Curse of Dimensionality

Definition:

The **Curse of Dimensionality** refers to the problems that arise when the number of **features (dimensions)** in a dataset increases too much. It becomes harder for machine learning models to perform well because of increased **sparsity**, **computation time**, and possible **drop in accuracy**.

It is also known as the **"Curse of Features."**

Why Is It a Problem?

- As **features increase**, the **distance** between data points increases.
- This causes **sparsity** — data becomes spread out and less meaningful in high dimensions.
- The model may still **work correctly**, but:
  - It becomes **slower**.
  - It requires **more computing power**.
  - It may lead to **overfitting** or **poor generalization**.

Note: There is an **optimal number of features**. Adding more features beyond that point can **hurt** model performance instead of helping.

Performance vs. Features

Increasing Features	Effects
Increase Accuracy (initially)	Improves up to a limit
Decrease Accuracy (later)	May decrease due to overfitting
Increase Computation Cost	Increases consistently

Solutions: Dimensionality Reduction

1. Feature Selection

Select the **most relevant features** and discard the rest.

a. Forward Selection

- Start with no features.
- Add one feature at a time that improves the model most.

b. Backward Elimination

- Start with all features.

- Remove the **least useful** ones step by step.

2. Feature Extraction

Create **new features** from the original ones using **linear combinations** or **transformations**.

a. PCA (Principal Component Analysis)

- Projects high-dimensional data into lower dimensions.
- Maximizes variance in the data.

b. LDA (Linear Discriminant Analysis)

- Like PCA but also considers **class labels**.
- Best for classification problems.

c. t-SNE (t-Distributed Stochastic Neighbor Embedding)

- Good for **visualizing high-dimensional data** in 2D or 3D.
- Non-linear technique.

Example:

Suppose you have a dataset with 1000 features, but only 20 of them are really useful. If you train a model on all 1000, it may take longer to train and might give lower accuracy. By applying **feature selection or PCA**, you can reduce it to, say, 25 features and still get better results.

Principal Component Analysis (PCA)

PCA is an **unsupervised machine learning technique** primarily used for **feature extraction** and **dimensionality reduction**. It is a classical yet powerful technique that helps address the **curse of dimensionality** by transforming high-dimensional data into a lower-dimensional space while preserving the essential structure and patterns of the original dataset.

Why Use PCA?

PCA is widely used for two main reasons:

1. Faster Execution and Efficiency

- Reduces the number of dimensions in the dataset.
- Improves the speed of machine learning algorithms.
- Decreases computational cost.
- Maintains a similar level of model performance.

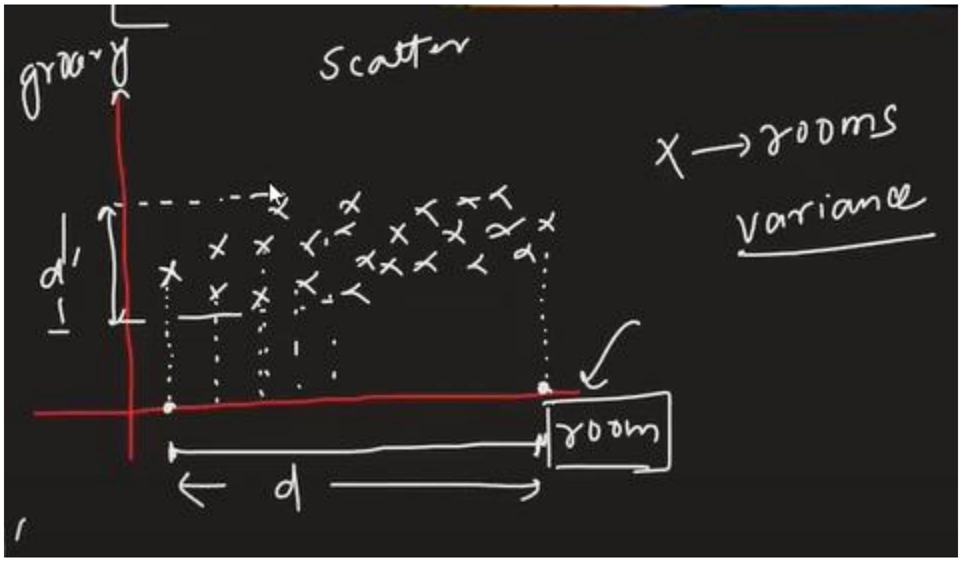
2. Data Visualization

- Enables conversion from high-dimensional data (e.g., 10D) to 2D.
- Makes it easier to visualize and interpret the data using graphs.

Feature Selection using Variance

When selecting features from a dataset, especially between two features:

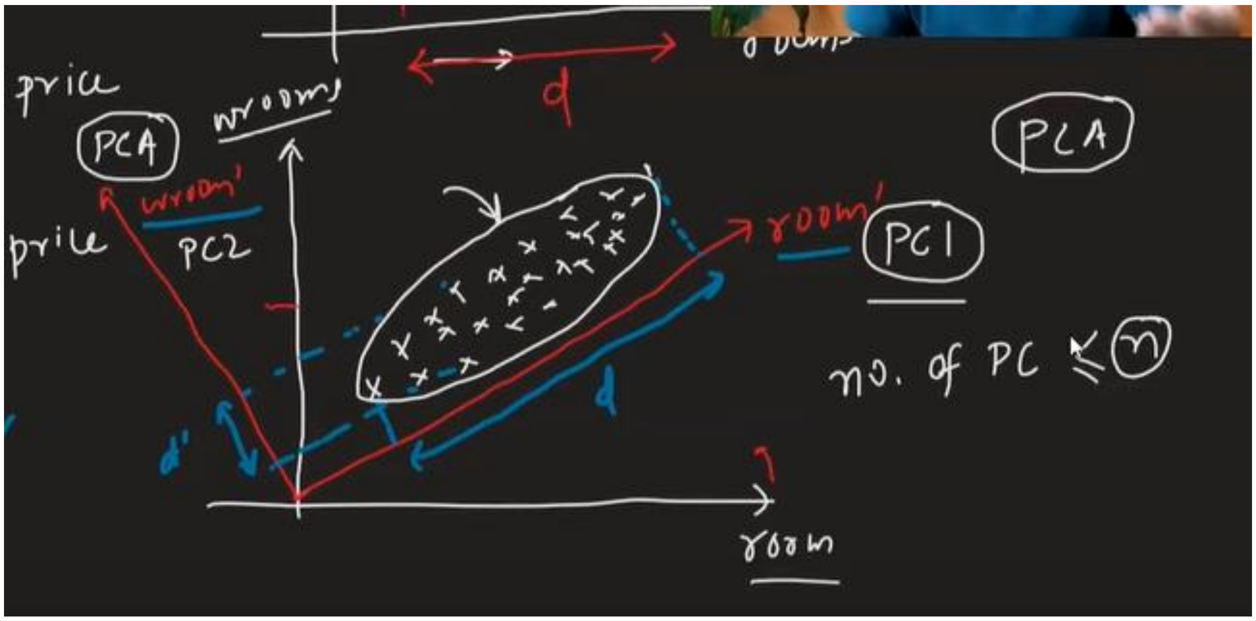
1. Plot the two features on the X and Y axes.
2. Calculate the distance or spread (variance) from the first point to the last point.
3. Choose the feature that exhibits **greater variance**, as it captures more meaningful information.



.

Feature Extraction:

PCA forget the existing feature and make a new feature. For example it makes house size from room and bathroom numbers



Feature Extraction with PCA

Principal Component Analysis (PCA) is a **feature extraction technique**, meaning it does not simply select from existing features, but **creates new features** by combining and transforming the original ones.

For example, instead of directly using the number of rooms and bathrooms in a house, PCA might combine them to generate a new feature like **"house size"**, which better represents the variation in the data.

How PCA Works Conceptually

When PCA identifies two features (e.g., rooms and bathrooms) that are **highly correlated** or contribute similarly, it tries to **reorient the axes** by viewing the data from a new perspective. This is done by:

- **Shifting the coordinate system** to find new axes (called **principal components**) that better capture the variation in the data.
- These new axes are named as **PC1 (Principal Component 1)** and **PC2 (Principal Component 2)**.
    - PC1 captures the direction of **maximum variance**.
    - PC2 captures the **remaining variance** orthogonal to PC1.

Even though the orientation changes, **the number of dimensions remains the same** initially. For example, two original features (room, bathroom) are transformed into two new components (PC1, PC2).

Understanding Variance in PCA

- **Variance** measures how much the data is spread out along a particular direction.
- PCA aims to **maximize the variance** in the first few components, as high variance typically indicates more meaningful structure.
- The **proportion of variance** captured by each principal component determines how much information from the original dataset is preserved.

```
from sklearn.decomposition import

PCA pca = PCA(n_components = 2)

X2D = pca.fit_transform(X)
```

1. ⍰ Standard PCA

- **What it does:**
  Reduces the number of features while keeping as much **variance (information)** as possible.
- **How it works:**
  Finds the principal components using **Singular Value Decomposition (SVD)** and projects the data onto a lower-dimensional space.
- **Use case:**
  Best for **medium-sized datasets** that fit in memory.
- **Example:**

  ```
  python
  CopyEdit
  from sklearn.decomposition import PCA
  pca = PCA(n_components=2)
  X2D = pca.fit_transform(X)
  ```

---

2. □ PCA for Compression

- **What it does:**
  Reduces dataset size significantly while preserving most of its information.
- **Why it's useful:**
  - Makes storage more efficient
  - Speeds up algorithms like **SVM**
  - Allows **reconstruction** of data (with some minor quality loss)
- **Example (MNIST):**
  Reduce from 784 to 154 features (95% variance preserved)

  ```
  python
  CopyEdit
  ```

```
pca = PCA(n_components=154)
X_reduced = pca.fit_transform(X_mnist)
X_recovered = pca.inverse_transform(X_reduced)
```

- **Term to remember:**
  - □ **Reconstruction error** – The difference between original and reconstructed data.

---

3. □ Incremental PCA (IPCA)

- **What it does:**
  Allows PCA to be done in **mini-batches** instead of all at once.
- **Why it's useful:**
  - Works on **large datasets** that don't fit in memory
  - Useful for **online learning** (real-time incoming data)
- **How:**
  Use `.partial_fit()` on chunks or use `np.memmap` for disk-based loading.
- **Example:**

```
python
CopyEdit
from sklearn.decomposition import IncrementalPCA
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_mnist, 100):
    inc_pca.partial_fit(X_batch)
X_reduced = inc_pca.transform(X_mnist)
```

---

4. □ Randomized PCA

- **What it does:**
  A **faster**, approximate version of PCA using a **randomized algorithm**.
- **Why it's useful:**
  When the number of components `d` is **much smaller** than the number of original features `n`, this method is **much quicker** than standard PCA.
- **Use case:**
  Large datasets where **speed is more important** than exact precision.
- **Example:**

```
python
CopyEdit
rnd_pca = PCA(n_components=154, svd_solver="randomized")
X_reduced = rnd_pca.fit_transform(X_mnist)
```

---

5. □ Kernel PCA (kPCA)

- **What it does:**
  Extends PCA to **nonlinear** data using the **kernel trick**.
- **Why it's useful:**
  Can uncover **nonlinear patterns**, **preserve clusters**, or **unroll curved data** like the Swiss roll.
- **Common kernels:**
  - RBF (Gaussian)
  - Sigmoid
  - Polynomial
- **Use case:**
  Data that lies on a **curved or complex manifold**.
- **Example:**

```
python
CopyEdit
from sklearn.decomposition import KernelPCA
kpca = KernelPCA(n_components=2, kernel="rbf", gamma=0.04)
X_reduced = kpca.fit_transform(X)
```