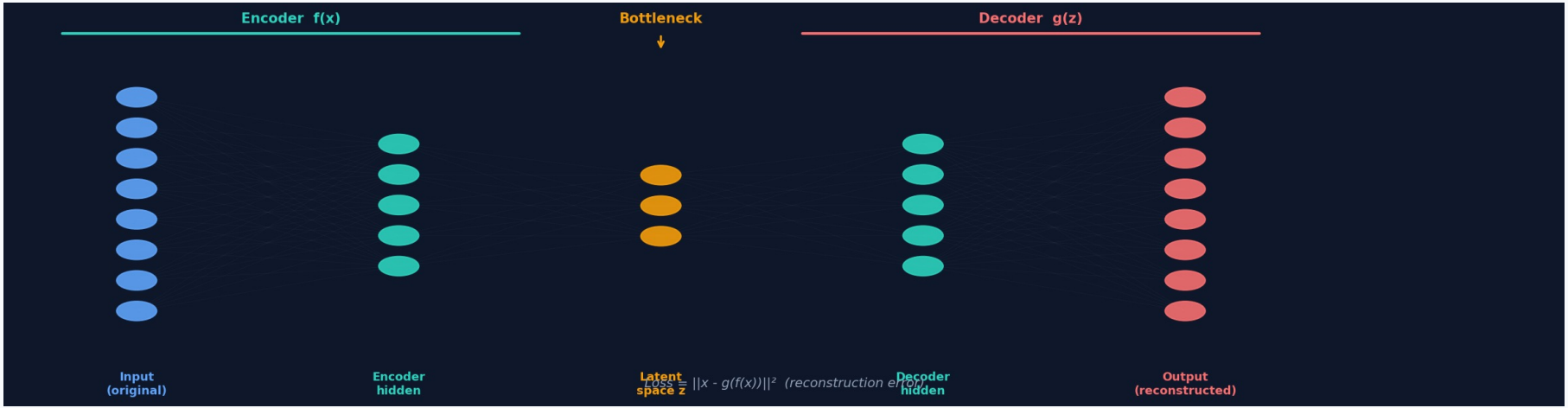
# --- Factor Analysis (EM-based, per-sensor noise) ---
fa = FactorAnalysis(n_components=5, random_state=42)
X_fa = fa.fit_transform(X_scaled)
ll_fa = fa.score(X_scaled)
print(f"FA log-likelihood:   {ll_fa:.2f}")
print(f"Noise variances:     {fa.noise_variance_[:5]}")

# --- Model selection: choose k via likelihood ---
for  in  2  5  10  20
```

- PCA.score() returns log-likelihood (probabilistic view). FactorAnalysis learns per-feature noise via EM. Use cross-validation on log-likelihood to select optimal k.
- PCA.score() returns log-likelihood (probabilistic view). FactorAnalysis learns per-feature noise via EM. Use cross-validation on log-likelihood to select optimal k — no more subjective scree plots!

Autoencoders — Nonlinear PCA

PCA is linear — Autoencoders learn nonlinear compression via neural nets. Encoder → bottleneck → Decoder. Bottleneck = reduced representation.



Advantages

- Captures nonlinear structure (unlike PCA)
- Scales to images, text, time-series

Limitations

- Needs more data & compute than PCA
- Latent space not interpretable

Autoencoder in Practice

```python
import torch
import torch.nn as nn

class Autoencoder(nn.Module):
    def __init__(self, input_dim, latent_dim):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, 128), nn.ReLU(),
            nn.Linear(128, latent_dim))
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 128), nn.ReLU(),
            nn.Linear(128, input_dim))
    def forward(self, x):
        z = self.encoder(x)       # compress
        return self.decoder(z)     # reconstruct

ae = Autoencoder(input_dim=784, latent_dim=32)
optimizer = torch.optim.Adam(ae.parameters(), lr=1e-3)
loss_fn = nn.MSELoss()

# Training loop (1 epoch shown)
for batch in dataloader:
    x_hat = ae(batch)
    loss = loss_fn(x_hat, batch)
    optimizer.zero_grad(); loss.backward(); optimizer.step()
```

Linear Discriminant Analysis

Supervised dimensionality reduction

LDA — Core Idea

Supervised: uses class labels. Maximizes between-class variance / minimizes within-class variance.

PCA (Unsupervised)

- No labels used
- Maximize total variance
- Global structure
- Works for any task

LDA (Supervised)

- Uses labels y
- Maximize class separation
- Pulls different classes apart
- For classification

Key Constraint:

Max dimensions = number of classes − 1. If you have 3 classes, LDA gives at most 2 dimensions (one axis per class boundary).

LDA — Mathematics & Assumptions

Optimize scatter ratio to maximize class separability.

Objective: Maximize J(w) = |S_B| / |S_W|

- S_B = Between-class scatter (covariance of class means)
- S_W = Within-class scatter (sum of within-class covariances)

Assumptions

- Gaussian classes
- Equal covariance matrices
- Linearly separable classes

When Violated

Quadratic DA (QDA) allows different covariances. Logistic regression needs no distributional assumptions. But LDA is still useful for preprocessing.

LDA maximizes: (class centers far apart) / (points within each class close together). Higher ratio = better class separation.

LDA — Code

```python
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.preprocessing import StandardScaler

# Scale first
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Fit LDA with labels
lda = LinearDiscriminantAnalysis(n_components=2)
X_lda = lda.fit_transform(X_scaled, y)  # Needs y!

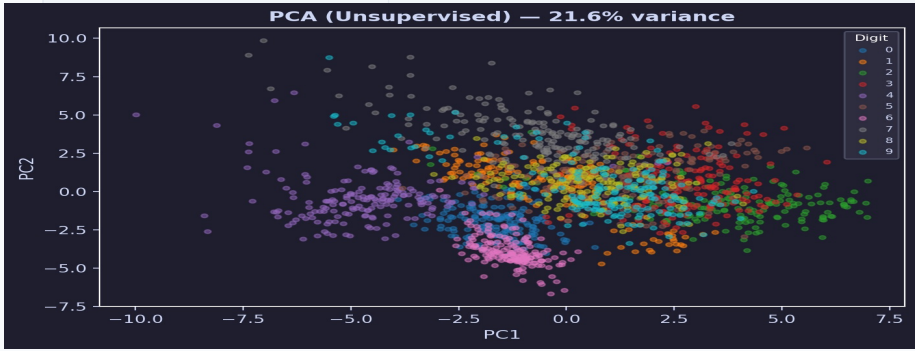
# Transform new data
X_new_lda = lda.transform(X_new)

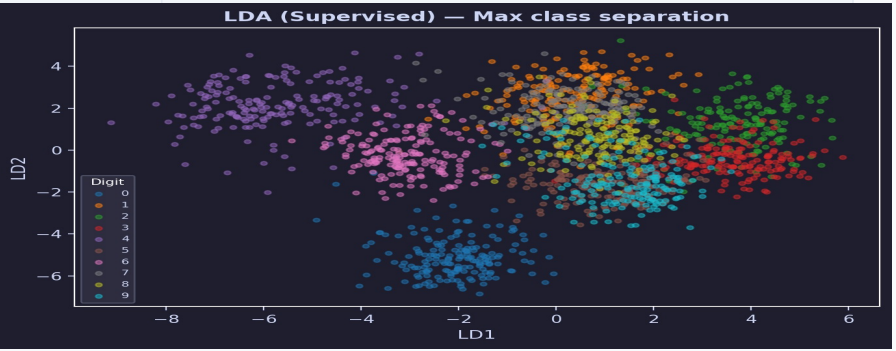
# Can use LDA for classification too
pred = lda.predict(X_new)
print(f"Score: {lda.score(X_scaled, y):.3f}")
print(f"Proba: {lda.predict_proba(X_scaled)}")  # soft
```

- fit_transform(X, y): must pass labels y for supervised learning
- n_components ≤ min(n_features, n_classes − 1): LDA limits dimensions to class separability
- LDA outputs both transformed data AND classification probabilities

PCA vs LDA — Side by Side

	PCA	LDA
Supervision	Unsupervised (ignores y)	Supervised (uses y)
Goal	Max variance	Max class separation
Dimensions	Any (min to p)	At most C−1
When to use	Any task, preprocessing, viz	Classification, known classes





t-SNE & UMAP — Nonlinear DR

PCA is linear — it can't capture curved or clustered structure. t-SNE and UMAP project to 2D/3D preserving neighborhoods.

t-SNE

Preserves LOCAL structure only. Beautiful 2D clusters but distances between clusters are meaningless. Slow $O(n^2)$, stochastic, no transform() for new data. Visualization ONLY — never use as ML features.

UMAP

Preserves LOCAL + GLOBAL structure. 10-100x faster than t-SNE, deterministic, has transform() for new data. Can feed into ML pipelines. Recommended default for most use cases.

How t-SNE Works — Algorithm Steps

Step 1: Pairwise Similarities in High-D

For each point, compute conditional probability p(j|i) using Gaussian kernel. Nearby points → high probability. Perplexity controls effective neighborhood size (bandwidth of Gaussian).

Step 2: Random 2D Layout → Similarities in Low-D

Initialize random 2D positions. Compute q(j|i) using Student's t-distribution (1 degree of freedom). Heavy tails of t-dist prevent 'crowding problem' — moderately distant points can spread out.

Step 3: Minimize KL Divergence

Gradient descent minimizes KL(P||Q). Forces low-D similarities to match high-D similarities. Local structure is preserved; global distances may distort.

Key insight: t-SNE is stochastic and $O(n^2)$. No transform() — must refit for new data. For visualization only.

How UMAP Works — Algorithm Steps

Step 1: Build Fuzzy k-NN Graph

For each point, find k nearest neighbors. Build weighted graph where edge weights decrease with distance. Local distances are normalized per-point (adapts to local density).

Step 2: Topological Representation

Interprets the graph as a fuzzy topological structure (simplicial set). Union of local neighborhoods forms a global picture. Based on Riemannian geometry — each point has its own notion of distance.

Step 3: Optimize Low-D Layout

Find 2D layout that preserves the same topological structure. Minimize cross-entropy between high-D and low-D fuzzy graphs. Attractive forces for connected points, repulsive for non-connected.

Key insight: UMAP is O(n·log n), deterministic, and supports transform(). Preserves both local AND global structure better than t-SNE.

t-SNE & UMAP in Practice

```python
from sklearn.manifold import TSNE
import umap  # pip install umap-learn

# --- t-SNE: visualization (2D) ---
tsne = TSNE(n_components=2, perplexity=30, random_state=42)
X_tsne = tsne.fit_transform(X_scaled)

plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=y, cmap='viridis', alpha=0.6)
plt.title('t-SNE'); plt.show()

# --- UMAP: visualization + pipeline-ready ---
reducer = umap.UMAP(n_neighbors=15, min_dist=0.1, random_state=42)
X_umap = reducer.fit_transform(X_scaled)

plt.scatter(X_umap[:, 0], X_umap[:, 1], c=y, cmap='viridis', alpha=0.6)
plt.title('UMAP'); plt.show()

# UMAP can transform new data (t-SNE cannot!)
X_new_umap = reducer.transform(X_new_scaled)
```

t-SNE & UMAP — Hyperparameter Effects

Results change dramatically with different settings. Always experiment before drawing conclusions.

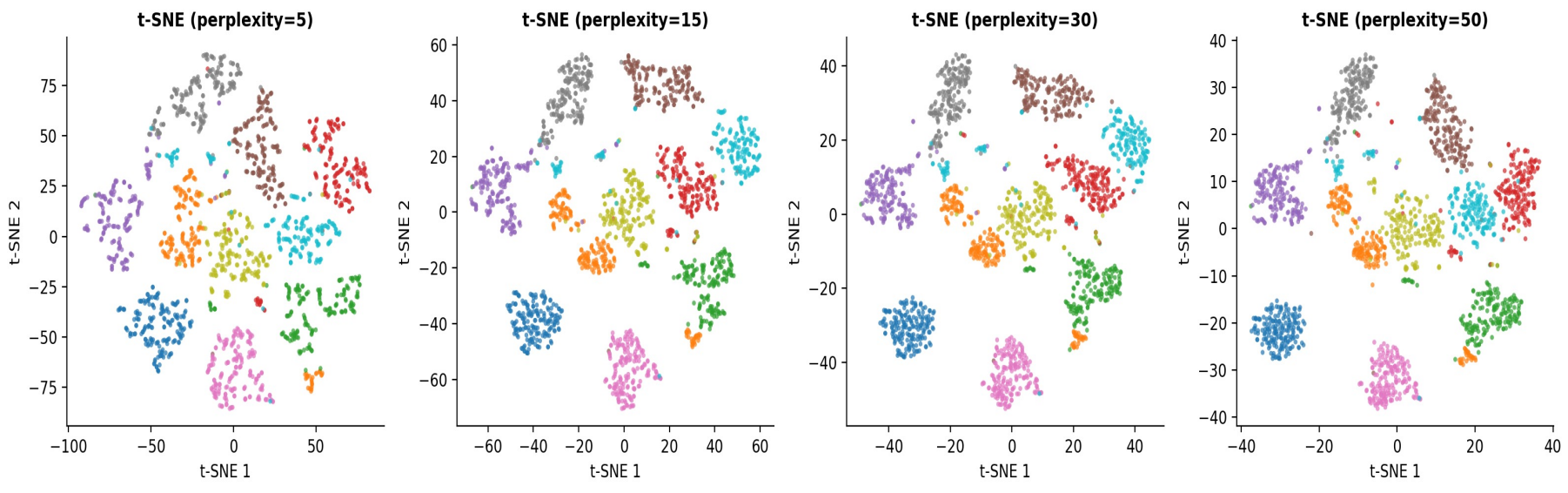
t-SNE: perplexity

Low (5–10): tight local clusters, noisy. Medium (30): balanced — good default. High (50–100): smoother, more global. Too high: everything merges. Always try 3+ values.

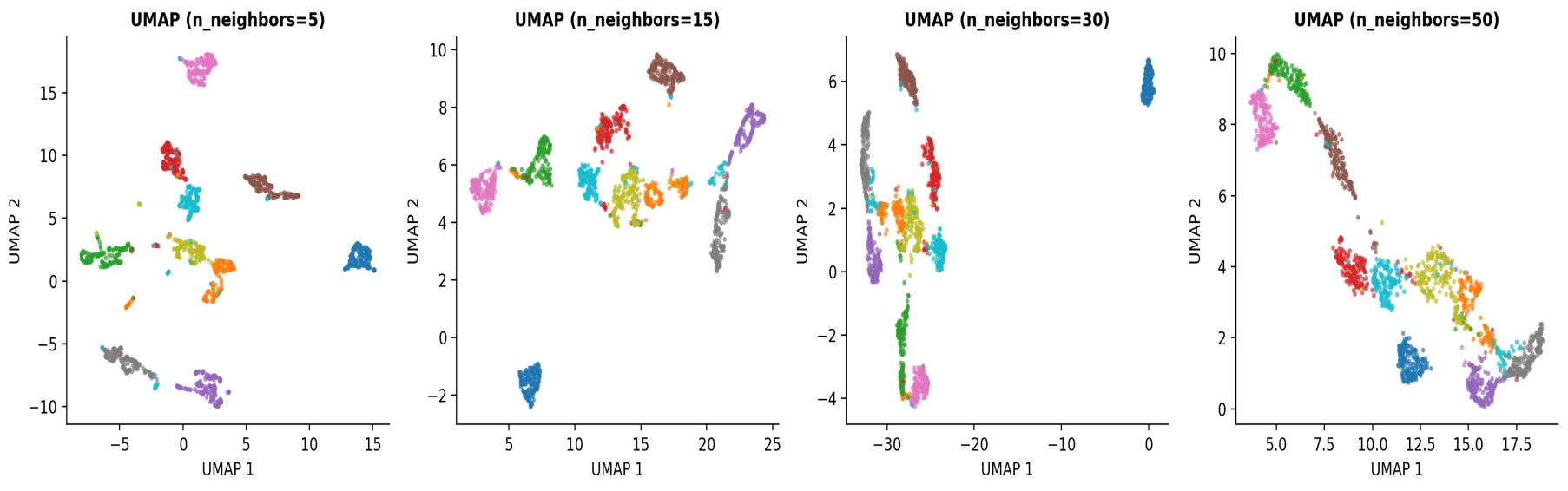
UMAP: n_neighbors & min_dist

n_neighbors (5–50): low = local detail, high = global view. min_dist (0.0–1.0): low = tight clusters, high = spread out. Default (15, 0.1) works well for most datasets.

t-SNE — Perplexity Effect (Digits Dataset)



UMAP — n_neighbors Effect (Digits Dataset)



t-SNE & UMAP — Interpretation Pitfalls

Cluster Sizes Are Meaningless

t-SNE/UMAP can shrink or expand clusters arbitrarily. Don't compare cluster sizes across groups.

Distances Between Clusters Are Unreliable

Gap between clusters in 2D does not reflect true distance in high dimensions. Only local neighborhood is preserved.

Run Multiple Times

t-SNE is stochastic — different random seeds give different layouts. Run 3+ times and check consistency.

Scale Your Data First

Like PCA, t-SNE/UMAP are distance-based. Always StandardScaler() before applying. Unscaled features dominate.

Don't Use t-SNE Features for ML

t-SNE output is for visualization only — non-deterministic and non-parametric. Use PCA or UMAP for ML pipelines.

PCA vs t-SNE vs UMAP — Side by Side

```python
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
import umap
import matplotlib.pyplot as plt

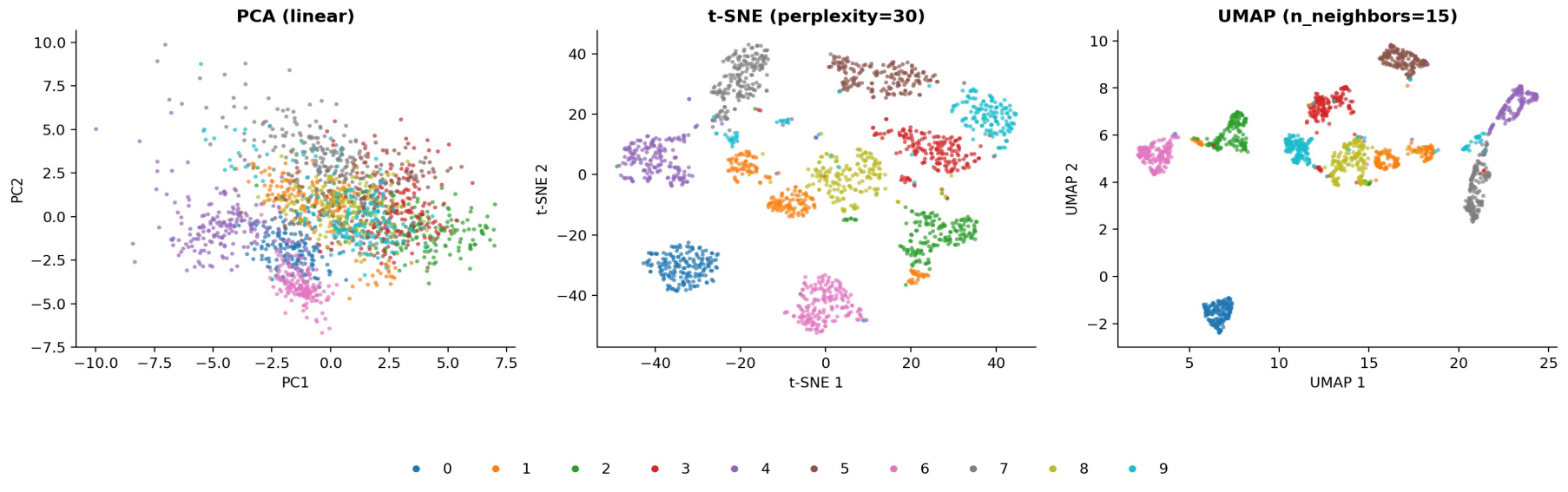
# Same data, three different views
fig, axes = plt.subplots(1, 3, figsize=(15, 5))

# PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)
axes[0].scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap='viridis', s=5)
axes[0].set_title('PCA (linear)')

# t-SNE
tsne = TSNE(n_components=2, perplexity=30, random_state=42)
X_tsne = tsne.fit_transform(X_scaled)
axes[1].scatter(X_tsne[:, 0], X_tsne[:, 1], c=y, cmap='viridis', s=5)
axes[1].set_title('t-SNE (local structure)')

# UMAP
reducer = umap.UMAP(n_neighbors=15, min_dist=0.1, random_state=42)
X_umap = reducer.fit_transform(X_scaled)
axes[2].scatter(X_umap[:, 0], X_umap[:, 1], c=y, cmap='viridis', s=5)
axes[2].set_title('UMAP (local + global)')
plt.tight_layout(); plt.show()
```

PCA vs t-SNE vs UMAP — Comparison (Digits Dataset)



Feature Selection — Practical Methods

Select the most relevant original features instead of creating new ones. Simpler, faster, and more interpretable than PCA.

Filter Methods

Fast, model-agnostic. Score features independently. Use when: many features (1000+), quick preprocessing, or as a first pass before wrapper methods.

- VarianceThreshold: drop near-zero variance
- SelectKBest + mutual_info: top k informative
- Correlation filter: drop redundant features

Wrapper Methods

Slower but more accurate. Uses ML model to evaluate subsets. Use when: fewer features (<100), final model tuning, or when feature interactions matter.

- RFE: recursively remove weakest features
- Forward/Backward selection: add/remove one at a time
- L1 (Lasso): automatic selection via regularization

Main libraries: sklearn.feature_selection, sklearn.linear_model (Lasso)

Feature Selection in Practice

```python
from sklearn.feature_selection import (VarianceThreshold, SelectKBest,
                                       mutual_info_classif, RFE)
from sklearn.linear_model import Lasso, LogisticRegression

# 1. Variance Threshold — drop near-constant features
selector = VarianceThreshold(threshold=0.01)
X_var = selector.fit_transform(X)
print(f"Kept {X_var.shape[1]}/{X.shape[1]} features")

# 2. SelectKBest — top k by mutual information
selector = SelectKBest(mutual_info_classif, k=20)
X_top = selector.fit_transform(X, y)
best_features = X.columns[selector.get_support()]

# 3. RFE — Recursive Feature Elimination
rfe = RFE(LogisticRegression(max_iter=1000), n_features_to_select=10)
rfe.fit(X, y)
print(f"Selected: {X.columns[rfe.support_].tolist()}")

# 4. L1 Regularization (Lasso) — automatic selection
lasso = Lasso(alpha=0.01).fit(X, y)
selected = X.columns[lasso.coef_ != 0]
print(f"Lasso selected {len(selected)} features")
```

SelectFromModel & Tree-Based Importance

```python
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import RandomForestClassifier
import numpy as np

# 1. Train a tree model and extract importances
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)

# Feature importance ranking
importances = rf.feature_importances_
indices = np.argsort(importances)[::-1]
for i in range(10):
    print(f"{X.columns[indices[i]]}: {importances[indices[i]]:.4f}")

# 2. SelectFromModel — auto-select above threshold
selector = SelectFromModel(rf, threshold='median')
X_selected = selector.fit_transform(X_train, y_train)
print(f"Selected {X_selected.shape[1]}/{X_train.shape[1]} features")

# 3. Use in Pipeline
from sklearn.pipeline import Pipeline
pipe = Pipeline([
    ('select', SelectFromModel(RandomForestClassifier())),
    ('clf', LogisticRegression(max_iter=1000))
])
pipe.fit(X_train, y_train)
```

Putting It All Together

End-to-end examples and best practices

Algorithm Comparison Table

Property	PCA	LDA	t-SNE	UMAP
Linear?	Yes	Yes	No	No
Supervised?	No	Yes	No	No
Global structure	Preserved	Preserved	Weak	Strong
Local structure	Weak	Weak	Excellent	Excellent
Speed	Fast	Fast	Slow	Fast
For ML pipeline?	Yes	Yes	No	Yes
Interpretable?	Loadings	Class axes	No	Somewhat
Key params	n_components	n_components	perplexity	n_neighbors

How to Choose an Algorithm

Decision guide:

Need features for ML?	Yes → PCA or LDA
Have class labels?	Yes → LDA. No → PCA
Just visualizing?	Yes → t-SNE or UMAP
Need global structure?	Yes → UMAP. No → t-SNE
Speed critical?	Yes → UMAP or PCA. No → t-SNE OK
Need to transform new data?	Yes → UMAP/PCA. No → t-SNE fine

Full Example 1/3: Data Loading & PCA

```python
from sklearn.datasets import load_digits
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

# Load 8×8 digit images (1797 samples, 64 features)
X, y = load_digits(return_X_y=True)
print(f"Shape: {X.shape}")  # (1797, 64)

# Scale
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# PCA with 95% variance
pca = PCA(n_components=0.95)
X_pca = pca.fit_transform(X_scaled)

print(f"Original dims: {X.shape[1]}")
print(f"Reduced to: {pca.n_components_} components")
print(f"Variance explained: {pca.explained_variance_ratio_.sum():.1%}")

# Visualize scree plot
import matplotlib.pyplot as plt
cumsum = np.cumsum(pca.explained_variance_ratio_)
plt.plot(cumsum)
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.axhline(y=0.95, color='r', linestyle='--')
plt.title('PCA Scree Plot'); plt.show()
```

Full Example 2/3: LDA + Side-by-Side Comparison

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# LDA (supervised)
lda = LinearDiscriminantAnalysis(n_components=2)
X_lda = lda.fit_transform(X_scaled, y)  # Uses labels!

# Visualize both
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# PCA projection
scatter1 = axes[0].scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap='viridis', alpha=0.6)
axes[0].set_title(f'PCA (var={pca.explained_variance_ratio_[:2].sum():.1%})')
axes[0].set_xlabel('PC1'); axes[0].set_ylabel('PC2')

# LDA projection
scatter2 = axes[1].scatter(X_lda[:, 0], X_lda[:, 1], c=y, cmap='viridis', alpha=0.6)
axes[1].set_title(f'LDA (supervised)')
axes[1].set_xlabel('LD1'); axes[1].set_ylabel('LD2')

plt.colorbar(scatter2, ax=axes[1], label='Digit')
plt.tight_layout(); plt.show()

# Notice: LDA clusters are MORE separated (uses labels)
```

Full Example 3/3: DR in ML Pipeline

Complete end-to-end pipeline: scale → DR → classifier → evaluate.

```python
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC

# Build pipeline
pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('dr', PCA(n_components=10)),  # or LDA or UMAP
    ('clf', SVC(kernel='rbf'))
])

# Grid search over DR n_components and SVM C
params = {
    'dr__n_components': [2, 5, 10, 20, 30],
    'clf__C': [0.1, 1, 10, 100]
}

gs = GridSearchCV(pipe, params, cv=5, n_jobs=-1)
gs.fit(X_train, y_train)

print(f"Best params: {gs.best_params_}")
print(f"Best CV score: {gs.best_score_:.3f}")
test_score = gs.score(X_test, y_test)
print(f"Test score: {test_score:.3f}")
```

Common Mistakes & Tips

Forgetting to Scale

DR is distance-based. Unscaled features dominate. Always StandardScaler first.

Using t-SNE Features for ML

t-SNE output is for viz only; non-deterministic & unstable. Use PCA/UMAP for pipelines.

Ignoring Explained Variance

Don't guess n_components. Check cumulative variance or use scree plots.

Too Few or Too Many Components

Too few = lose important info. Too many = no reduction benefit. Use 90-95% variance rule.

Not Running t-SNE Multiple Times

t-SNE is random. Always run 3+ times and compare before trusting results.

Key Takeaways

PCA

Linear, unsupervised, fast. Solvable via SVD. Probabilistic extensions: PPCA, Factor Analysis.

LDA

Supervised DR. Maximizes class separation. Use when labels available.

t-SNE

Nonlinear, beautiful 2D plots. Visualization only; don't use for ML features.

UMAP

Modern, fast, global + local structure. Visualization & preprocessing. Recommended default.