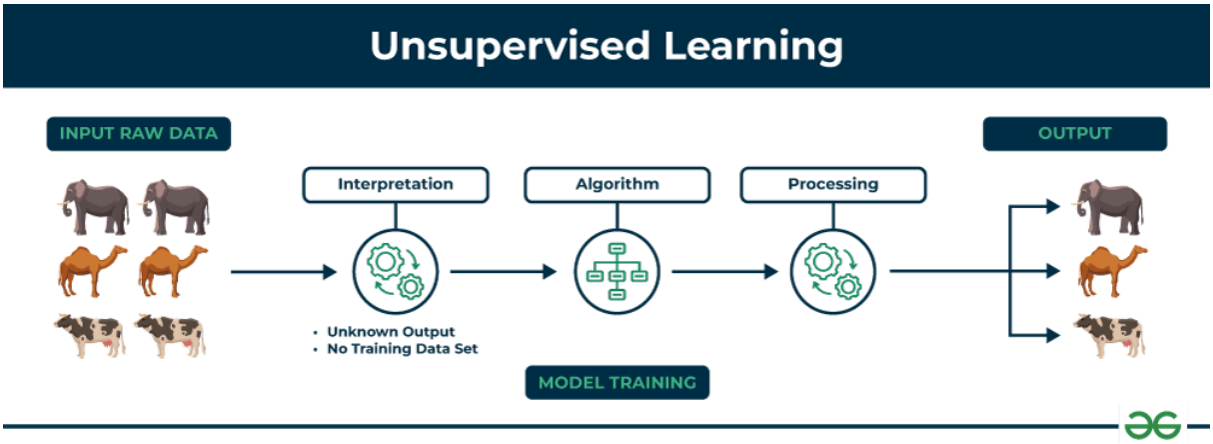


MOOC 4: Unsupervised Machine Learning

🔗 View Original Document

https://www.notion.so/MOOC-4-Unsupervised-Machine-Learning-28ff95c1eb34805385c1f6ac02854c05?source=copy_link

Module 1: Introduction to Unsupervised Learning & K-Means

Giới thiệu Unsupervised Learning

Unsupervised Learning là các thuật toán học máy được sử dụng khi **không có biến mục tiêu (label)** để dự đoán. Mục đích chính là tìm ra **cấu trúc ẩn** trong dữ liệu.

Hai loại Unsupervised Learning chính

Loại	Mục đích	Ví dụ	Thuật toán
Clustering	Nhóm các observations tương tự nhau	Phân khúc khách hàng	K-Means, Hierarchical, DBSCAN

Loại	Mục đích	Ví dụ	Thuật toán
Dimensionality Reduction	Giảm số lượng features mà vẫn giữ được thông tin	Giảm kích thước ảnh, khử nhiễu	PCA, NMF, t-SNE

The Curse of Dimensionality

Vấn đề: Càng nhiều features thì performance càng **tệ đi** thay vì tốt hơn!

Tại sao?

- Một số features là **spurious correlations** (tương quan giả)
- Quá nhiều features tạo ra **nhiều nhiễu hơn signal**
- Thuật toán khó phân biệt features có ý nghĩa
- Cần **exponentially nhiều training examples** hơn
- **Computational cost** tăng cao
- Tăng khả năng xuất hiện outliers

👉 **Giải pháp:** Dimensionality Reduction!

Ứng dụng Clustering trong thực tế

1. Anomaly Detection (Phát hiện bất thường)

Ví dụ: Phát hiện giao dịch gian lận

- Các giao dịch gian lận thường tạo thành clusters nhỏ với patterns bất thường
- High volume attempts, small amounts, new merchants

2. Customer Segmentation (Phân khúc khách hàng)

- Phân khúc theo RFM: Recency, Frequency, Monetary value
- Phân khúc theo demographics + engagement level
- Tối ưu hóa marketing campaigns

3. Improve Supervised Learning

- Train một model riêng cho mỗi cluster

- Cải thiện độ chính xác classification

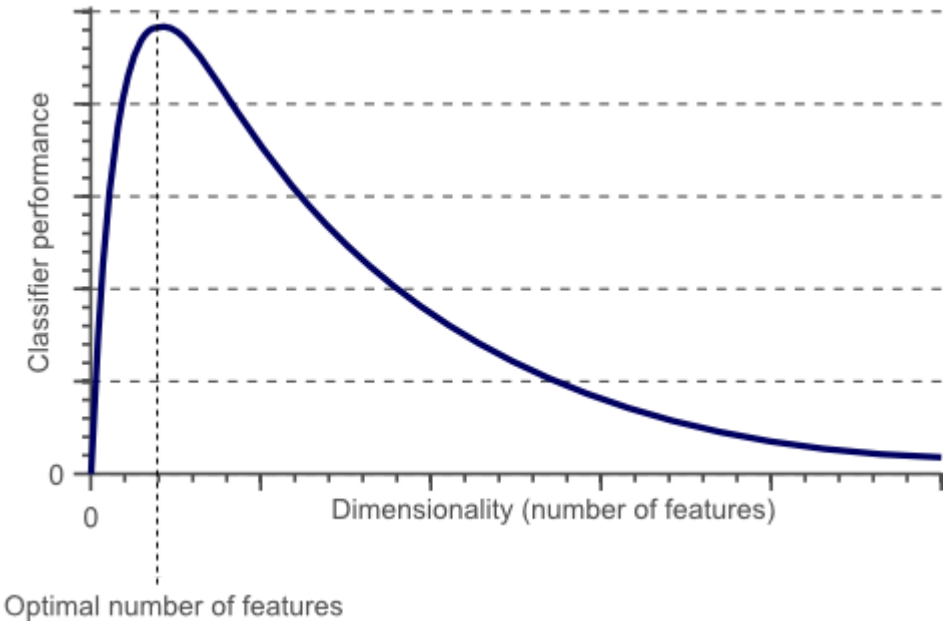
Ứng dụng Dimensionality Reduction

1. Image Compression

Chuyển high-resolution images → compressed images giữ lại thông tin quan trọng

2. Image/Video Tracking

Giảm noise → tăng tốc độ computational efficiency của detection algorithms



K-Means Clustering

K-Means là thuật toán **iterative** nhóm các observations tương tự nhau thành K clusters.

Cách hoạt động của K-Means

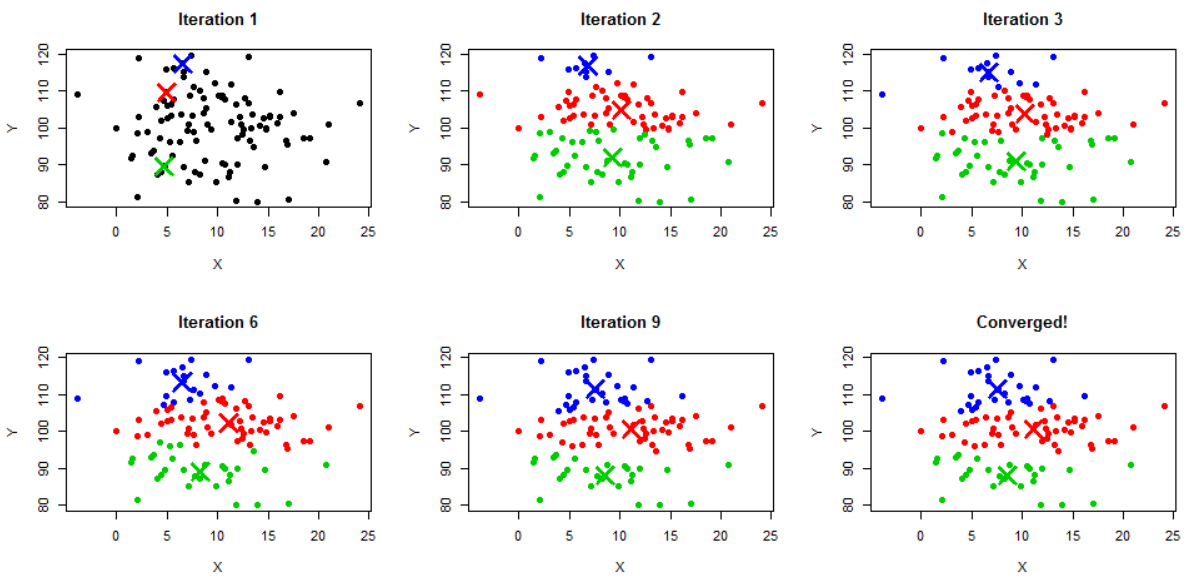
Bước 1: Chọn K centroids ngẫu nhiên

Bước 2: Tính khoảng cách từ mỗi điểm đến các centroids

Bước 3: Gán mỗi điểm vào cluster của centroid gần nhất

Bước 4: Tính lại centroids mới = trung bình của các điểm trong cluster

Bước 5: Lặp lại Bước 2-4 cho đến khi **hội tụ** (clusters không thay đổi)



Công thức toán học

Mục tiêu: Minimize Within-Cluster Sum of Squares (WCSS)

$$\text{WCSS} = \sum_{i=1}^{K} \sum_{x \in C_i} ||x - \mu_i||^2$$

Trong đó:

- K = số clusters
- C_i = cluster thứ i
- μ_i = centroid của cluster C_i
- $||x - \mu_i||$ = khoảng cách Euclidean

Ưu & Nhược điểm

Ưu điểm:

- Dễ hiểu và implement
- Tính toán nhanh, hiệu quả với large datasets
- Scale tốt với số lượng lớn samples

Nhược điểm:

- **Sensitive** với initial centroids (K-Means++ giải quyết vấn đề này)
- Phải biết trước **K** (số clusters)
- Chỉ tạo ra **spherical clusters**
- Nhạy cảm với outliers

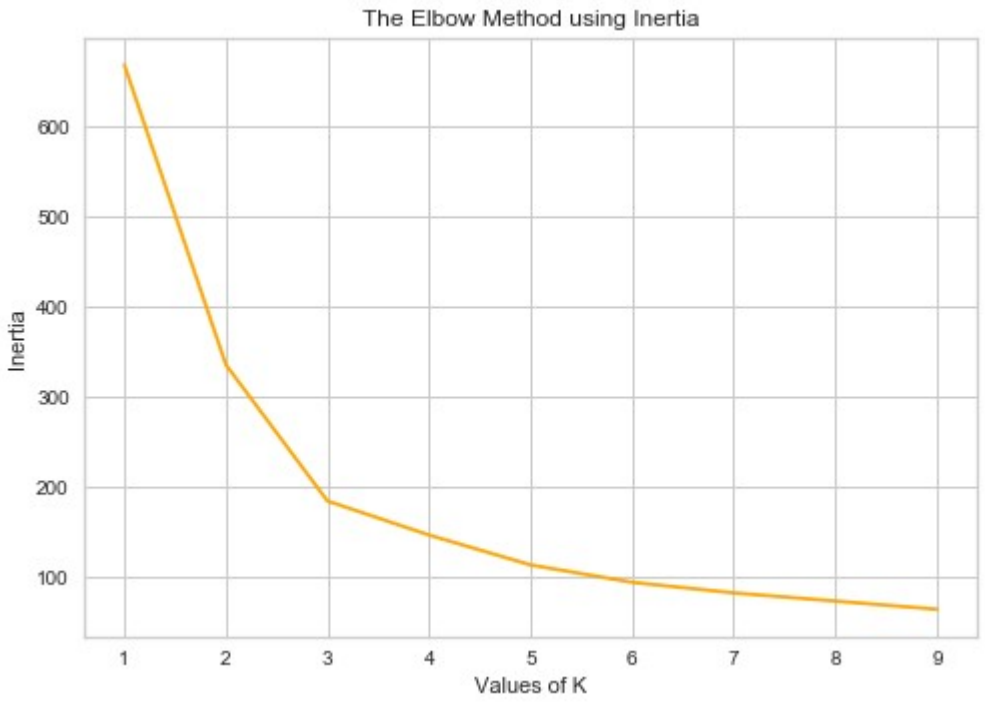
Model Selection: Chọn K tối ưu

Khi không biết trước số clusters, ta cần metrics để chọn K:

1. Inertia (Sum of Squared Distances)

$$\text{Inertia} = \sum_{i=1}^{n} ||x_i - \mu_{c_i}||^2$$

- **Giá trị nhỏ** = clusters chặt chẽ hơn
- **Nhược điểm:** Tăng khi thêm nhiều points vào cluster



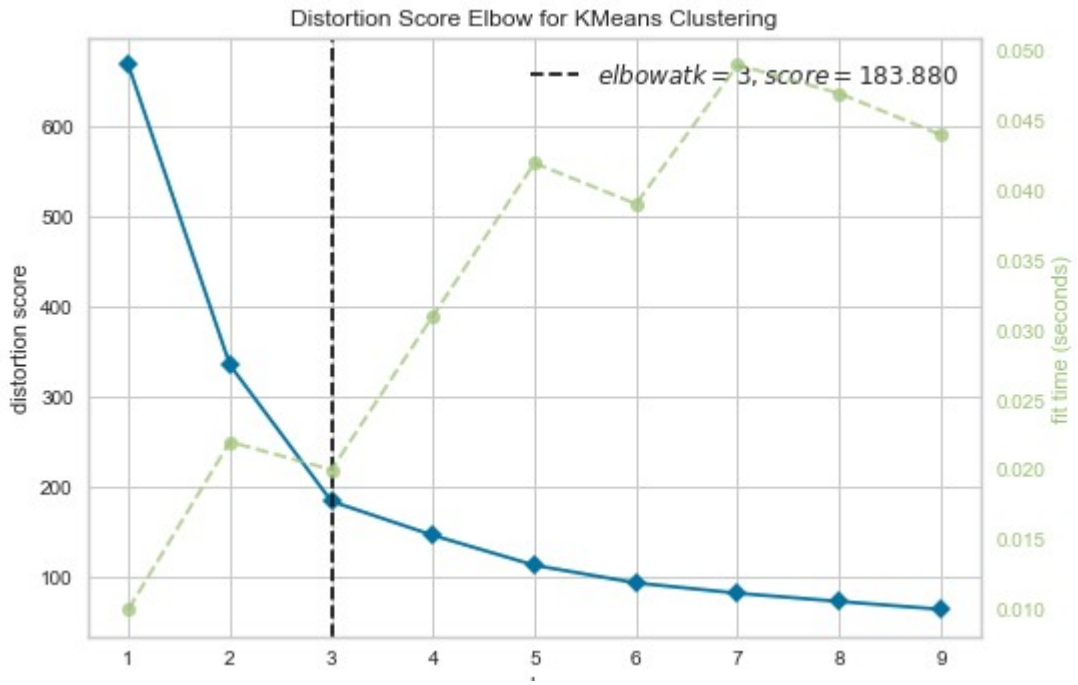
2. Distortion (Average Squared Distance)

$$\text{Distortion} = \frac{1}{n} \sum_{i=1}^{n} ||x_i - \mu_{c_i}||^2$$

- **Giá trị nhỏ** = clusters chặt chẽ hơn
- **Ưu điểm:** Không tăng khi thêm points gần centroid

Lựa chọn metric:

- **Distortion:** Khi quan tâm đến similarity của points trong cluster
- **Inertia:** Khi muốn clusters có số lượng points tương đương nhau



Code Implementation - K-Means

```
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
import numpy as np
import matplotlib.pyplot as plt

# Scale data (IMPORTANT!)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Train K-Means with K=3
```

```
kmeans = KMeans(
    n_clusters=3,
    init='k-means++',  # Smart initialization
    n_init=10,         # Run 10 times with different centroids
    max_iter=300,
    random_state=42
)

# Fit and predict
kmeans.fit(X_scaled)
labels = kmeans.predict(X_scaled)

# Get cluster centers
centers = kmeans.cluster_centers_

print(f"Inertia: {kmeans.inertia_:.2f}")
```

Finding Optimal K - Elbow Method

```
# Test different K values
K_range = range(2, 11)
inertias = []
distortions = []

for k in K_range:
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(X_scaled)

    inertias.append(kmeans.inertia_)
    distortions.append(kmeans.inertia_ / len(X_scaled))

# Plot Elbow Curve
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(K_range, inertias, 'bo-')
plt.xlabel('Number of Clusters (K)')
plt.ylabel('Inertia')
```

```
plt.title('Elbow Method - Inertia')
plt.grid(True, alpha=0.3)

plt.subplot(1, 2, 2)
plt.plot(K_range, distortions, 'ro-')
plt.xlabel('Number of Clusters (K)')
plt.ylabel('Distortion')
plt.title('Elbow Method - Distortion')
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

Module 2: Distance Metrics & Computational Hurdles

Distance Metrics là nền tảng của clustering algorithms. Lựa chọn metric phù hợp **cực kỳ quan trọng** cho kết quả clustering!

Các Distance Metrics phổ biến

1. Euclidean Distance (L2 Distance)

Distance metric phổ biến nhất - khoảng cách "thẳng" giữa 2 điểm

$$d(p, q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2} = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + ... + (p_n - q_n)^2}$$

Sử dụng khi:

- Coordinate-based measurements
- Data có scale tương đương nhau

Nhược điểm:

- Sensitive với curse of dimensionality
- Cần scale data trước khi sử dụng

---

2. Manhattan Distance (L1 / City Block)

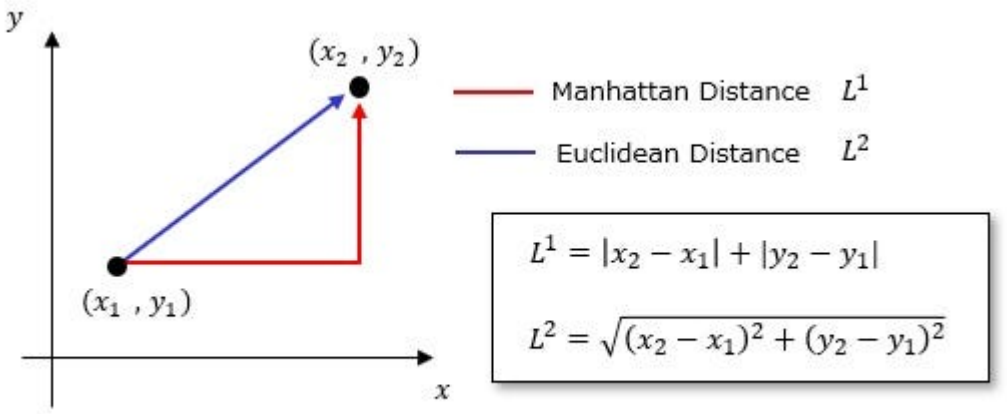
Tổng **absolute differences** - giống như đi trên lưới đường phố Manhattan

$$d(p,q) = \sum_{i=1}^{n} |p_i - q_i| = |p_1 - q_1| + |p_2 - q_2| + ... + |p_n - q_n|$$

Sử dụng khi:

- High dimensionality data
- Robust hơn với outliers

So sánh: Manhattan distance **luôn lớn hơn hoặc bằng** Euclidean distance



---

3. Cosine Distance

Đo **góc** giữa 2 vectors, không quan tâm magnitude

$$\text{cosine similarity} = \frac{A \cdot B}{||A|| \cdot ||B||} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \cdot \sqrt{\sum_{i=1}^{n} B_i^2}}$$

$$\text{cosine distance} = 1 - \text{cosine similarity}$$

Sử dụng khi:

- Text data (word occurrence)
- Location không quan trọng bằng direction
- Document similarity

Đặc điểm: Insensitive với scaling - 2 điểm trên cùng 1 đường thẳng qua origin có distance = 0

Euclidean vs Cosine:

- **Euclidean:** Coordinate-based, sensitive với dimensionality
- **Cosine:** Direction-based, tốt cho text data

---

4. Jaccard Distance

Sử dụng cho **sets** (word occurrence, binary features)

Ví dụ:

- Sentence A: "I like chocolate ice cream"
  - Set A = {I, like, chocolate, ice, cream}
- Sentence B: "Do I want chocolate cream or vanilla cream?"
  - Set B = {Do, I, want, chocolate, cream, or, vanilla}

$$\text{Jaccard Similarity} = \frac{|A \cap B|}{|A \cup B|} = \frac{\text{len(shared)}}{\text{len(unique)}}$$

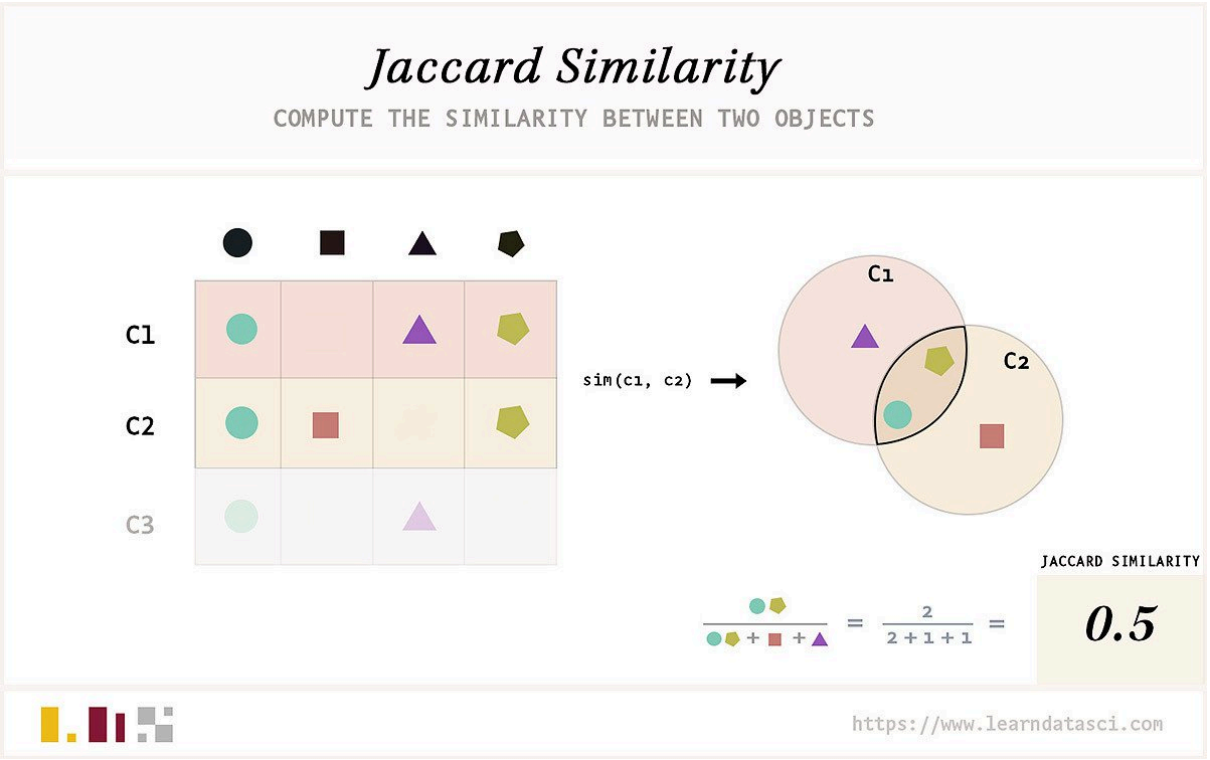
$$\text{Jaccard Distance} = 1 - \text{Jaccard Similarity} = 1 - \frac{3}{9} = \frac{6}{9}$$

Shared words: {I, chocolate, cream} = 3 words

Unique words: {I, like, chocolate, ice, cream, Do, want, or, vanilla} = 9 words

Sử dụng khi:

- Text documents
- Grouping similar topics



So sánh Distance Metrics

Metric	Use Case	Pros	Cons
Euclidean	Coordinate data, low dimensions	Intuitive, fast	Sensitive to scale & dimensionality
Manhattan	High dimensions	Better with curse of dimensionality	Less intuitive
Cosine	Text, directions matter	Scale-invariant	Ignores magnitude
Jaccard	Sets, binary features	Simple for text	Limited to sets

Code Implementation - Distance Metrics

```
from sklearn.metrics.pairwise import euclidean_distances, manhattan_dista
nces, cosine_distances
import numpy as np

# Sample data
X = np.array([[1, 2, 3],
              [4, 5, 6],
```

```
              [7, 8, 9]])

# Euclidean Distance
euclidean_dist = euclidean_distances(X)
print("Euclidean Distance:\n", euclidean_dist)

# Manhattan Distance
manhattan_dist = manhattan_distances(X)
print("\nManhattan Distance:\n", manhattan_dist)

# Cosine Distance
cosine_dist = cosine_distances(X)
print("\nCosine Distance:\n", cosine_dist)

# Jaccard Distance (for binary data)
from sklearn.metrics import jaccard_score

# Example: binary vectors
A = np.array([1, 1, 0, 1, 0])
B = np.array([1, 0, 0, 1, 1])

jaccard_sim = jaccard_score(A, B)
jaccard_dist = 1 - jaccard_sim
print(f"\nJaccard Similarity: {jaccard_sim:.3f}")
print(f"Jaccard Distance: {jaccard_dist:.3f}")
```

Module 3: Hierarchical Agglomerative Clustering

Hierarchical Clustering liên tục **merge** và **split** clusters cho đến khi hội tụ, tạo ra một **hierarchy tree (dendrogram)**.

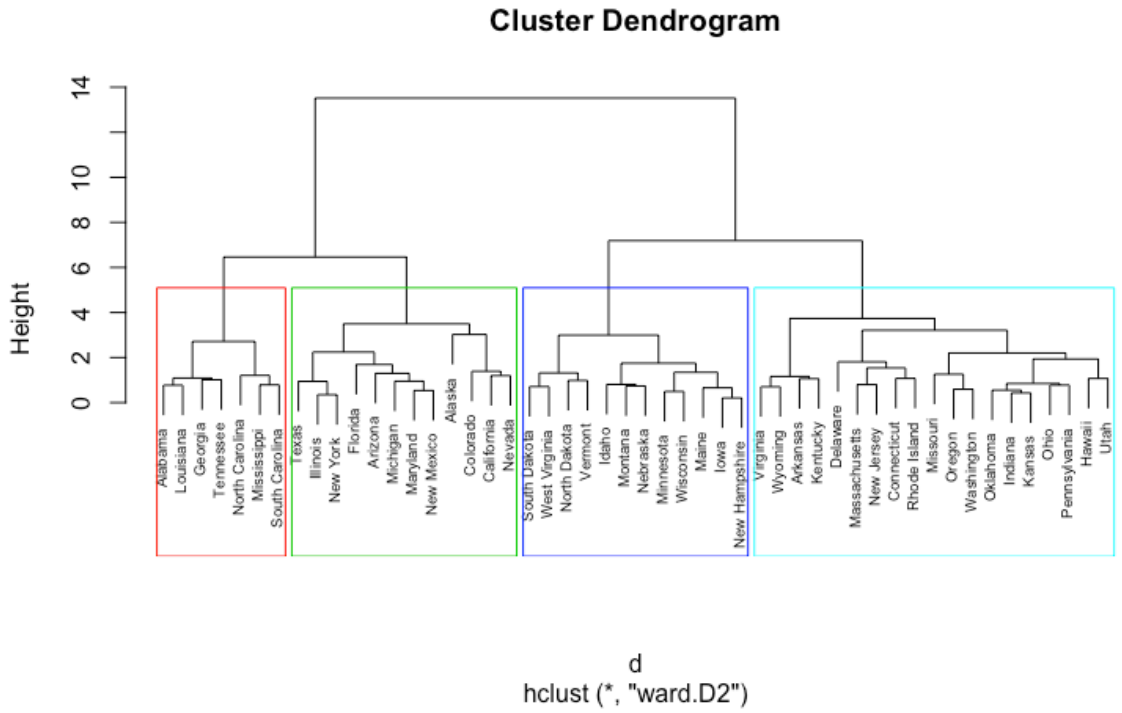
Nguyên lý hoạt động

Bước 1: Mỗi điểm là 1 cluster riêng

Bước 2: Tìm 2 clusters **gần nhất** và merge lại

Bước 3: Lặp lại cho đến khi:

- Chỉ còn 1 cluster (stopping criterion)
- Hoặc đạt số clusters mong muốn



Linkage Methods (Cách đo distance giữa clusters)

Linkage method quyết định cách tính khoảng cách giữa 2 clusters!

1. Single Linkage (Minimum)

$$d(C_i, C_j) = \min_{x \in C_i, y \in C_j} d(x, y)$$

- Lấy **khoảng cách nhỏ nhất** giữa bất kỳ 2 điểm nào của 2 clusters
- **Pros:** Clear separation giữa clusters
- **Cons:** Không tốt nếu có noise/overlap

2. Complete Linkage (Maximum)

$$d(C_i, C_j) = \max_{x \in C_i, y \in C_j} d(x, y)$$

- Lấy **khoảng cách lớn nhất** giữa bất kỳ 2 điểm nào
- **Pros:** Tốt với noisy data
- **Cons:** Có thể break apart large clusters

3. Average Linkage

$$d(C_i, C_j) = \frac{1}{|C_i| \cdot |C_j|} \sum_{x \in C_i} \sum_{y \in C_j} d(x, y)$$

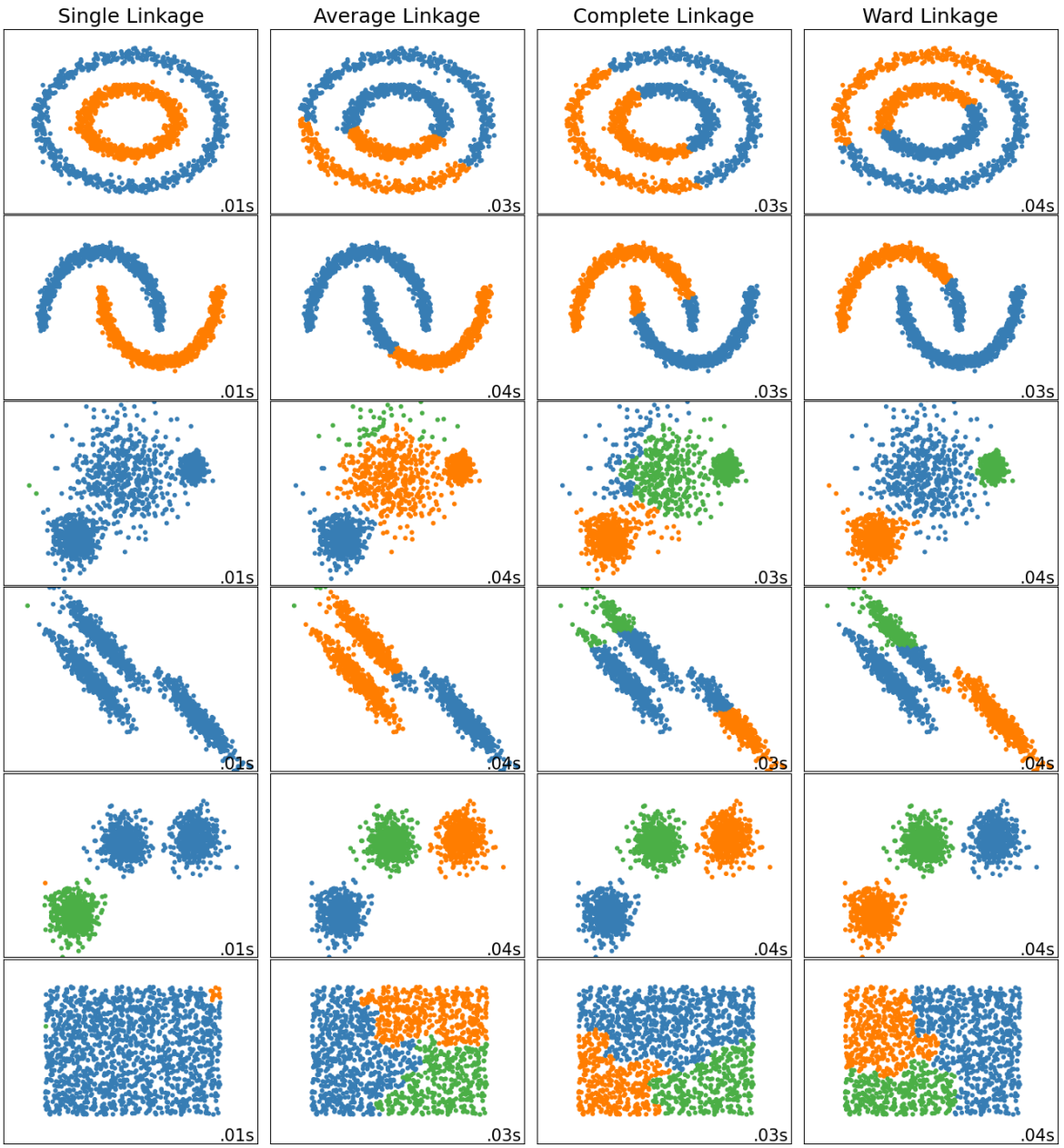
- Lấy **trung bình** khoảng cách giữa tất cả các cặp điểm
- **Pros:** Balance giữa single và complete
- **Cons:** Có thể break apart large clusters

4. Ward Linkage (Most Common)

Merge clusters minimize **inertia increase**

$$d(C_i, C_j) = \sum_{x \in C_i \cup C_j} ||x - \mu_{i \cup j}||^2 - \left(\sum_{x \in C_i} ||x - \mu_i||^2 + \sum_{x \in C_j} ||x - \mu_j||^2 \right)$$

- **Pros:** Tạo ra balanced clusters
- **Cons:** Giống average linkage



Code Implementation - Hierarchical Clustering

```
from sklearn.cluster import AgglomerativeClustering
from scipy.cluster.hierarchy import dendrogram, linkage
import matplotlib.pyplot as plt

# Create Agglomerative Clustering
agg = AgglomerativeClustering(
    n_clusters=3,
    affinity='euclidean',  # Distance metric
```

```
    linkage='ward'         # Linkage method
)

# Fit and predict
labels = agg.fit_predict(X_scaled)

print(f"Cluster labels: {labels}")
```

Visualize Dendrogram

```
# Create linkage matrix
Z = linkage(X_scaled, method='ward')

# Plot dendrogram
plt.figure(figsize=(12, 6))
dendrogram(
    Z,
    truncate_mode='lastp',
    p=12,
    leaf_font_size=12,
    show_contracted=True
)
plt.title('Hierarchical Clustering Dendrogram')
plt.xlabel('Sample Index or (Cluster Size)')
plt.ylabel('Distance')
plt.axhline(y=10, color='r', linestyle='--', label='Cut threshold')
plt.legend()
plt.show()
```

Module 4: Dimensionality Reduction với PCA

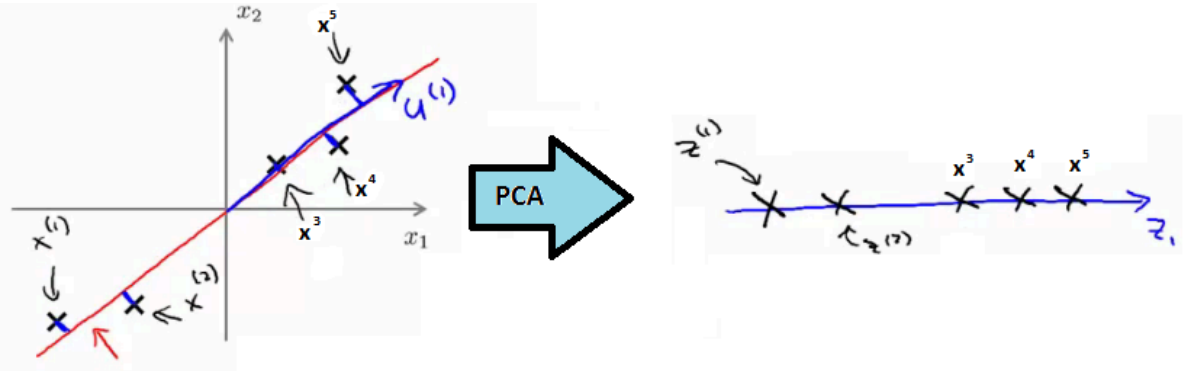
Principal Component Analysis (PCA) là kỹ thuật dimensionality reduction phổ biến nhất, tạo ra **principal components** bằng **linear transformations**.

PCA là gì?

PCA tìm các **directions** (principal components) trong data space mà có **variance lớn nhất**.

Ý tưởng:

1. PC1 (Principal Component 1) = direction có **variance lớn nhất**
2. PC2 = direction **orthogonal** với PC1, variance lớn thứ 2
3. PC3, PC4, ... tương tự



Toán học đằng sau PCA

Covariance Matrix

$$\text{Cov}(X) = \frac{1}{n-1} X^T X$$

Eigenvalue Decomposition

$$\text{Cov}(X) = V \Lambda V^T$$

Trong đó:

- V = **Eigenvectors** (principal components)
- Λ = **Eigenvalues** (variance explained)

Singular Value Decomposition (SVD)

$$X = U \Sigma V^T$$

- U = Left singular vectors
- Σ = Diagonal matrix (singular values)

- V = **Principal components** (Right singular vectors)

Explained Variance Ratio:

$$\text{Variance Ratio}_i = \frac{\lambda_i}{\sum_{j=1}^{n} \lambda_j}$$

Eigenvalue càng **lớn** → Principal component càng **quan trọng**!

Quy trình PCA

Bước 1: Scale data (StandardScaler)

$$x_{\text{scaled}} = \frac{x - \mu}{\sigma}$$

Cực kỳ quan trọng vì PCA dựa trên distance!

Bước 2: Compute covariance matrix

Bước 3: Find eigenvectors và eigenvalues (SVD)

Bước 4: Sort eigenvalues giảm dần

Bước 5: Chọn top K eigenvectors

Bước 6: Transform data

$$X_{\text{reduced}} = X \cdot V_k$$

Chọn số Components

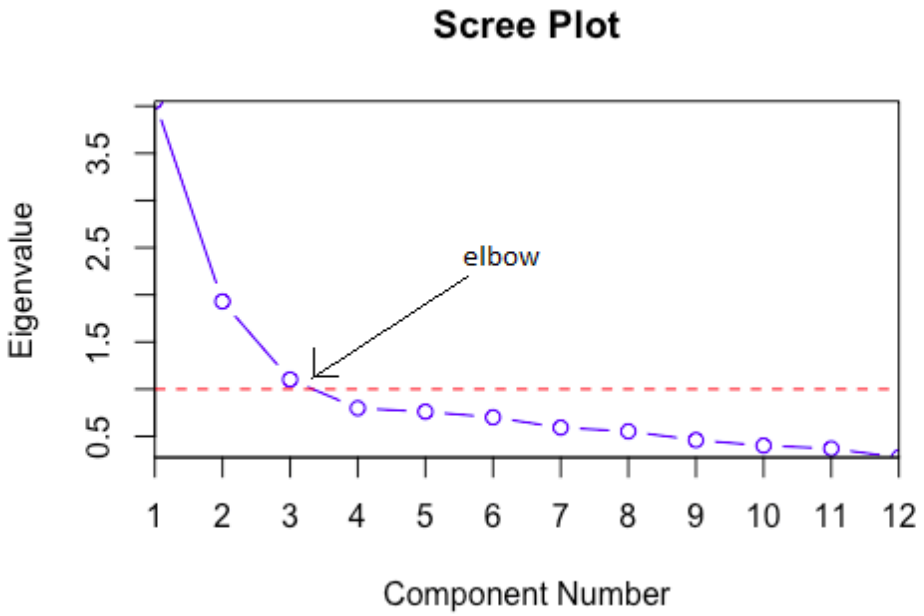
Method 1: Explained Variance Threshold

Chọn số components sao cho giải thích được **≥ 95%** variance

```
cumsum_variance = np.cumsum(pca.explained_variance_ratio_)
n_components = np.argmax(cumsum_variance >= 0.95) + 1
```

Method 2: Elbow Method

Vẽ biểu đồ variance vs number of components, tìm "elbow"



Code Implementation - PCA

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import numpy as np
import matplotlib.pyplot as plt

# Step 1: Scale data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Step 2: Apply PCA
pca = PCA(n_components=3)  # Reduce to 3 dimensions
X_pca = pca.fit_transform(X_scaled)

# Results
print(f"Original shape: {X_scaled.shape}")
print(f"Reduced shape: {X_pca.shape}")
print(f"\nExplained Variance Ratio: {pca.explained_variance_ratio_}")
```

```
print(f"Total Variance Explained: {pca.explained_variance_ratio_.sum():.
2%}")
```

Finding Optimal Number of Components

```
# Test different number of components
pca_full = PCA()
pca_full.fit(X_scaled)

# Plot explained variance
plt.figure(figsize=(12, 5))

# Subplot 1: Individual variance
plt.subplot(1, 2, 1)
plt.bar(range(1, len(pca_full.explained_variance_ratio_)+1),
        pca_full.explained_variance_ratio_)
plt.xlabel('Principal Component')
plt.ylabel('Variance Ratio')
plt.title('Explained Variance by Component')
plt.grid(True, alpha=0.3)

# Subplot 2: Cumulative variance
plt.subplot(1, 2, 2)
cumsum = np.cumsum(pca_full.explained_variance_ratio_)
plt.plot(range(1, len(cumsum)+1), cumsum, 'bo-')
plt.axhline(y=0.95, color='r', linestyle='--', label='95% threshold')
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Variance Ratio')
plt.title('Cumulative Explained Variance')
plt.legend()
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Find optimal n_components for 95% variance
```

```
n_components_95 = np.argmax(cumsum >= 0.95) + 1
print(f"Components needed for 95% variance: {n_components_95}")
```

Ưu & Nhược điểm PCA

Ưu điểm:

- Giảm dimensionality hiệu quả
- Loại bỏ noise và redundant features
- Speed up training
- Visualization (giảm xuống 2D/3D)
- **Perfect reconstruction** nếu giữ tất cả components

Nhược điểm:

- **Linear transformation only** (không handle nonlinear relationships)
- Loss of interpretability (PC không còn ý nghĩa như original features)
- Sensitive với outliers
- Phải scale data trước

Module 5: Nonlinear & Distance-Based Dimensionality Reduction

Kernel PCA

Kernel PCA mở rộng PCA bằng cách map data lên **higher-dimensional space** sử dụng kernel trick, giúp handle **nonlinear relationships**!

Kernel Functions phổ biến

1. Linear Kernel

$$K(x, x') = x^T x'$$

Giống như PCA thông thường

2. Polynomial Kernel

$$K(x, x') = (x^T x' + c)^d$$

- c = constant
- d = degree

3. RBF (Gaussian) Kernel Most Popular

$$K(x, x') = \exp(-\gamma ||x - x'||^2)$$

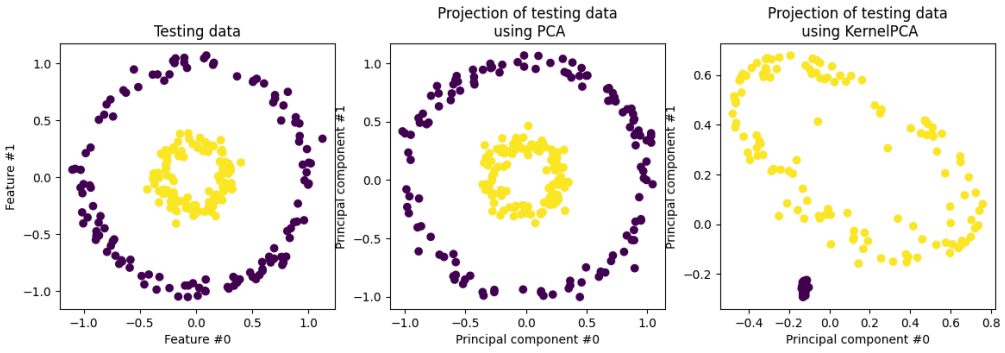
- γ = kernel coefficient
- Higher γ → more complex boundary

4. Sigmoid Kernel

$$K(x, x') = \tanh(\alpha x^T x' + c)$$

So sánh PCA vs Kernel PCA

Aspect	PCA	Kernel PCA
Transformation	Linear	Nonlinear
Reconstruction	Perfect	May not be perfect
Complexity	Low	High
Parameters	n_components only	kernel, gamma, alpha, etc.
Use Case	Linear patterns	Nonlinear patterns



Code Implementation - Kernel PCA

```
from sklearn.decomposition import KernelPCA

# RBF Kernel PCA
kernel_pca = KernelPCA(
    kernel='rbf',
    gamma=10,
    n_components=2,
    fit_inverse_transform=True,  # Allow reconstruction
    alpha=0.1
)

# Fit and transform
X_kpca = kernel_pca.fit_transform(X_scaled)

print(f"Original shape: {X_scaled.shape}")
print(f"Reduced shape: {X_kpca.shape}")

# Try different kernels
kernels = ['linear', 'poly', 'rbf', 'sigmoid']
fig, axes = plt.subplots(2, 2, figsize=(12, 10))

for idx, kernel in enumerate(kernels):
    kpca = KernelPCA(kernel=kernel, n_components=2, gamma=10)
    X_transformed = kpca.fit_transform(X_scaled)

    ax = axes[idx // 2, idx % 2]
    ax.scatter(X_transformed[:, 0], X_transformed[:, 1], c=y, cmap='viridis')
    ax.set_title(f'Kernel PCA - {kernel.upper()}')
    ax.set_xlabel('PC 1')
    ax.set_ylabel('PC 2')

plt.tight_layout()
plt.show()
```

Multi-Dimensional Scaling (MDS)

MDS là family of algorithms preserve **distances** giữa data points khi project xuống lower dimensions.

Key difference với PCA:

- **PCA:** Preserve **variance**
- **MDS:** Preserve **distances**

Metric MDS

Minimize distance metric trực tiếp:

$$\text{Stress} = \sqrt{\sum_{i<j}(d_{ij} - \hat{d}_{ij})^2}$$

Trong đó:

- d_{ij} = original distance
- $\hat{d}_{ij}$ = embedding distance

Non-Metric MDS

Apply function $f(\cdot)$ lên distance metric trước khi minimize:

$$\text{Stress} = \sqrt{\sum_{i<j}(f(d_{ij}) - \hat{d}_{ij})^2}$$

👉 **Flexible hơn!** Function f có thể là monotonic transformation

---

Code Implementation - MDS

```
from sklearn.manifold import MDS
from sklearn.metrics import euclidean_distances

# Compute distance matrix
distances = euclidean_distances(X_scaled)

# Metric MDS
mds_metric = MDS(
```

```
    n_components=2,
    metric=True,
    dissimilarity='precomputed',
    random_state=42
)
X_mds_metric = mds_metric.fit_transform(distances)

# Non-Metric MDS
mds_nonmetric = MDS(
    n_components=2,
    metric=False,
    dissimilarity='precomputed',
    random_state=42
)
X_mds_nonmetric = mds_nonmetric.fit_transform(distances)

# Visualize
fig, axes = plt.subplots(1, 2, figsize=(14, 6))

axes[0].scatter(X_mds_metric[:, 0], X_mds_metric[:, 1], c=y, cmap='viridis')
axes[0].set_title('Metric MDS')
axes[0].set_xlabel('Dimension 1')
axes[0].set_ylabel('Dimension 2')

axes[1].scatter(X_mds_nonmetric[:, 0], X_mds_nonmetric[:, 1], c=y, cmap
='viridis')
axes[1].set_title('Non-Metric MDS')
axes[1].set_xlabel('Dimension 1')
axes[1].set_ylabel('Dimension 2')

plt.tight_layout()
plt.show()
```

Module 6: Matrix Factorization - NMF

Non-Negative Matrix Factorization (NMF) decompose matrix thành 2 matrices với **tất cả giá trị ≥ 0**, tạo ra features **dễ interpret** hơn!

NMF Decomposition

$$V \approx W \times H$$

Trong đó:

- V = Original matrix $(m \times n)$
- W = Feature matrix $(m \times k)$
- H = Coefficient matrix $(k \times n)$
- k = số components (dimensions)

Constraint: $W \geq 0, H \geq 0$ (all elements non-negative)

NMF vs PCA

Aspect	PCA	NMF
Values	Any (positive/negative)	Non-negative only
Orthogonality	Orthogonal components	Not necessarily orthogonal
Interpretability	Low	**High** ⭐
Use Cases	General dimensionality reduction	Images, text, audio
Information Loss	Less	More (due to truncation)

Ưu điểm của NMF:
Features tend to be **additive** và **interpretable**!
Ví dụ: Facial recognition → components = nose, eyebrows, mouth

---

NMF cho Text Mining (NLP)

Input: Vectorized Text

TF-IDF (Term Frequency - Inverse Document Frequency)

$$\text{TF-IDF}(t, d) = \text{TF}(t, d) \times \text{IDF}(t)$$

$$\text{IDF}(t) = \log \frac{N}{|\{d : t \in d\}|}$$

- t = term (word)
- d = document
- N = total documents

Parameters to Tune

1. **Number of Topics** (n_components)
2. **Text Preprocessing:**
    - Stop words
    - Min/max document frequency
    - N-grams
    - Parts of speech

Output Matrices

1. W Matrix (Feature Matrix)

- Shape: $(m \text{ documents} \times k \text{ topics})$
- Tells: How terms relate to topics

2. H Matrix (Coefficient Matrix)

- Shape: $(k \text{ topics} \times n \text{ terms})$
- Tells: How to reconstruct documents from topics

Ứng dụng NMF

1. Topic Modeling

Tìm hidden topics trong document collections

2. Image Processing

- Image compression
- Feature extraction

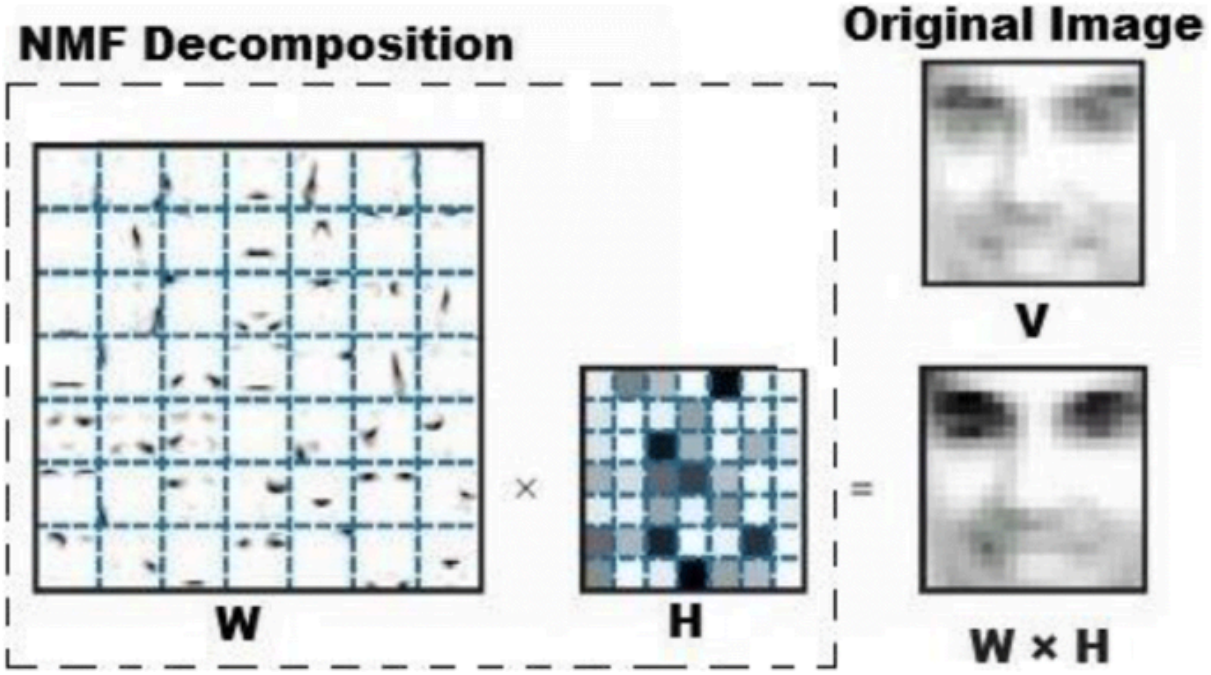
- Facial recognition

3. Audio Processing

- Music transcription
- Source separation

4. Recommender Systems

User-item matrix factorization



Code Implementation - NMF

```
from sklearn.decomposition import NMF
from sklearn.feature_extraction.text import TfidfVectorizer

# Example: Text data
documents = [
    "Machine learning is awesome",
    "Deep learning uses neural networks",
    "Natural language processing is fun",
    "Computer vision detects objects",
    "AI revolutionizes technology"
]
```

```
# Step 1: Vectorize text using TF-IDF
vectorizer = TfidfVectorizer(max_features=100, stop_words='english')
X_tfidf = vectorizer.fit_transform(documents)

# Step 2: Apply NMF
n_topics = 3
nmf = NMF(
    n_components=n_topics,
    init='random',
    random_state=42,
    max_iter=500
)

# Fit and transform
W = nmf.fit_transform(X_tfidf)  # Document-topic matrix
H = nmf.components_             # Topic-term matrix

print(f"W shape (documents × topics): {W.shape}")
print(f"H shape (topics × terms): {H.shape}")
```

Interpret Topics

```
# Get feature names (words)
feature_names = vectorizer.get_feature_names_out()

# Display top words for each topic
def display_topics(model, feature_names, n_top_words=10):
    for topic_idx, topic in enumerate(model.components_):
        top_words_idx = topic.argsort()[-n_top_words:][::-1]
        top_words = [feature_names[i] for i in top_words_idx]
        print(f"\n📌 Topic {topic_idx + 1}:")
        print(", ".join(top_words))

display_topics(nmf, feature_names, n_top_words=5)
```

NMF for Image Data

```
from sklearn.datasets import fetch_olivetti_faces

# Load face images
faces = fetch_olivetti_faces(shuffle=True, random_state=42)
X_faces = faces.data  # 400 faces, 64×64 pixels (flattened to 4096)

# Apply NMF
nmf_faces = NMF(n_components=25, init='random', random_state=42)
W_faces = nmf_faces.fit_transform(X_faces)
H_faces = nmf_faces.components_

# Visualize components
fig, axes = plt.subplots(5, 5, figsize=(10, 10))
for idx, ax in enumerate(axes.flat):
    ax.imshow(H_faces[idx].reshape(64, 64), cmap='gray')
    ax.axis('off')
    ax.set_title(f'Component {idx+1}')
plt.suptitle('NMF Components - Face Parts', fontsize=16)
plt.tight_layout()
plt.show()
```