Supervised learning

Linear Regression :

- ✓ **Definition :** Linear Regression is a supervised learning algorithm used to model the relationship between a dependent variable (target) and one or more independent variables (features) by fitting a linear equation (straight line or hyperplane) to the observed data.
- ✓ It is used primarily for predicting continuous numeric values.
- ✓ **Example Use Case :** - Example Use Case
  - Predicting house price based on size, number of rooms, and location.
  - Predicting a student's marks based on hours of study.
  - Predicting salary based on years of experience.

- ✓ **Code :**

```python
from sklearn.linear_model import LinearRegression
# Sample Data
X = [[1], [2], [3], [4], [5]]
y = [2, 4, 5, 4, 5]
# Step 1: Create Model
model = LinearRegression()
# Step 2: Train Model
model.fit(X, y)
# Step 3: Predict
prediction = model.predict([[6]])
print("Prediction for x=6:", prediction)
```

- ✓ **Intuition :**
  1. Imagine plotting points of data on a 2D graph.
  2. You want to draw a straight line that best fits the data.
  3. The line should minimize the vertical distance (error) between the actual data points and the predicted values on the line.
  4. In multiple dimensions, this becomes a hyperplane instead of a line.

✓ Formula :

$$\beta_1 = \frac{\sum_{i=1}^{m}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{m}(x_i - \bar{x})^2}$$

$$\beta_0 = \bar{y} - \beta_1\bar{x}$$

- **Y** = Predicted target/output
- **$X_{(i)}$** = Feature/input variable
- **β0** = Intercept (bias term)
- **βi** = Coefficient (slope) for each feature,
- **ϵ** = Error/residual (difference between prediction and actual value)

✓ How It Works :

1. Takes the input features (X) and target values (y).
2. Fits a line (or hyperplane) that minimizes the error using Least Squares Method.
3. The model finds the best coefficients (**$B_{(i)}$** values) that reduce the Mean Squared Error (MSE):

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

4. After training, predictions are made using:

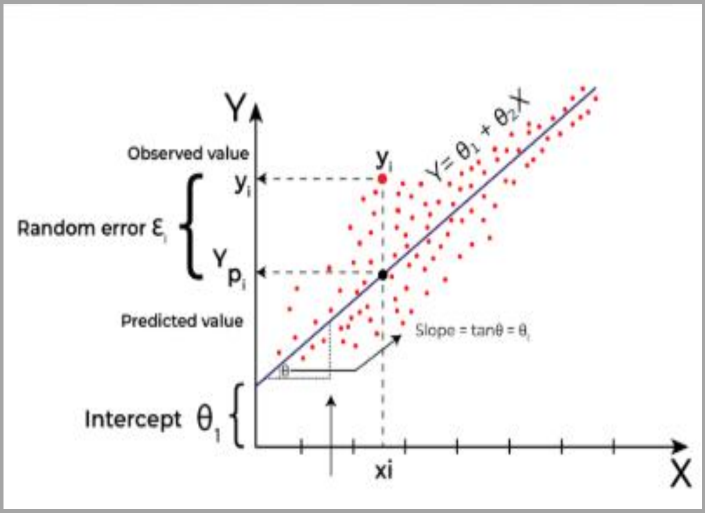
$$\hat{y} = \beta_0 + \sum_{i=1}^{n}\beta_i x_i$$

✓ Advantages :

□ Simple and easy to understand
□ Fast to train and test
□ Interpretable — you can see impact of each feature
□ Works well with linearly correlated data
□ Useful as a **baseline model**

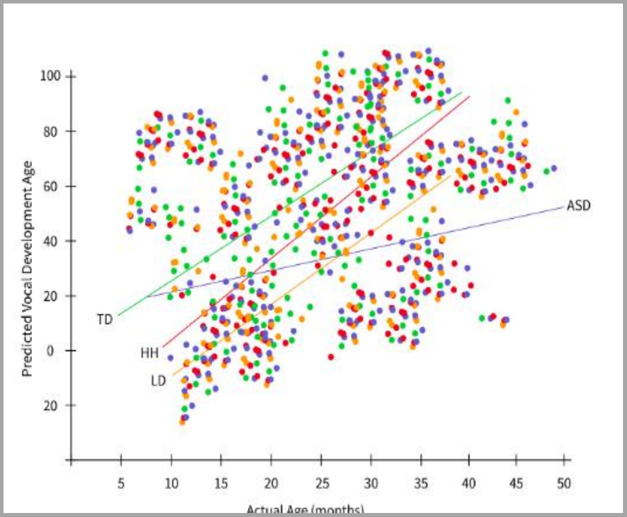
✓ Limitations :

□ Assumes linear relationship between input and output
□ Sensitive to **outliers**
□ Cannot handle complex (nonlinear) patterns
□ Assumes no **multicollinearity** between features
□ Assumes **homoscedasticity** (constant variance of error)
□ Poor performance if assumptions are violated



Y
Observed value
Random error ε_i
y_i
Y_{p_i}
Predicted value
Intercept θ_1
$Y = \theta_1 + \theta_2 X$
Slope = tanθ = θ_i
θ
xi
X

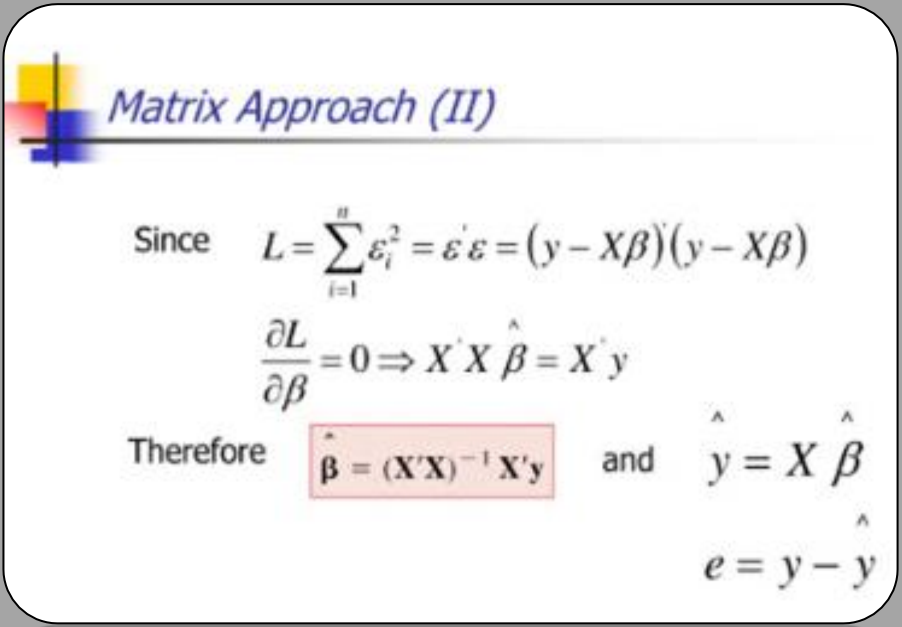
Simple



Multiple

✓ **From Scratch**

```python
class MeraLR:

    def __init__(self):
        self.coef_ = None
        self.intercept_ = None

    def fit(self,X_train,y_train):
        X_train = np.insert(X_train,0,1,axis=1)

        # calculate the coefficient
        betas = np.linalg.inv(np.dot(X_train.T,X_train)).dot(X_train.T).dot(y_train)
        self.intercept_ = betas[0]
        self.coef_ = betas[1:]

    def predict(self,X_test):
        y_pred = np.dot(X_test,self.coef_) + self.intercept_
        return y_pred
```



Matrix Approach (II)

Since $L = \sum_{i=1}^{n} \varepsilon_i^2 = \varepsilon'\varepsilon = (y - X\beta)'(y - X\beta)$

$$\frac{\partial L}{\partial \beta} = 0 \Rightarrow X'X\hat{\beta} = X'y$$

Therefore $\boxed{\hat{\beta} = (X'X)^{-1}X'y}$ and $\hat{y} = X\hat{\beta}$

$$e = y - \hat{y}$$

R2 Score :

- ✓ **Definition :** The R^2 score measures the proportion of the variance in the dependent variable (target) that is predictable from the independent variables (features).
- ✓ It ranges from:
  - 1 → perfect prediction
  - 0 → model predicts no better than the mean
  - < 0 → model performs worse than simply predicting the mean

1. **Formula :**

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

Where:

- $SS_{res} = \sum(y_i - \hat{y}_i)^2$ → residual sum of squares (error between actual and predicted)
- $SS_{tot} = \sum(y_i - \bar{y})^2$ → total sum of squares (variation from the mean)
- y_i = actual value
- $\hat{y}_i$ = predicted value
- $\bar{y}$ = mean of actual values

2. **Intuition:**
   - R^2 = 0.9 means 90% of the variation in the target can be explained by the model.
   - R^2 = 0.0 means the model does no better than predicting the average.
   - R^2 < 0.0 means the model is worse than just predicting the mean for every value.

```python
from sklearn.metrics import r2_score
y_true = [3, -0.5, 2, 7]
y_pred = [2.5, 0.0, 2, 8]
score = r2_score(y_true, y_pred)
print("R² Score:", score)
```

3. **Advantages:**
   - Easy to interpret
   - Widely used in regression evaluation
   - Shows how well the model explains the variance
4. **Limitations:**
   - Can be misleading if used alone
   - Doesn’t indicate whether predictions are biased
   - Can be negative (confusing for beginners)
   - Not useful for comparing different datasets

5. **When to Use :**
   - Use R^2 score when:
   - You’re dealing with regression tasks
   - You want to understand how well your model explains variability in data

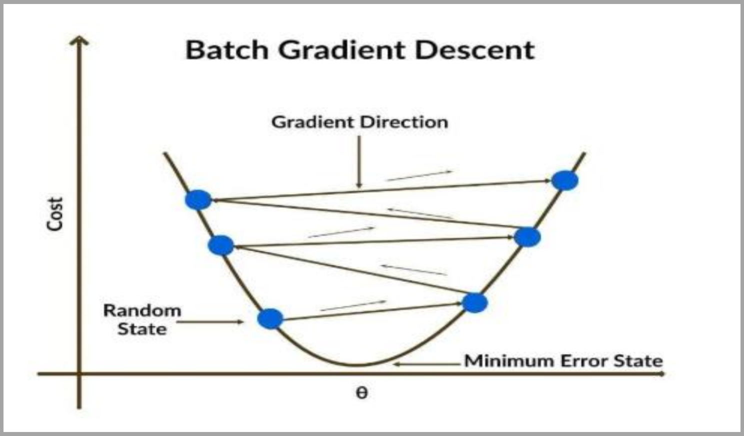
Gradient Descent :

1. **Definition :**
   - Gradient Descent is an optimization algorithm used to minimize the cost (loss) function of a machine learning model by updating the model parameters iteratively in the direction of steepest descent (negative gradient).
   - It is widely used in algorithms like Linear Regression, Logistic Regression, and Neural Networks to find the best-fitting parameters.
2. **Example Use Case**
   - Finding the optimal line in Linear Regression by minimizing the Mean Squared Error (MSE)
   - Training a Neural Network by minimizing the loss between predicted and actual labels
   - Optimizing Logistic Regression for binary classification

➢ **Types of Gradient Descent :**

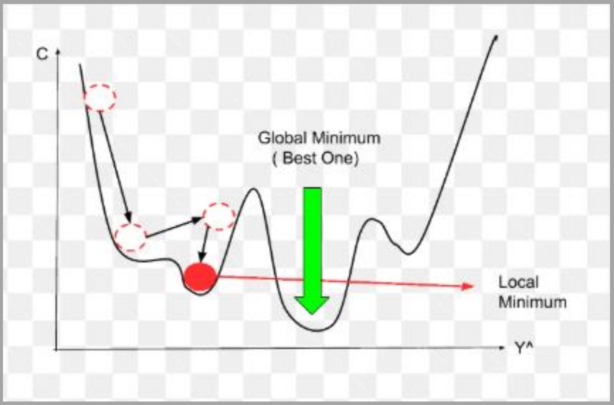
1. Batch gradient descent:

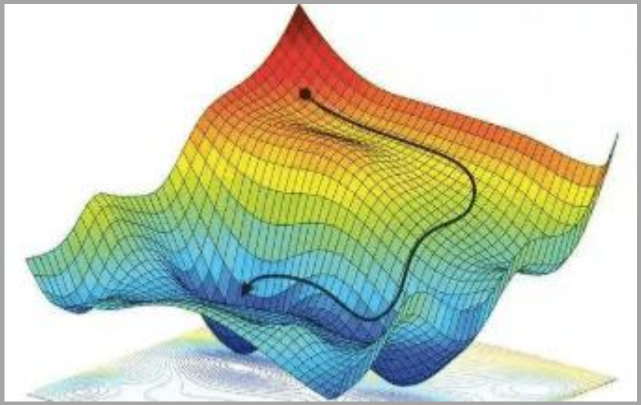
```
Class GDRegressor:

    def __init__(self, learning_rate=0.01, epochs=100):
        self.m = None
        self.b = None
        self.lr = learning_rate
        self.epochs = epochs

    def fit(self, X_train, y_train):
        # Ensure X_train is 2D
        if X_train.ndim == 1:
            X_train = X_train.reshape(-1, 1)

        n_samples, n_features = X_train.shape

        # Initialize parameters
        self.m = np.zeros(n_features)
        self.b = 0

        for _ in range(self.epochs):
            y_hat = np.dot(X_train, self.m) + self.b

            # Calculate gradients
            slope_b = -2 * np.mean(y_train - y_hat)
            slope_m = -2 * np.mean((y_train - y_hat).reshape(-1,1) * X_train, axis=0)
            # Update parameters
            self.b = self.b - self.lr * slope_b
            self.m = self.m - self.lr * slope_m

        print("Learned coefficients:", self.m)
        print("Learned intercept:", self.b)

    def predict(self, X_test):
        return np.dot(X_test, self.m) + self.b
```



2. Stochastic gradient descent:

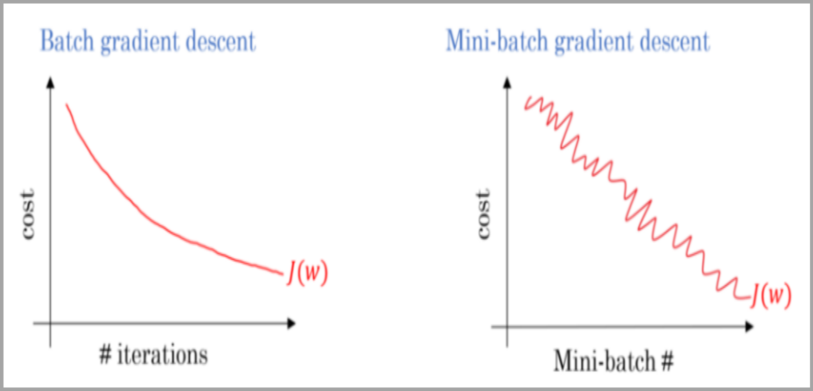
```
class SGDRegressor:
    def __init__(self, learning_rate=0.01, epochs=100):
        self.m = None
        self.b = None
        self.lr = learning_rate
        self.epochs = epochs

    def fit(self, X_train, y_train):
        if X_train.ndim == 1:
            X_train = X_train.reshape(-1, 1)
        n_samples, n_features = X_train.shape
        # Initialize parameters
        self.m = np.zeros(n_features)
        self.b = 0

        for epoch in range(self.epochs):
            for _ in range(self.iterations_per_epoch):
                # Randomly pick an index
                idx = np.random.randint(0, n_samples)

                x_i = X_train[idx]
                y_i = y_train[idx]
                # Prediction for single sample
                y_hat_i = np.dot(x_i, self.m) + self.b
                # Error for single sample
                error_i = y_i - y_hat_i
                # Compute gradients
                slope_b = -2 * error_i
                slope_m = -2 * error_i * x_i
                # Update parameters
                self.b -= self.lr * slope_b
                self.m -= self.lr * slope_m

    def predict(self, X_test):
        if X_test.ndim == 1:
            X_test = X_test.reshape(-1, 1)
        return np.dot(X_test, self.m) + self.b
```

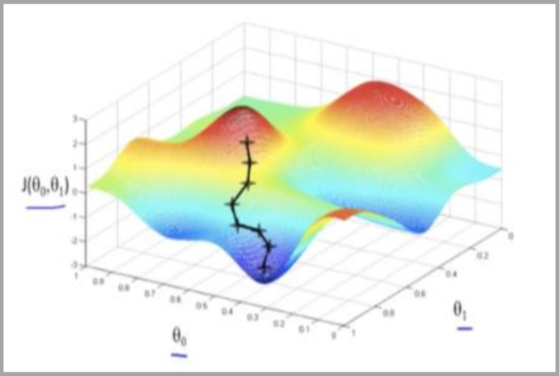


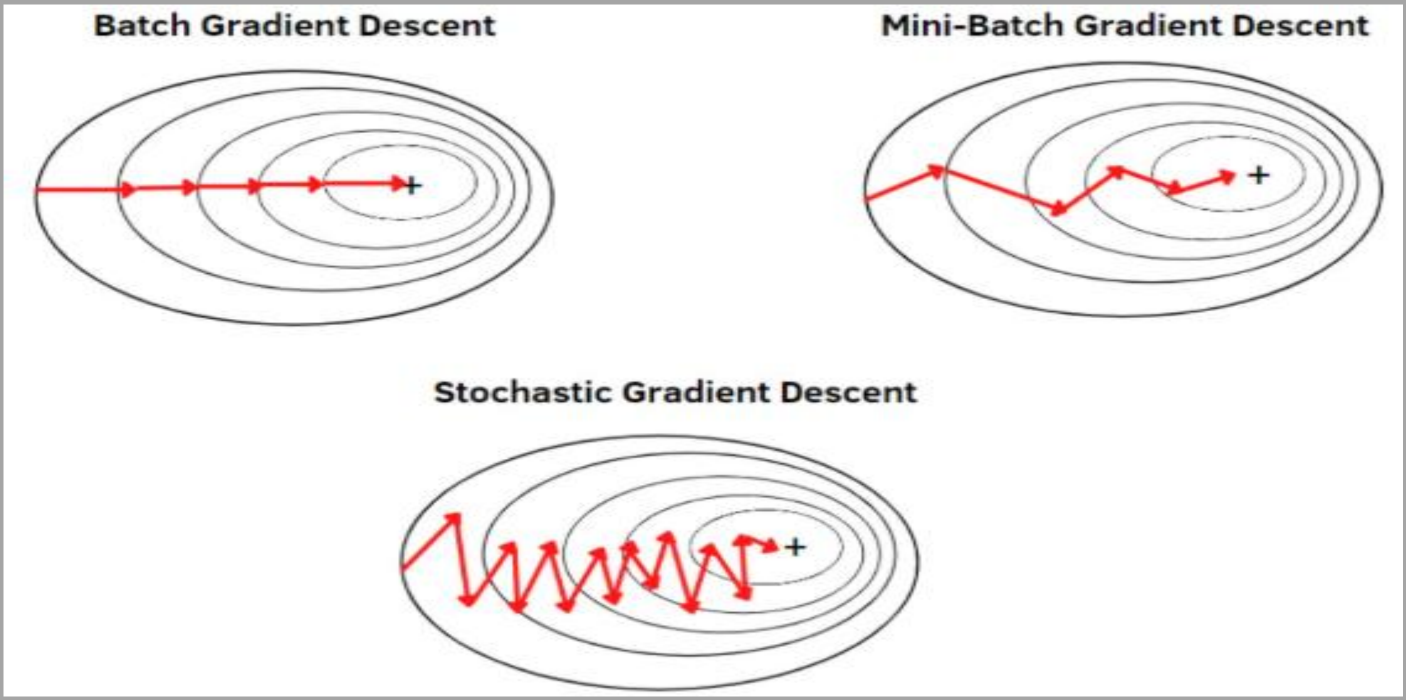


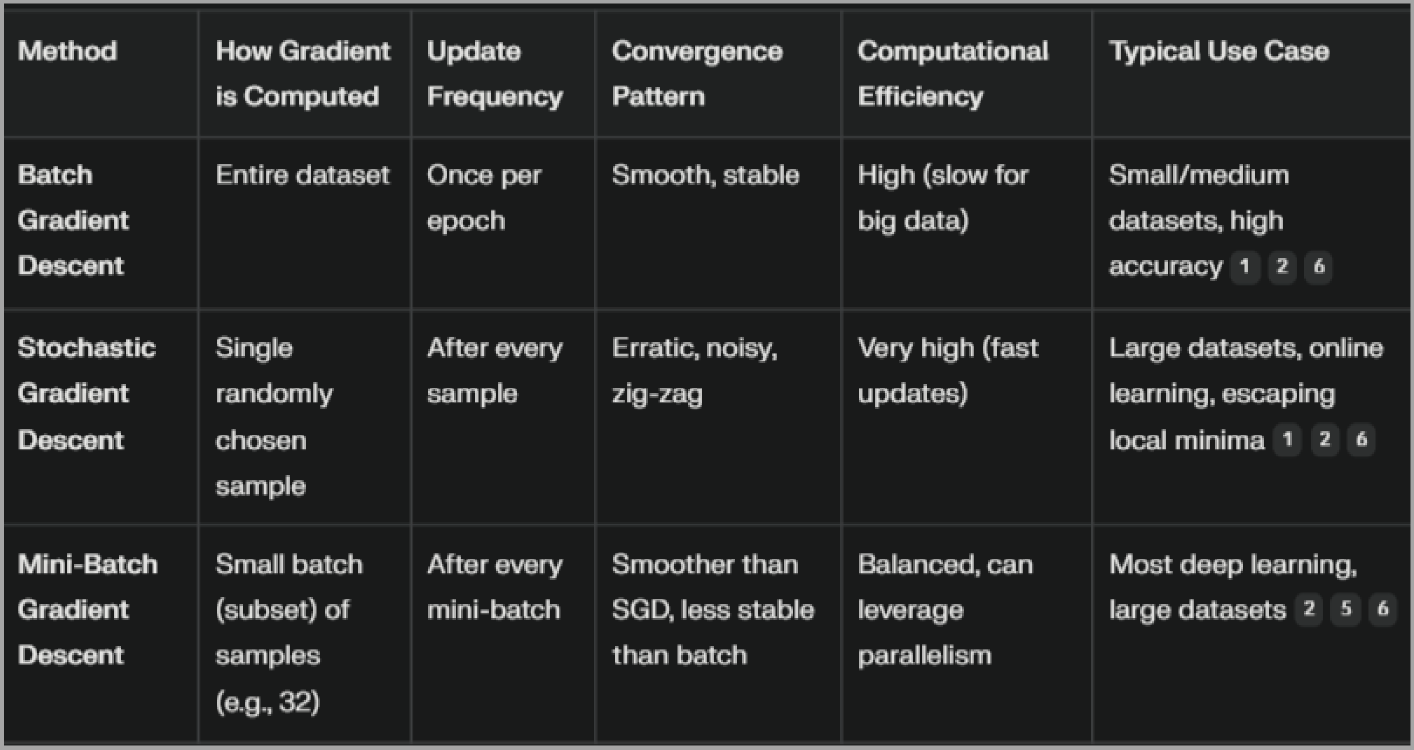
3. Mini-Batch Gradient descent

```
class MiniBatchGDRegressor:
    def __init__(self, learning_rate=0.01, epochs=100, batch_size=32):
        self.m = None
        self.b = None
        self.lr = learning_rate
        self.epochs = epochs
        self.batch_size = batch_size

    def fit(self, X_train, y_train):
        if X_train.ndim == 1:
            X_train = X_train.reshape(-1, 1)
        n_samples, n_features = X_train.shape
        # Initialize parameters
        self.m = np.zeros(n_features)
        self.b = 0

        for epoch in range(self.epochs):
            batch = n_samples // self.batch_size
            for _ in range(batch):
                # Randomly sample batch indices with replacement
                idx = np.random.randint(0, n_samples, size=self.batch_size)

                X_batch = X_train[idx]
                y_batch = y_train[idx]
                # Predictions for the batch
                y_hat_batch = np.dot(X_batch, self.m) + self.b
                # Errors for the batch
                error_batch = y_batch - y_hat_batch
                # Compute gradients (mean over batch)
                slope_b = -2 * np.mean(error_batch)
                slope_m = -2 * np.mean(error_batch.reshape(-1,1) * X_batch, axis=0)

                # Update parameters
                self.b -= self.lr * slope_b
                self.m -= self.lr * slope_m

    def predict(self, X_test):
        if X_test.ndim == 1:
            X_test = X_test.reshape(-1, 1)
        return np.dot(X_test, self.m) + self.b
```





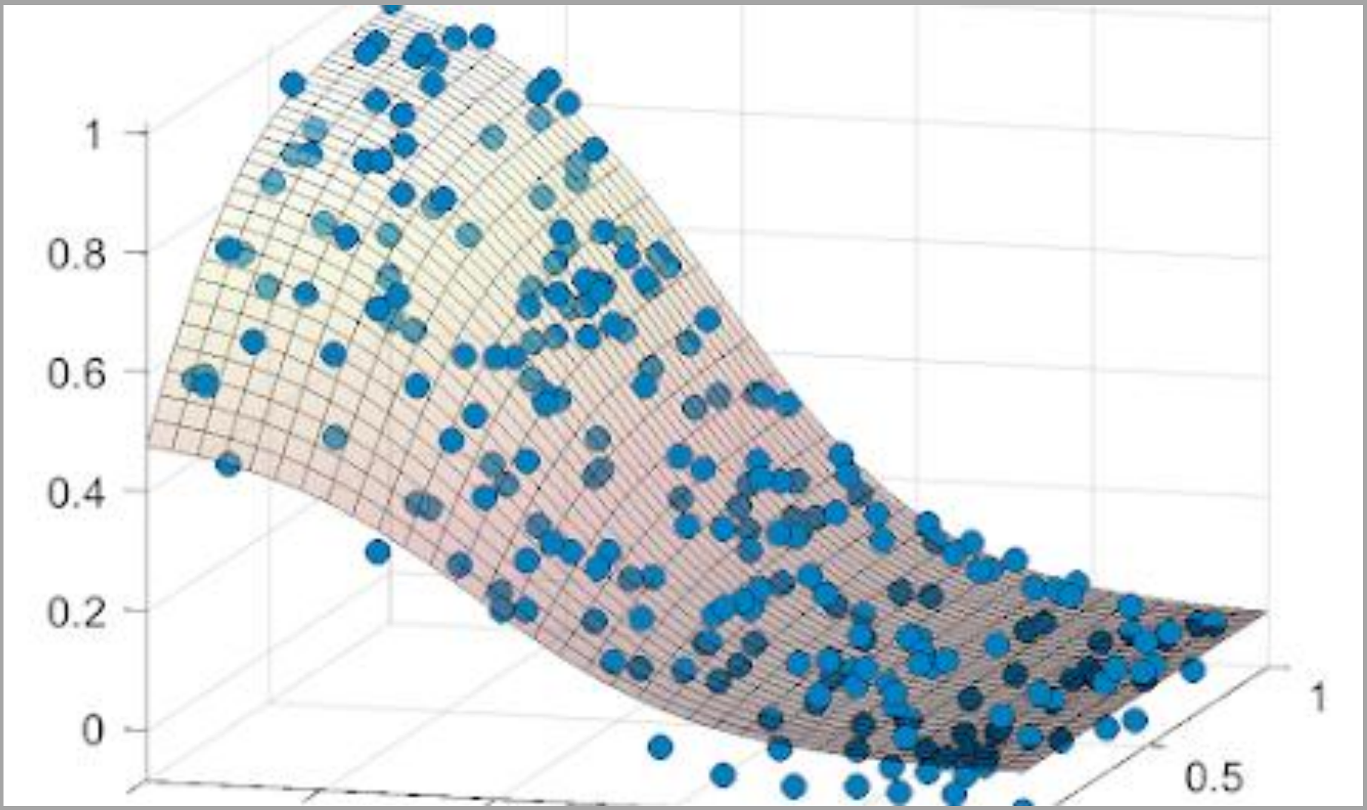


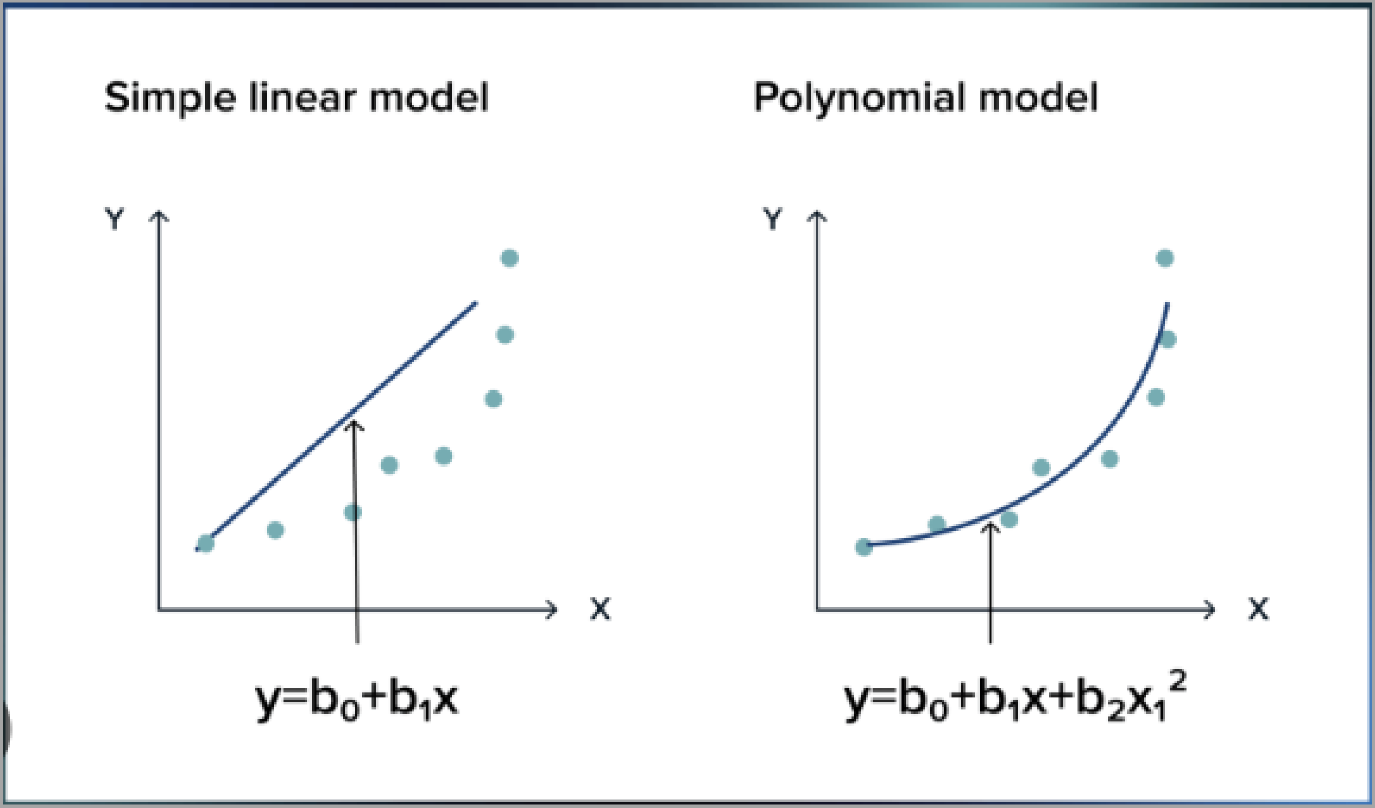


Method	How Gradient is Computed	Update Frequency	Convergence Pattern	Computational Efficiency	Typical Use Case
Batch Gradient Descent	Entire dataset	Once per epoch	Smooth, stable	High (slow for big data)	Small/medium datasets, high accuracy [1] [2] [6]
Stochastic Gradient Descent	Single randomly chosen sample	After every sample	Erratic, noisy, zig-zag	Very high (fast updates)	Large datasets, online learning, escaping local minima [1] [2] [6]
Mini-Batch Gradient Descent	Small batch (subset) of samples (e.g., 32)	After every mini-batch	Smoother than SGD, less stable than batch	Balanced, can leverage parallelism	Most deep learning, large datasets [2] [5] [6]

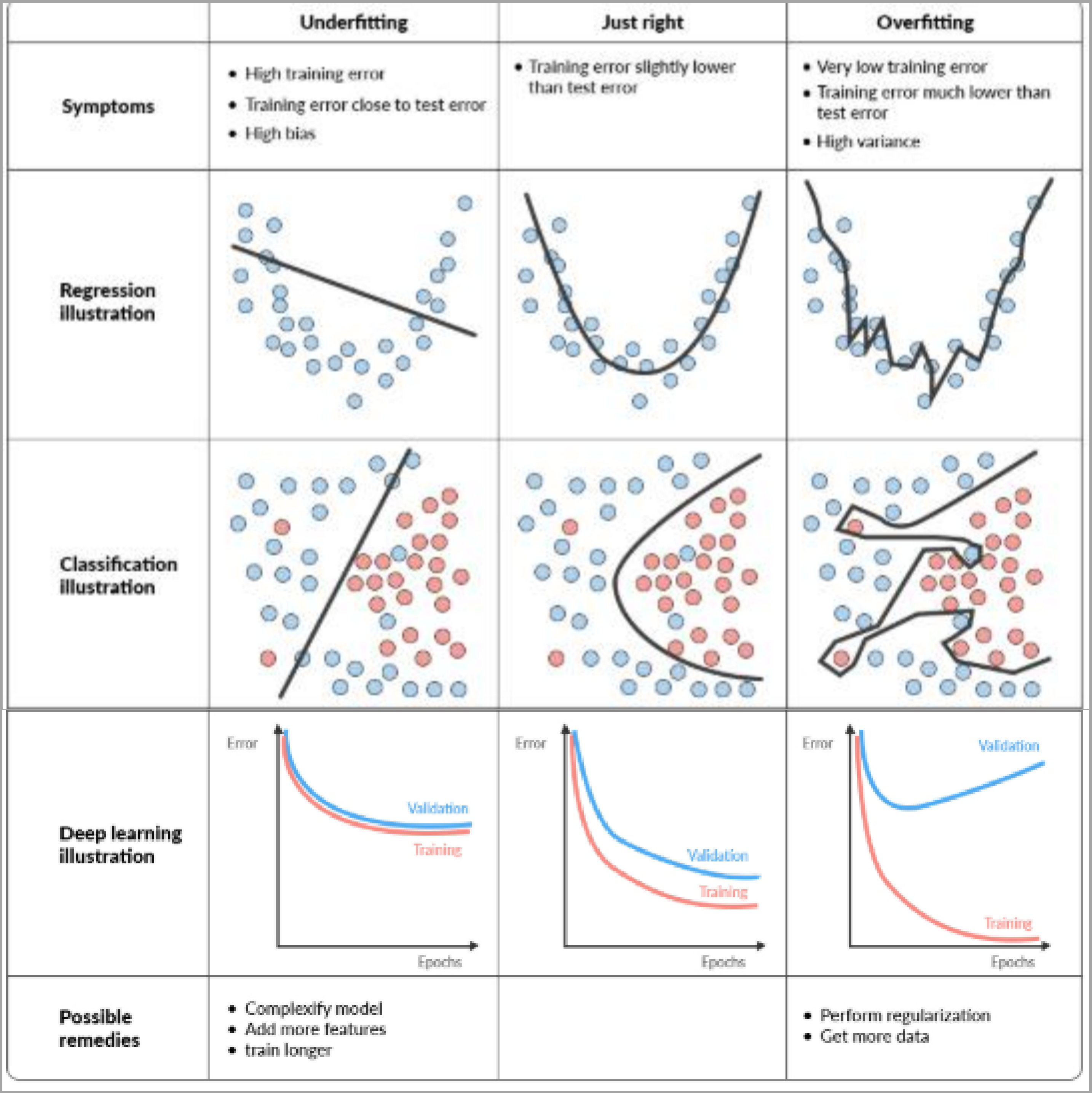
Type	Description	Use Case / When to Use	Pros	Cons
Batch Gradient Descent	Computes the gradient of the cost function using **entire dataset** for each step.	When dataset is **small to medium-sized** and can fit in memory.	- Converges smoothly- Deterministic steps	- Computationally expensive- Slow for large datasets
Stochastic Gradient Descent (SGD)	Computes the gradient using **one training sample at a time**.	When dataset is **very large** and cannot fit into memory.	- Fast updates- Can escape local minima	- High variance in updates- May never converge exactly
Mini-Batch Gradient Descent	Uses a **small batch (subset)** of training data to compute the gradient at each step (e.g., 32, 64, 128 samples).	Most commonly used in **deep learning**; balances performance and speed.	- Efficient memory usage- Faster convergence than full batch- Smoother than SGD	- Requires tuning of batch size- Slightly complex to implement
Momentum Gradient Descent	Adds a momentum term to accelerate convergence by **accumulating previous gradients** to determine direction.	When gradients oscillate (zig-zag) in high-dimensional space.	- Reduces oscillation- Faster convergence	- Needs tuning of momentum hyperparameter (usually 0.9)

Polyniomial Regession :





➢ Bias Varience tradeoff :



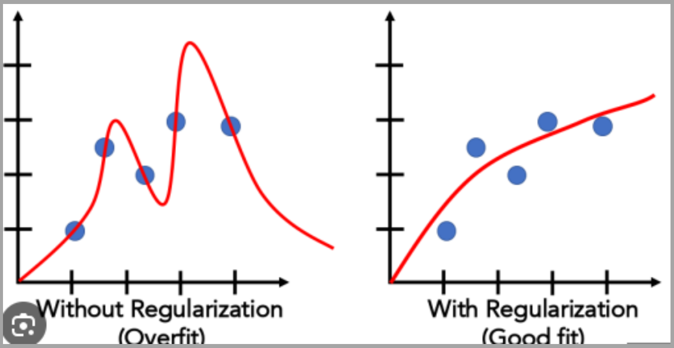
	Underfitting	Just right	Overfitting
Symptoms	• High training error • Training error close to test error • High bias	• Training error slightly lower than test error	• Very low training error • Training error much lower than test error • High variance
Regression illustration			
Classification illustration			
Deep learning illustration			
Possible remedies	• Complexify model • Add more features • train longer		• Perform regularization • Get more data

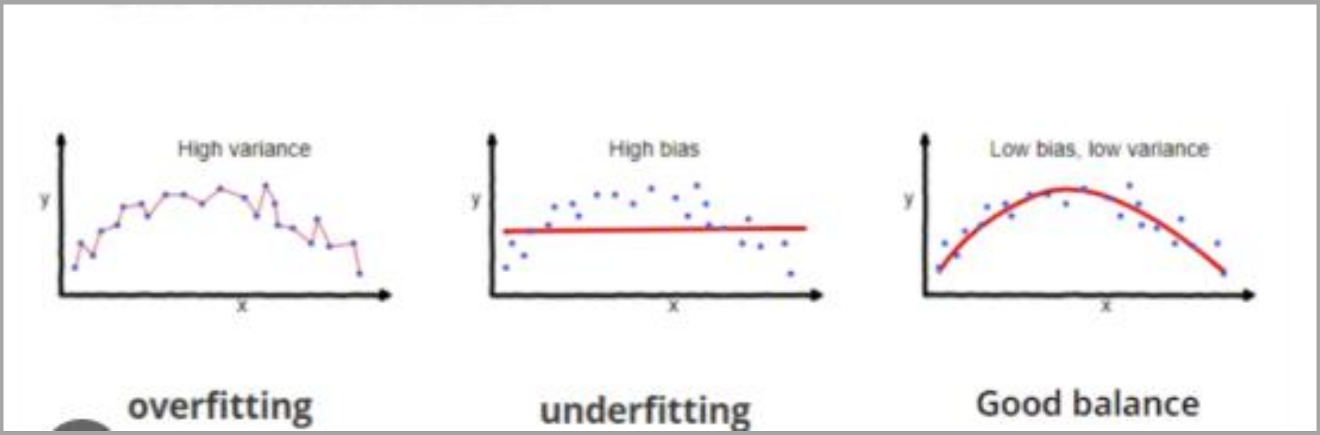
Regularization

1. Ridge Regression :

Ridge penalty term

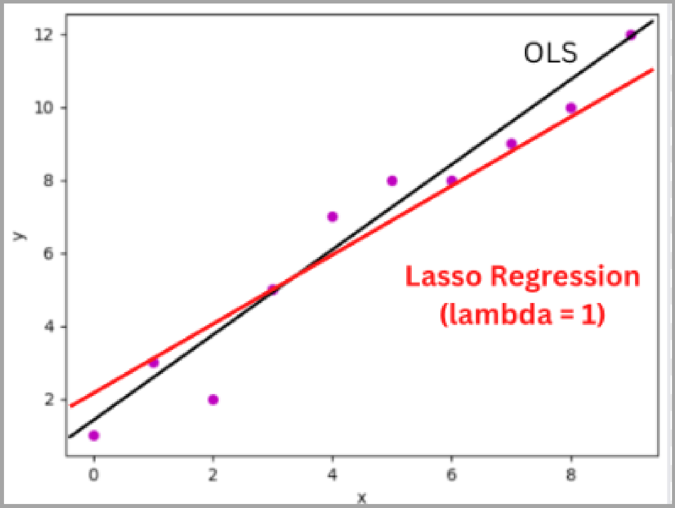
$$\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \boxed{\lambda \sum_{j=1}^{p} \beta_j^2}$$

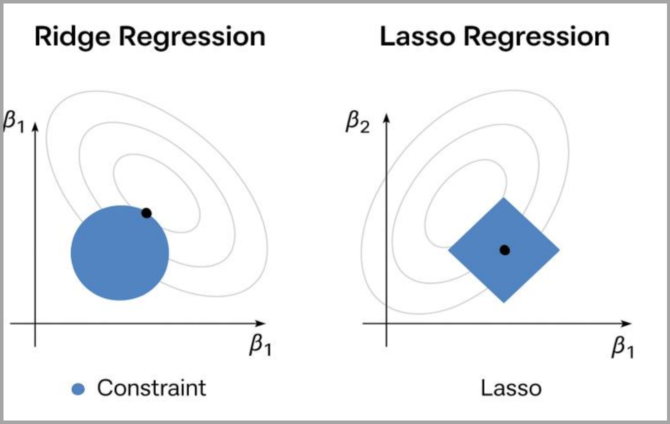




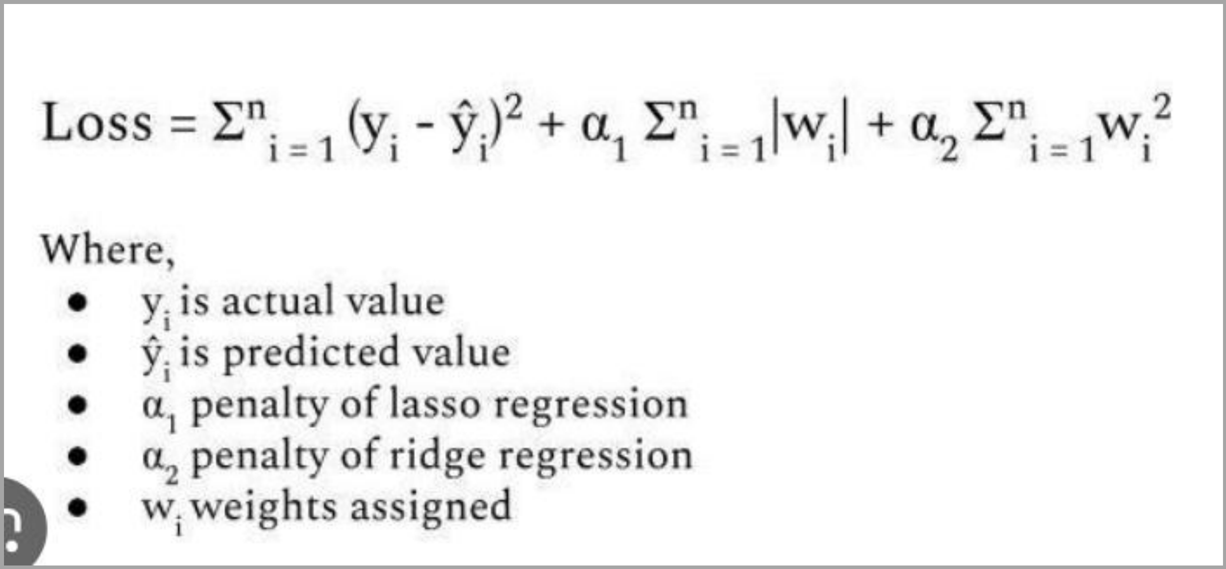
2. Lasso Regression :

$$\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2 + \lambda \sum_{j=1}^{p}|\beta_j|$$





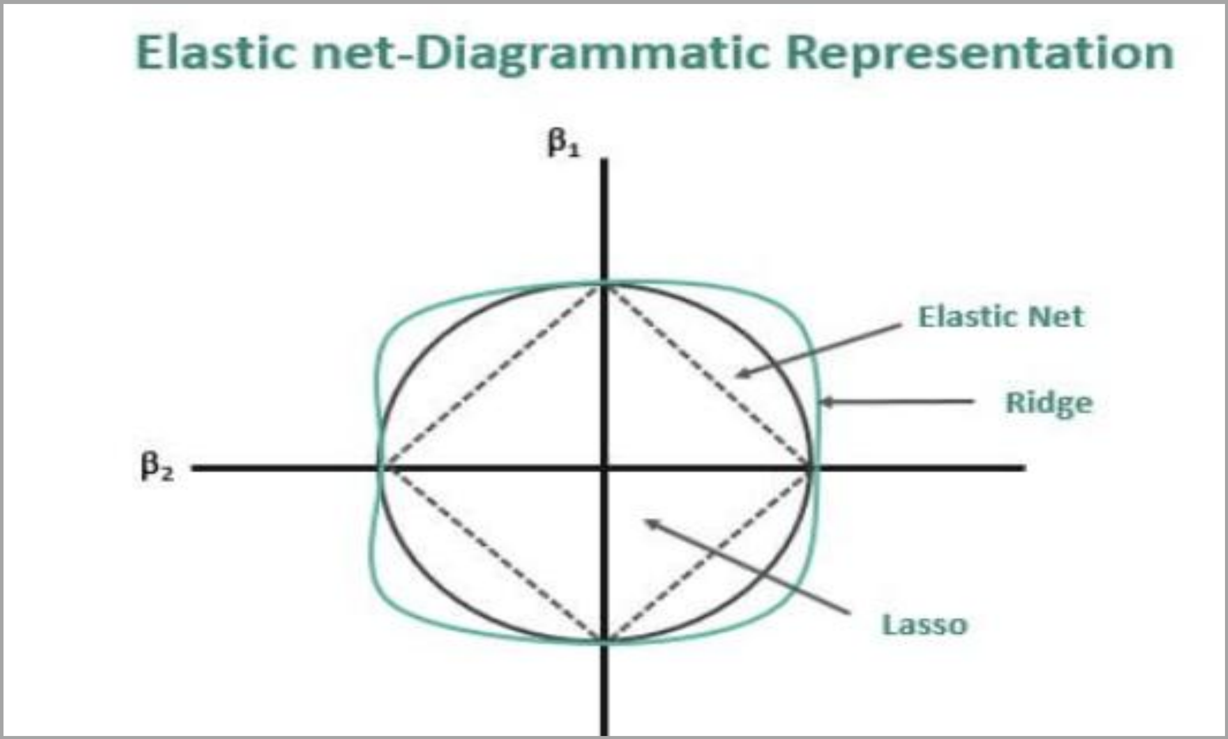
ElasticNet :



$$\text{Loss} = \Sigma^n_{i=1}(y_i - \hat{y}_i)^2 + \alpha_1 \Sigma^n_{i=1}|w_i| + \alpha_2 \Sigma^n_{i=1} w_i^2$$

Where,

- y_i is actual value
- $\hat{y}_i$ is predicted value
- α_1 penalty of lasso regression
- α_2 penalty of ridge regression
- w_i weights assigned



Feature / Aspect	Ridge Regression (L2 Regularization)	Lasso Regression (L1 Regularization)
Penalty Term	Adds L2 norm: λ∑j=1nβj2\lambda \sum_{j=1}^n \beta_j^2	Adds L1 norm: (\lambda \sum_{j=1}^n
Effect on Coefficients	Shrinks coefficients closer to zero, but never exactly zero	Can shrink some coefficients exactly to zero (feature selection)
Feature Selection	□ No — keeps all features in the model	□ Yes — removes irrelevant features automatically
When to Use	When most features are important and multicollinearity exists	When you expect many features to be irrelevant or want a sparse model
Handling Multicollinearity	Very effective	Also effective, but may drop one of the correlated features entirely
Computation	Slightly faster for very large datasets with many features	Slightly slower when features > observations, but still efficient
Interpretability	Less interpretable because all features remain	More interpretable because irrelevant features are removed
Bias–Variance Tradeoff	Slightly higher bias than OLS, lower variance	Can have higher bias than Ridge, especially if λ is large
Mathematical Penalty Shape	Circular constraint region (L2 ball)	Diamond-shaped constraint region (L1 ball)
Best For	When you want to reduce model complexity but keep all features	When you want to reduce complexity and select features

Interview takeaway line:

Lasso makes models sparse by using the L1 penalty, which encourages some coefficients to be exactly zero. This removes less important features, simplifying the model and improving interpretability.

Logistic Regeession :

1. **Definition :** It predicts the probability that a given input belongs to a certain class (usually binary: 0 or 1)
2. **Core Idea :** Instead of predicting values directly (like Linear Regression), it predicts a value between 0 and 1 using the Sigmoid (Logistic) Function

$$\text{sigmoid}(z) = \frac{1}{1 + e^{-z}}$$

$Z = w_1x_1 + w_2x_2 + ... + w_nx_n + b$

Output is **probability** → if > 0.5 → class 1, else class 0.

3. Steps in Logistic Regression :

a. Initialize weights (**w**) and bias (**b**).
b. Linear combination: Calculate **z** for each sample.
c. Sigmoid: Convert **z** to probability ppp.
d. Loss Function: Use Binary Cross-Entropy Loss:
e. Gradient Descent: Adjust **w** and **b** to minimize the loss.

Prediction: If p≥0.5p \ge 0.5p≥0.5 → class 1, else class 0

$$\text{Loss} = -\frac{1}{m}\sum [y \log(p) + (1 - y)\log(1 - p)]$$

4. Code :

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

5. Parameters :

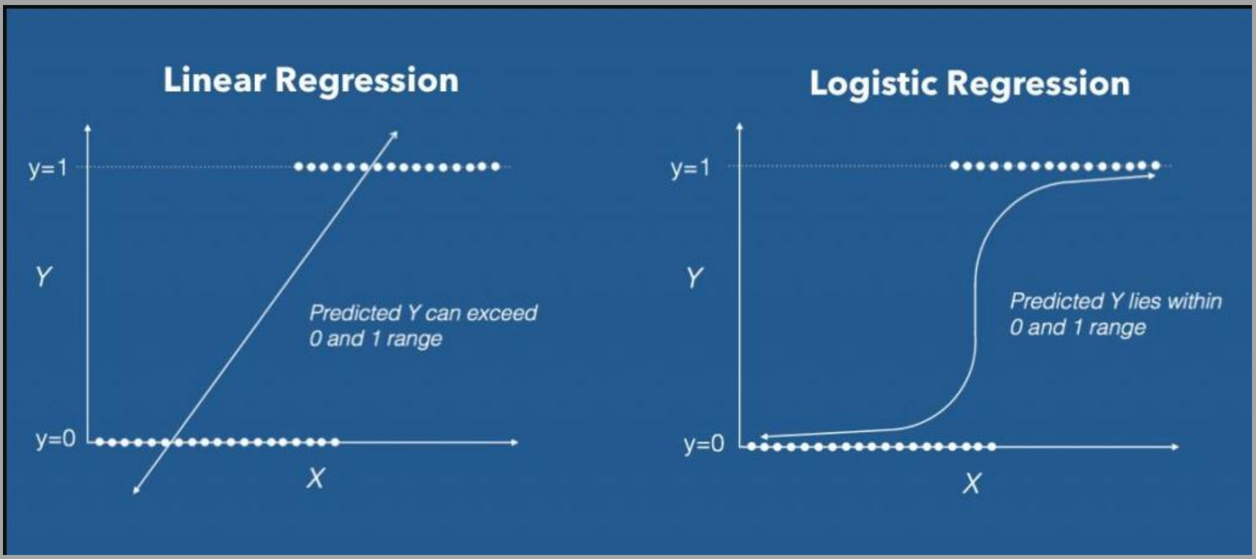
Parameter	Description
penalty	Type of regularization ('l1', 'l2', 'elasticnet', 'none').
C	Inverse of regularization strength. Smaller values = stronger regularization.
solver	Algorithm to use ('liblinear', 'saga', 'newton-cg', 'lbfgs').
max_iter	Maximum number of iterations to converge.
multi_class	'auto', 'ovr' (One-vs-Rest) or 'multinomial'.
random_state	Controls randomness for reproducibility.

6. Advantage :

- Simple and interpretable.
- Works well for binary classification.
- Outputs probabilities (not just class labels).
- Efficient on large datasets.

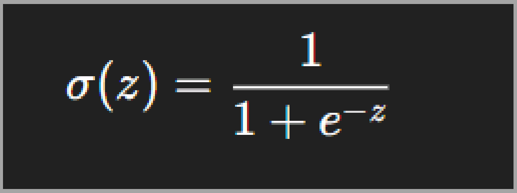
7. Limitation :

- Only works well with **linearly separable** data.
- Not suitable for complex non-linear decision boundaries.
- Sensitive to multicollinearity.
- Assumes independent features.


Linear Regression
y=1
Y
y=0
X
Predicted Y can exceed 0 and 1 range
Logistic Regression
y=1
Y
y=0
X
Predicted Y lies within 0 and 1 range

Sigmoid Function :

1. **Definition :** The sigmoid function is a mathematical function that transforms any real-valued number into a value between 0 and 1. It is commonly used in logistic regression, neural networks, and probability estimation.
2. **Mathematical Formula :**

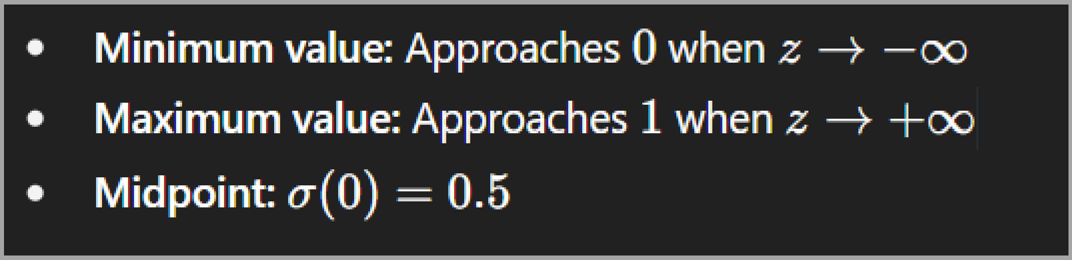


$$\sigma(z) = \frac{1}{1+e^{-z}}$$

Where:

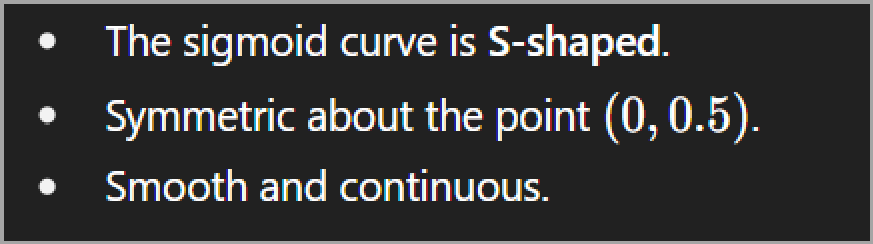
- z = input (can be any real number)
- e = Euler's number (≈2.718\approx 2.718≈2.718)

3. **Output Range :**



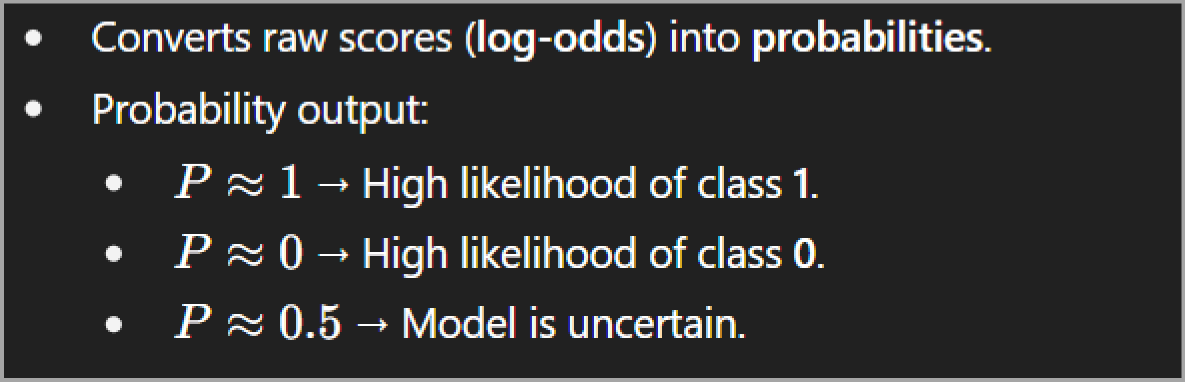
- **Minimum value:** Approaches 0 when $z \to -\infty$
- **Maximum value:** Approaches 1 when $z \to +\infty$
- **Midpoint:** $\sigma(0) = 0.5$

4. **Shape :**



- The sigmoid curve is **S-shaped**.
- Symmetric about the point $(0, 0.5)$.
- Smooth and continuous.

5. **Interpretation :**



- Converts raw scores (**log-odds**) into **probabilities**.
- Probability output:
  - $P \approx 1$ → High likelihood of class 1.
  - $P \approx 0$ → High likelihood of class 0.
  - $P \approx 0.5$ → Model is uncertain.

6. **Derivative :** The derivative of the sigmoid function is important in gradient-based learning algorithms:

$$\sigma'(z) = \sigma(z) \cdot \left(1 - \sigma(z)\right)$$

- Maximum slope occurs at z = 0 (steepest point).

7. **Advantage :**

- Smoothly maps inputs to (0,1).
- Useful for probabilistic interpretation.
- Differentiable everywhere (good for optimization).

8. **Disadvantages :**

- **Vanishing Gradient Problem :**

When the input value is very large (positive or negative), the slope of the sigmoid becomes almost zero.
This makes learning very slow during training because the weight updates are tiny.

- **Not Zero-Centered :**

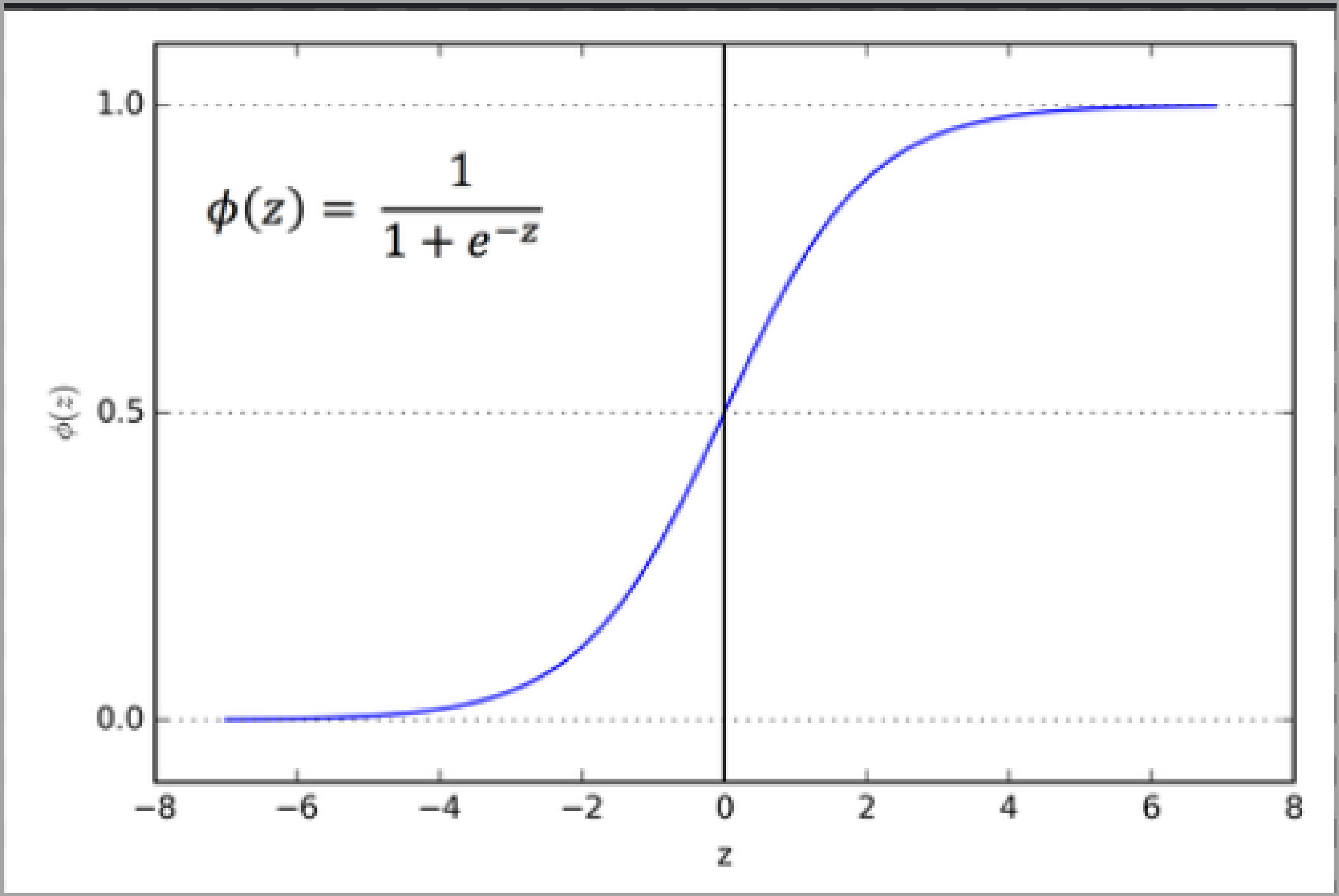
The output of sigmoid is always positive (0 to 1).
This can cause zig-zag movement in gradient descent, making optimization slower.

- **Computationally Expensive :**

It needs the exponential function (e^x), which takes more time compared to simple functions like ReLU.

- **Poor for Deep Networks :**

In deep neural networks, repeated multiplication of small gradients can make the network stop learning.


$\phi(z) = \frac{1}{1+e^{-z}}$
1.0
0.5
0.0
$\phi(z)$
−8
−6
−4
−2
0
2
4
6
8
z

When and Which Metric used for Score?

Task Type	Common Algorithms	Best Primary Metrics	Alternative / Situational Metrics
Binary Classification	Logistic Regression, Random Forest Classifier, Gradient Boosting Classifier, XGBoost, LightGBM, SVM, Neural Networks	F1-score (balanced precision & recall)	Precision, Recall, ROC-AUC, PR-AUC, Log Loss
Multiclass Classification	Random Forest Classifier, Gradient Boosting Classifier, XGBoost, LightGBM, SVM, Neural Networks, KNN	Weighted F1-score, Macro F1-score	Accuracy (balanced classes), Top-K Accuracy, Log Loss
Regression	Linear Regression, Random Forest Regressor, Gradient Boosting Regressor, XGBoost, LightGBM, SVR, Neural Networks	RMSE, MAE	R^2 score, MSE, MAPE
Ranking / Recommendation	XGBoost Ranker, LightGBM Ranker, Matrix Factorization, Neural Collaborative Filtering	NDCG, MAP	Recall@K, Hit Rate
Clustering (Unsupervised)	K-Means, DBSCAN, Agglomerative Clustering, Gaussian Mixture Models	Silhouette Score	Davies–Bouldin Index, Adjusted Rand Index (if labels available)
Anomaly Detection	Isolation Forest, One-Class SVM, Autoencoders	F1-score (if labels), ROC-AUC	Precision-Recall AUC, Specificity, Matthews Correlation Coefficient

Comparison of all Algorithm :

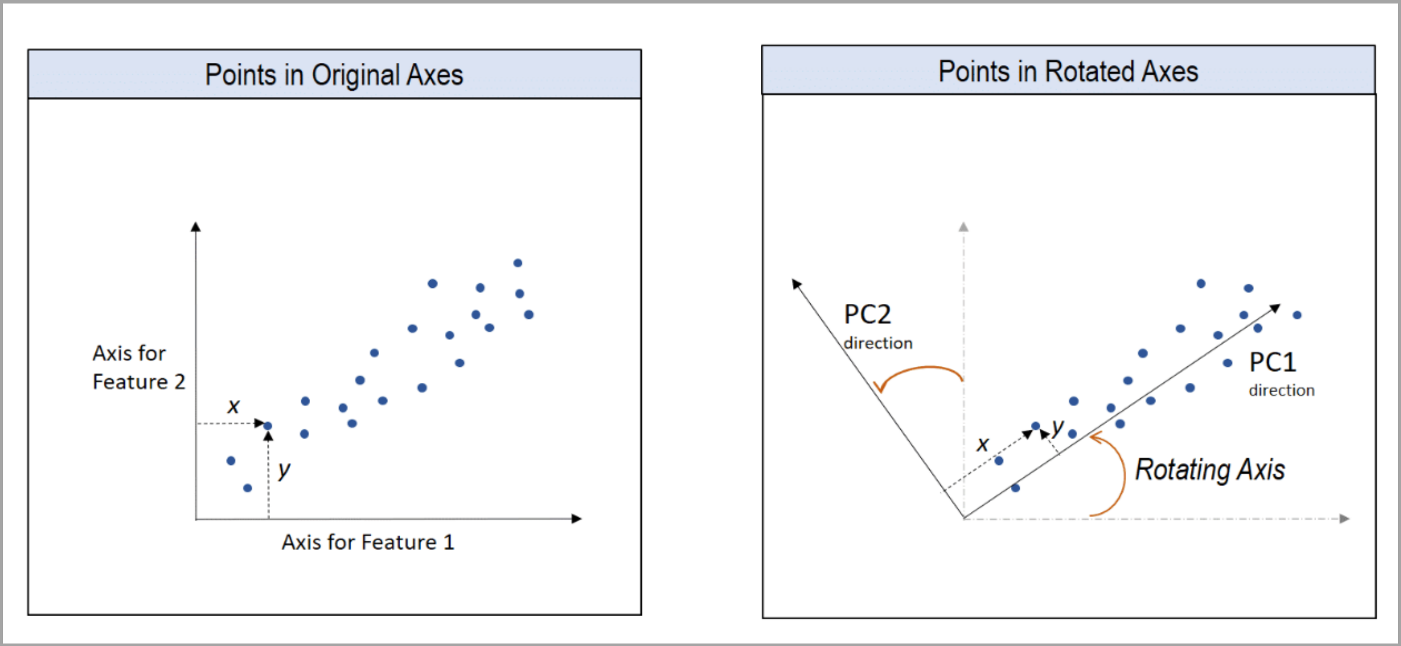
Unsupevised Learning

❖ PCA (Principal Component Analysis) :

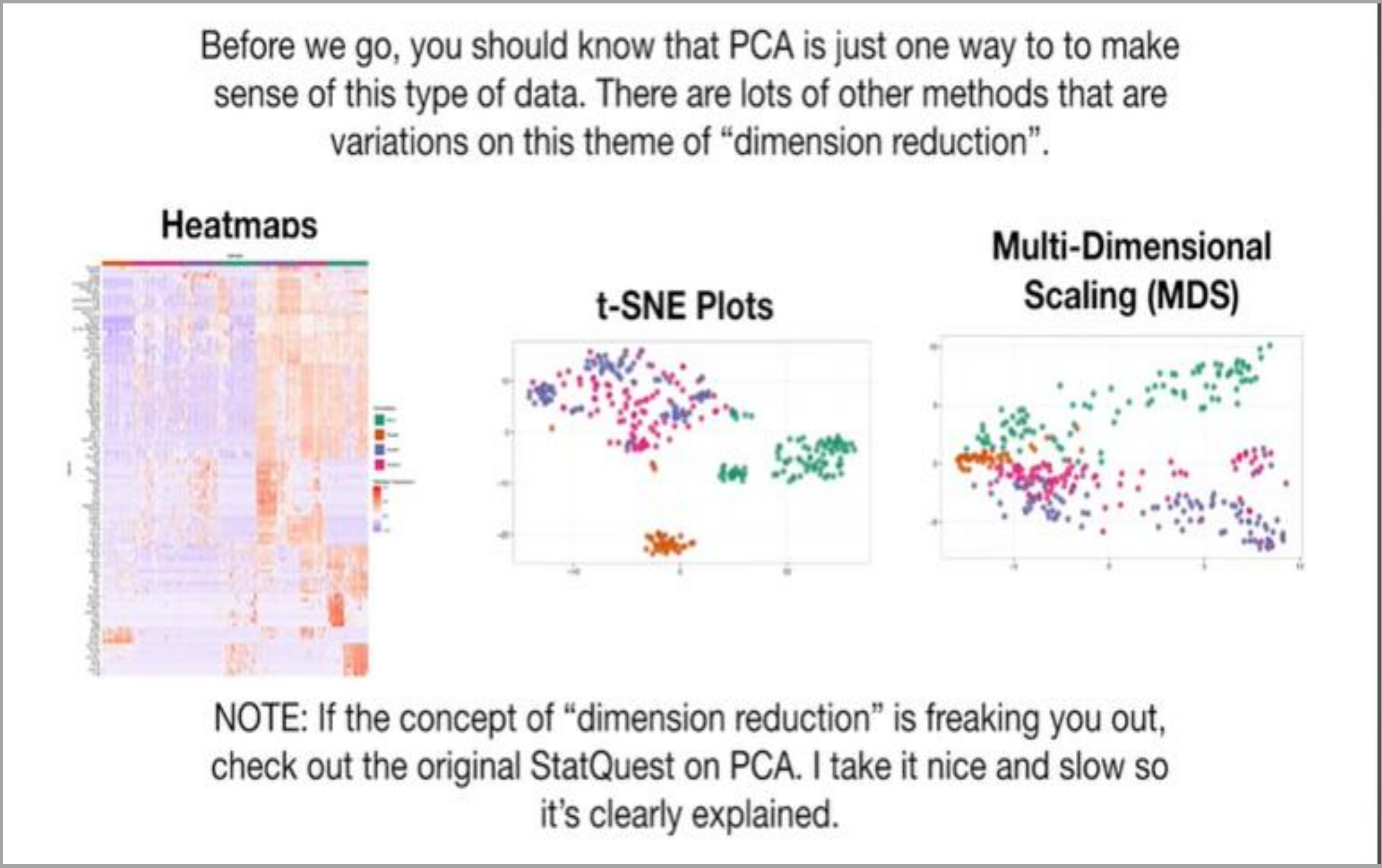
- ➢ Principal Component Analysis (PCA) is an unsupervised learning technique widely used in machine learning and data analysis for dimensionality reduction, feature extraction, and data visualization

Steps:

1. Standardize or Normalize the data **(Mean =0 , S.D = 1)**
   Adjust each feature so it has a mean of zero and a standard deviation of one. This ensures all features contribute equally.(Ex. Data = [-3,-2,-1,0,1,2,3])
2. Center the data (subtract the mean)
   Make sure each feature has a mean of zero by subtracting its mean from every value.
3. Compute the covariance matrix
   Calculate how much the features vary together (covariance) to understand their relationships.
4. Calculate eigen vectors and eigen values of the covariance matrix
   Identify the directions (principal components) where the data varies the most (eigen vectors) and how much variance is in those directions (eigen values).
5. Sort eigen vectors by eigen values in descending order
   Rank the components by the amount of variance they explain (most important first).
6. Select the top k principal components
   Choose the number of components (k) that capture most of the variance (information).
7. Project the data onto the top k principal components
   Transform the original dataset by expressing it in terms of these new axes to reduce dimensions while preserving as much information as possible.



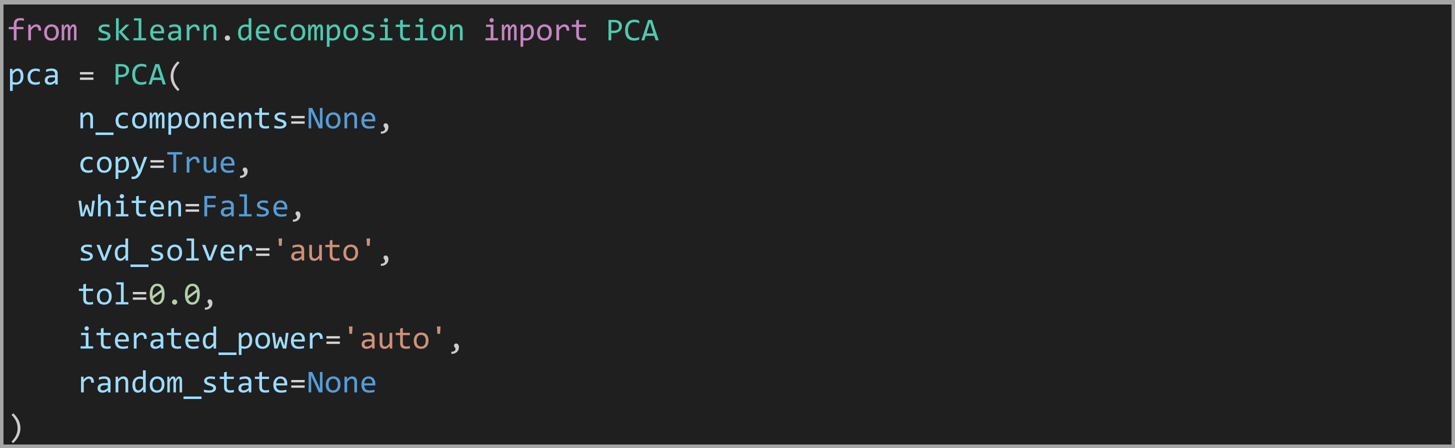
```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
pca = PCA(n_components=2)  # reduce to 2 dimensions
X_pca = pca.fit_transform(X_scaled)
```



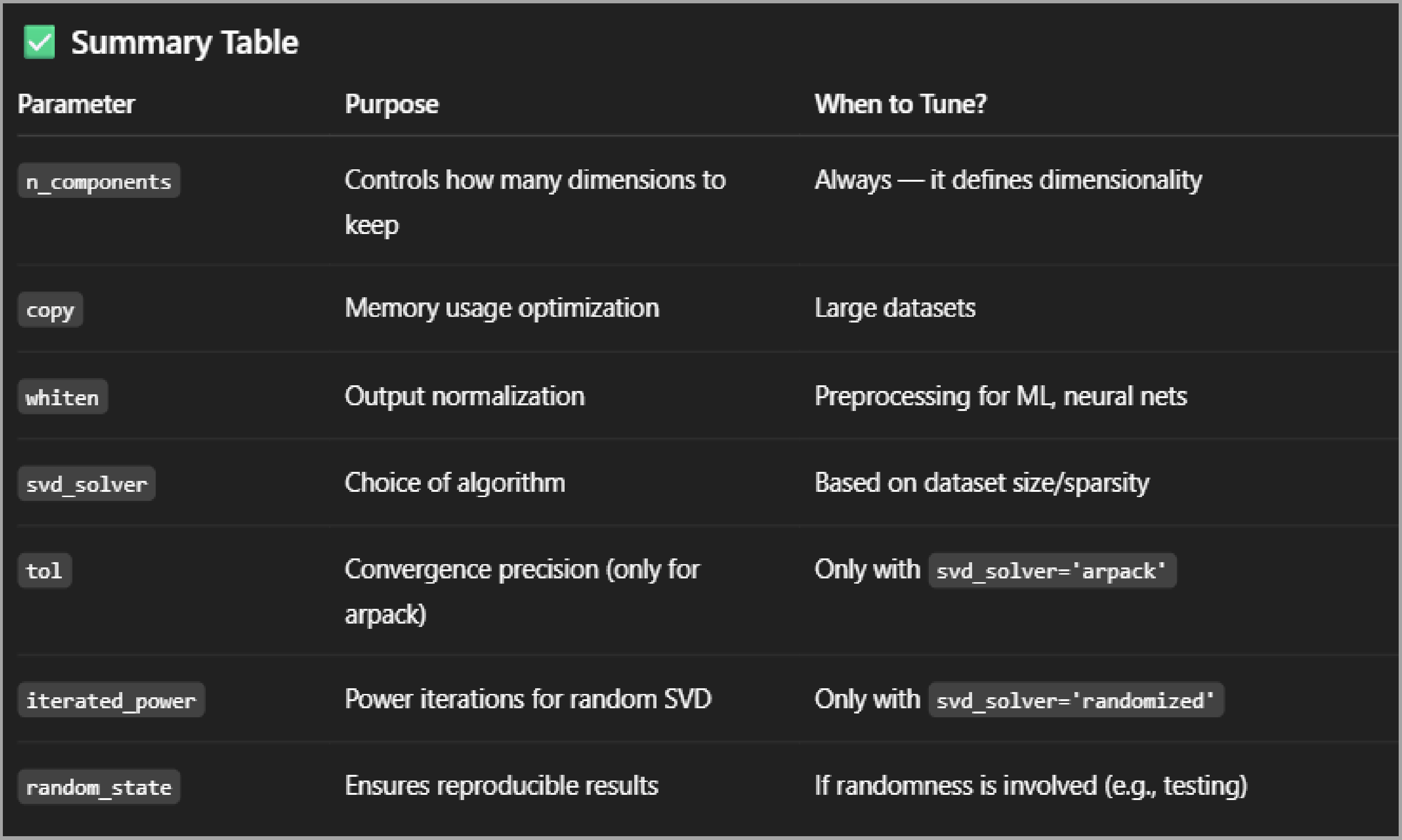
Before we go, you should know that PCA is just one way to to make sense of this type of data. There are lots of other methods that are variations on this theme of "dimension reduction".

Heatmaps
t-SNE Plots
Multi-Dimensional Scaling (MDS)

NOTE: If the concept of "dimension reduction" is freaking you out, check out the original StatQuest on PCA. I take it nice and slow so it's clearly explained.



```python
from sklearn.decomposition import PCA
pca = PCA(
    n_components=None,
    copy=True,
    whiten=False,
    svd_solver='auto',
    tol=0.0,
    iterated_power='auto',
    random_state=None
)
```



✅ Summary Table

Parameter	Purpose	When to Tune?
`n_components`	Controls how many dimensions to keep	Always — it defines dimensionality
`copy`	Memory usage optimization	Large datasets
`whiten`	Output normalization	Preprocessing for ML, neural nets
`svd_solver`	Choice of algorithm	Based on dataset size/sparsity
`tol`	Convergence precision (only for arpack)	Only with `svd_solver='arpack'`
`iterated_power`	Power iterations for random SVD	Only with `svd_solver='randomized'`
`random_state`	Ensures reproducible results	If randomness is involved (e.g., testing)

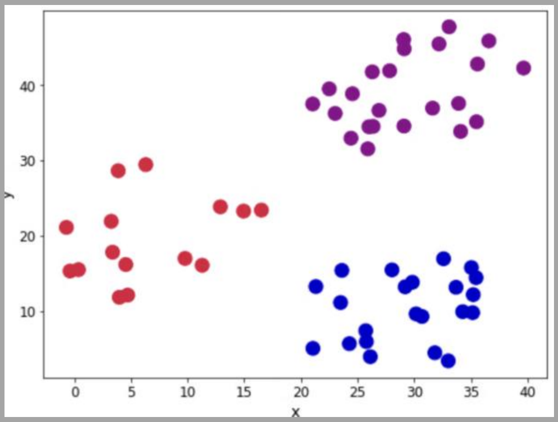
Check Column Importance :

S.No	Method Type	Method Name	Model Agnostic	Global / Local	Description
1	**Model-Based**	`feature_importances_` (e.g., Random Forest, XGBoost)	□	□	Importance based on split gain or impurity decrease (e.g., Gini, entropy).
2	**Model-Based**	Coefficients (`coef_`) — Linear/Logistic Regression	□	□	Use absolute value of coefficients to determine importance.
3	**Model-Based**	Coefficients — Linear SVM	□	□	Linear SVM assigns weights similar to linear models.
4	**Permutation**	Permutation Importance (`sklearn`)	□	□	Randomly shuffles each feature and observes drop in performance.
5	**Explainable AI**	SHAP (SHapley Additive exPlanations)	□	□ + □	Based on Shapley values from game theory. Explains both local & global importance.
6	**Explainable AI**	LIME (Local Interpretable Model-Agnostic Explanations)	□	□	Approximates the model locally with interpretable models. Good for local explanations.
7	**Statistical Tests**	Chi-Square Test	□	□	Measures dependency between categorical feature and target. Used in `SelectKBest`.
8	**Statistical Tests**	ANOVA F-test (`f_classif`)	□	□	Measures variance between classes for numeric features.
9	**Statistical Tests**	Mutual Information	□	□	Measures mutual dependency between features and target.
10	**Statistical Tests**	Pearson/Spearman/Kendall Correlation	□	□	Measures linear/nonlinear relationship between numeric features and target.
11	**Wrapper Method**	RFE (Recursive Feature Elimination)	□	□	Recursively removes least important features based on model weights.
12	**Tree Split Criteria**	Information Gain	□	□	Used in Decision Trees. Measures reduction in entropy after a split.
13	**Tree Split Criteria**	Gini Importance	□	□	Similar to Information Gain but based on Gini impurity.
14	**Regularization-Based**	Lasso (L1 Regularization)	□	□	Shrinks less important feature coefficients to zero. Good for sparse models.
15	**Embedded Selection**	Feature selection during training (e.g., ElasticNet, Decision Trees)	□	□	Model selects features inherently during training.

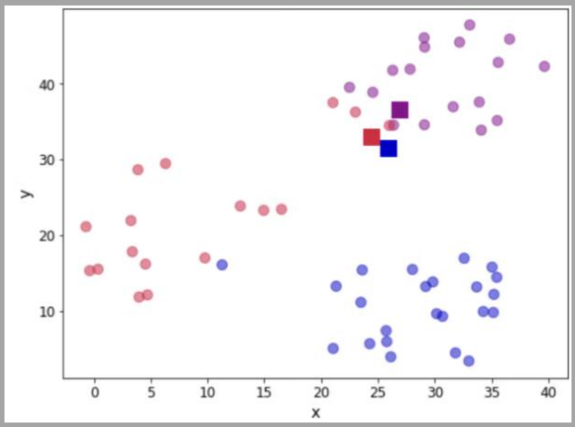
1. K Means Clustering : Viz , video

The k-means clustering algorithm is an unsupervised learning technique used to group unlabeled data into a predefined number, k of clusters based on their similarity. Here's what it does, step by step:

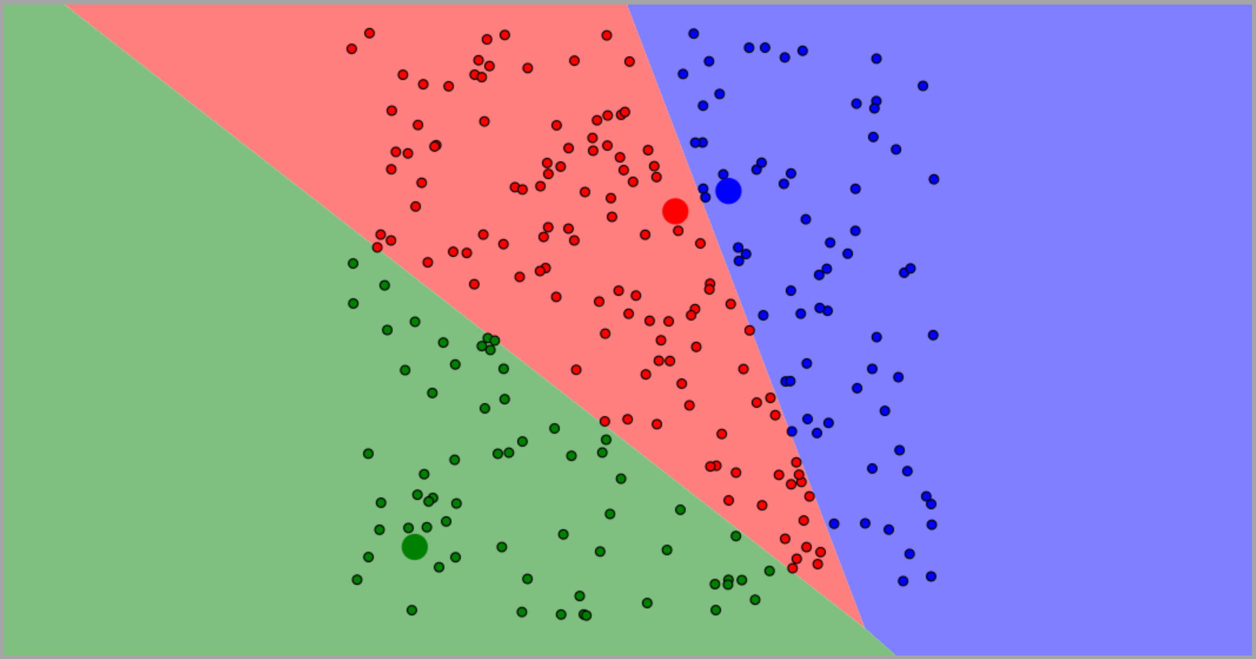
- Initializes k centroids: It randomly selects k central points (called centroids), which serve as the centers of the clusters.
- Assigns each data point to the nearest centroid: Each data point is assigned to the cluster whose centroid is closest (usually based on Euclidean distance).
- Updates centroids: For each cluster, the algorithm recalculates the centroid by taking the mean (average) position of all the points currently assigned to that cluster.
- Repeats assignment and update steps: These two steps (assignment and centroid update) repeat until the centroids do not move significantly, or until a maximum number of iterations is reached.



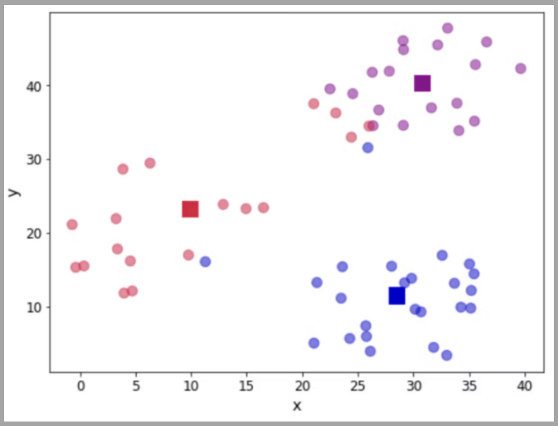
Data



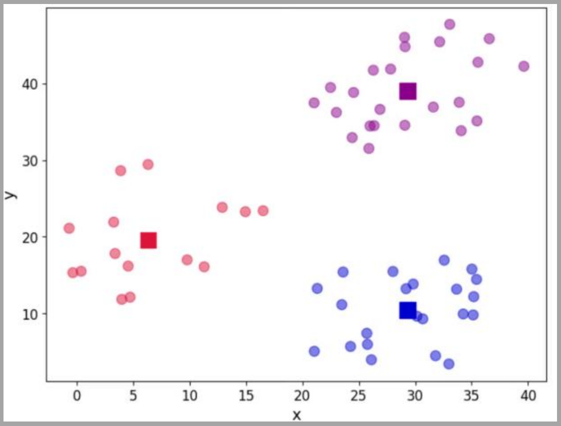
Define k and initiate the centroids



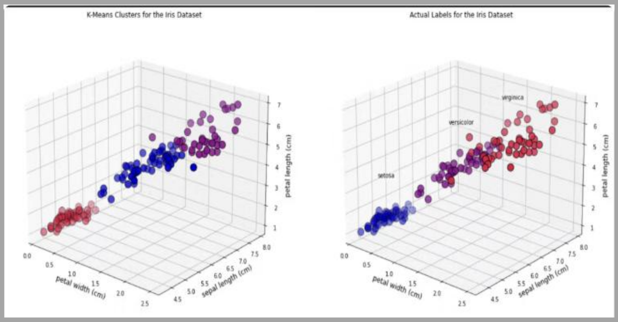
✓ How to Make cluster : calculate center between two centroid and draw perpendicular line.



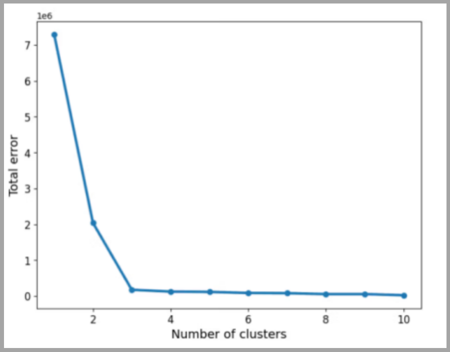
Calculate distance & Assign again centroids

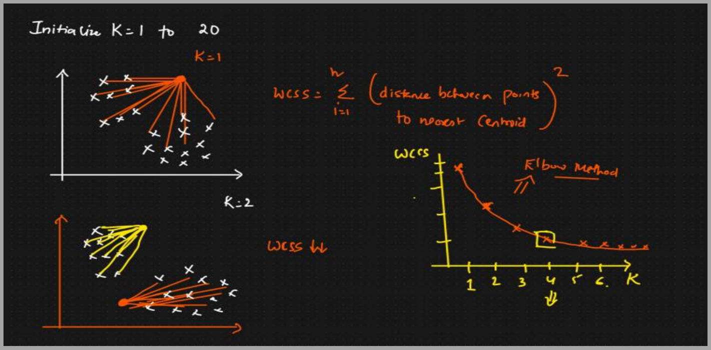


Update centroid location



- How to find K value (No. of Clusters) : Elbow method
    - Calculate wscc for each cluster (within cluster sum of squre)

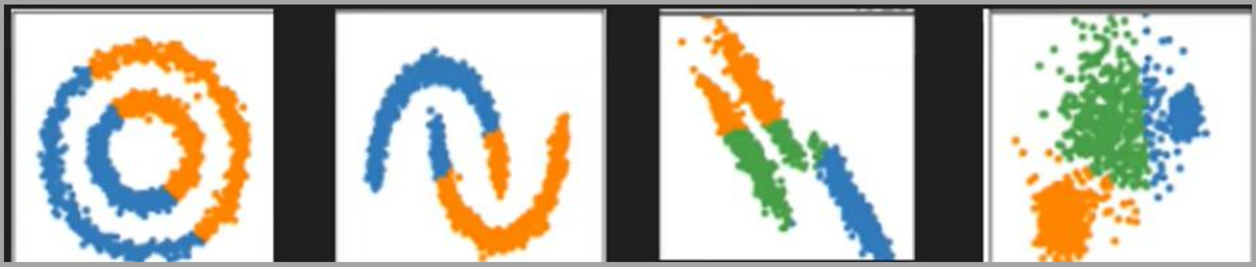



Initialize K=1 to 20
K=1
K=2
WCSS = $\sum_{i=1}^{n}$ (distance between points to nearest centroid)^2
WCSS
Elbow Method
WCSS ↓
K

2. Hierarichal Clustering :

- need if hierarchal clustering

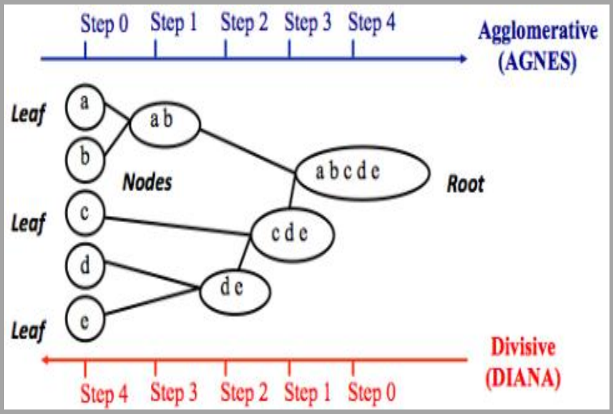
- k-Mean clustering is not work on given type of datsset,that why use this clustering.

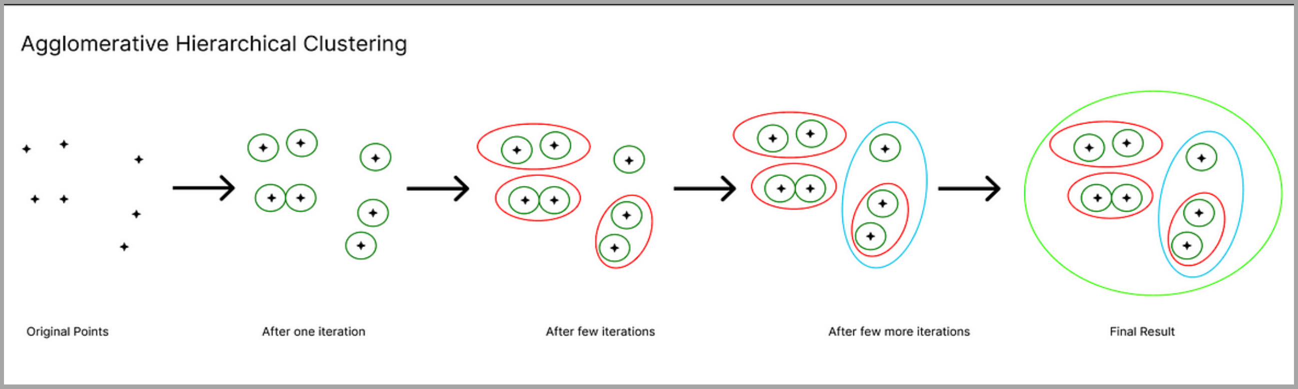


a. Agglomerative Hierarchical Clustering (Bottom-Up) : click

Input: Dataset D = {x_1, x_2, ..., x_n}, distance metric, linkage method
Output: Dendrogram of nested clusters

```
1. Initialize each data point as a separate cluster.
2. Compute initial distance matrix between all clusters.
3. While number of clusters > 1:
   a. Find the pair of clusters ($C_i$, $C_j$) with the smallest
      distance.
   b. Merge $C_i$ and $C_j$ to form a new cluster $C_k$.
   c. Update the distance matrix using the chosen
      linkage method.
4. Return the hierarchy of clusters.
```

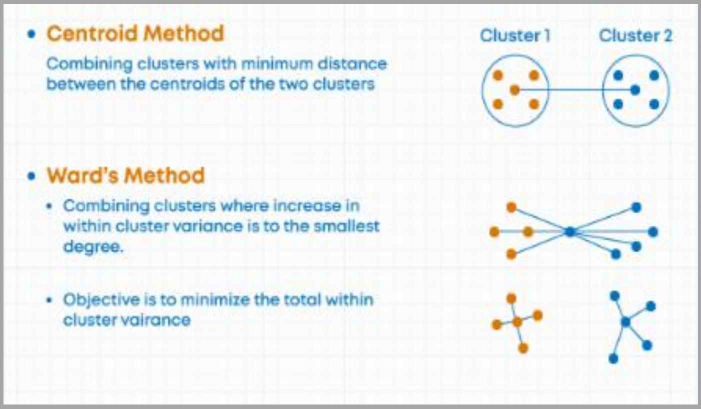




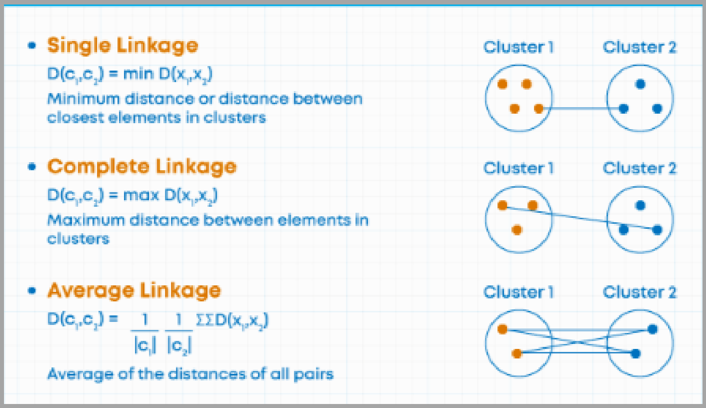
- **Types of agglomerative clustering base on distance calcuation :**

Linkage Type	**Distance Metric Used**	**Cluster Shape**	**Sensitivity**
Single Linkage	Min distance between points	Long, chained	Sensitive to noise
Complete Linkage	Max distance between points	Compact	Sensitive to outliers

Average Linkage	Average pairwise distance	Balanced	Moderate
Centroid Linkage	Distance between cluster centroids	Unstable (inversions)	Less commonly used
Ward's Method	Increase in SSE	Compact, equal-size	Least sensitive



- **Centroid Method**
  Combining clusters with minimum distance between the centroids of the two clusters
- **Ward's Method**
  - Combining clusters where increase in within cluster variance is to the smallest degree.
  - Objective is to minimize the total within cluster vairance



- **Single Linkage**
  $D(c_1,c_2) = \min D(x_1,x_2)$
  Minimum distance or distance between closest elements in clusters
- **Complete Linkage**
  $D(c_1,c_2) = \max D(x_1,x_2)$
  Maximum distance between elements in clusters
- **Average Linkage**
  $D(c_1,c_2) = \frac{1}{|c_1|}\frac{1}{|c_2|}\sum\sum D(x_1,x_2)$
  Average of the distances of all pairs

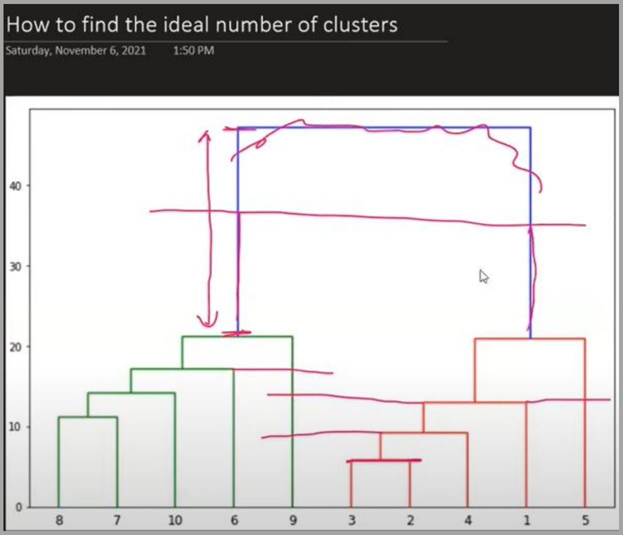
b. Divisive Hierarchical Clustering (Top-Down)

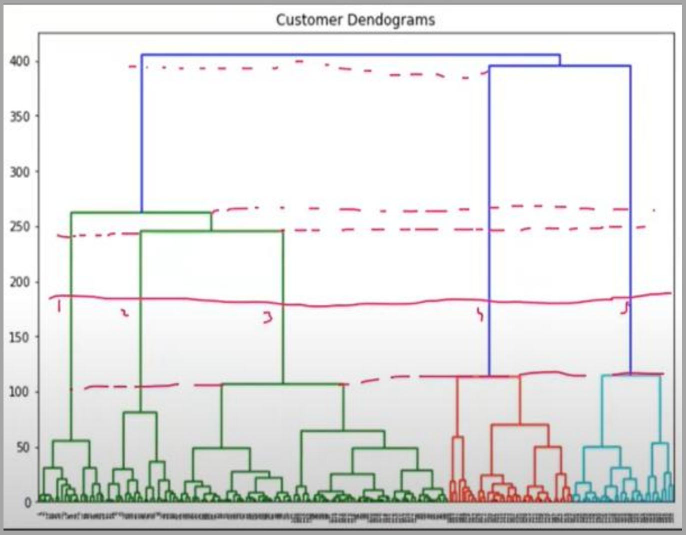
```
Input: Dataset D = {$x_1$, $x_2$, ..., $x_n$}
Output: Dendrogram of nested clusters

1. Initialize one cluster containing all data points.
2. While stopping criteria not met:
   a. Select the cluster to split (e.g., the largest or least cohesive).
   b. Apply a flat clustering algorithm (like K-Means with k=2) to split it.
   c. Replace the selected cluster with the two resulting clusters.
3. Return the hierarchy of clusters.
```

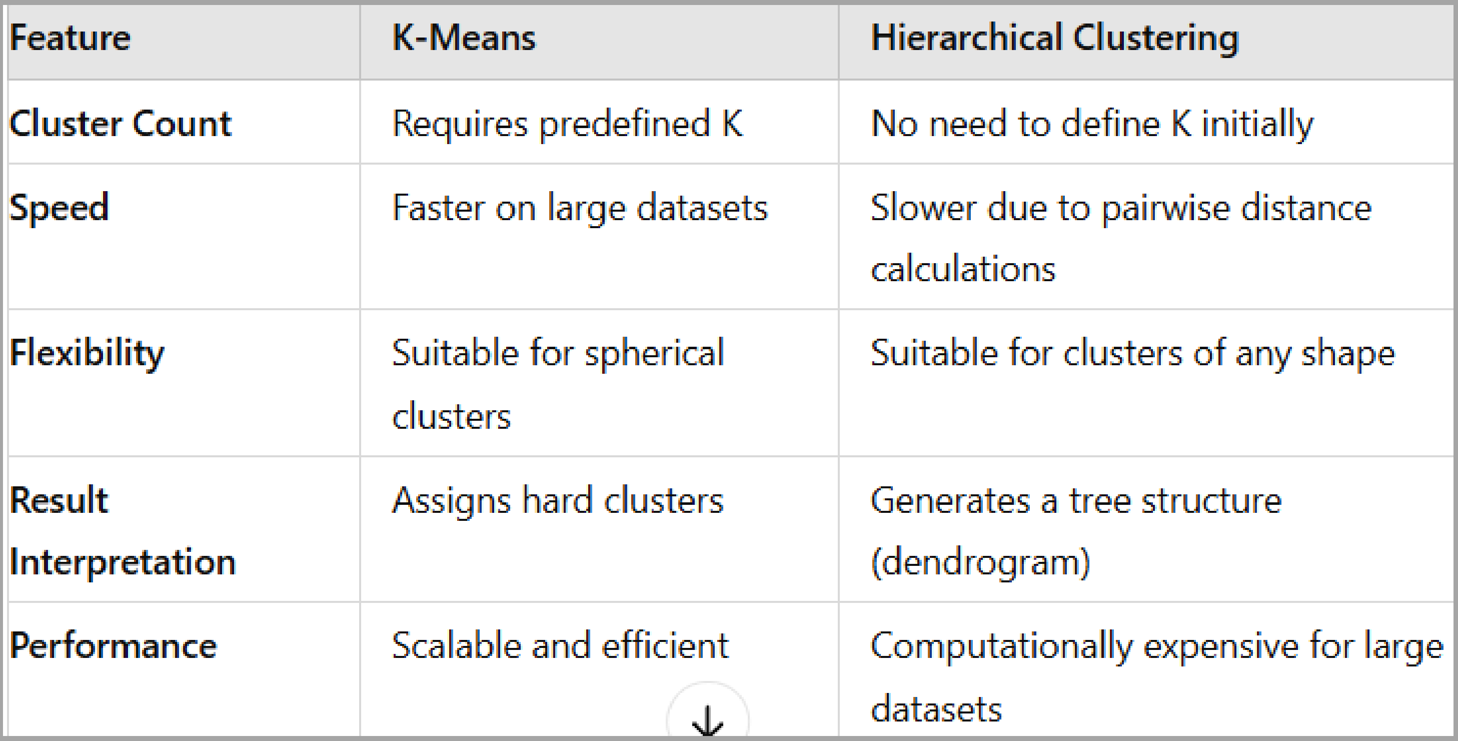
(Note : maximum time choose the agglomerative clustering)

❖ How to decide the number of clusters :

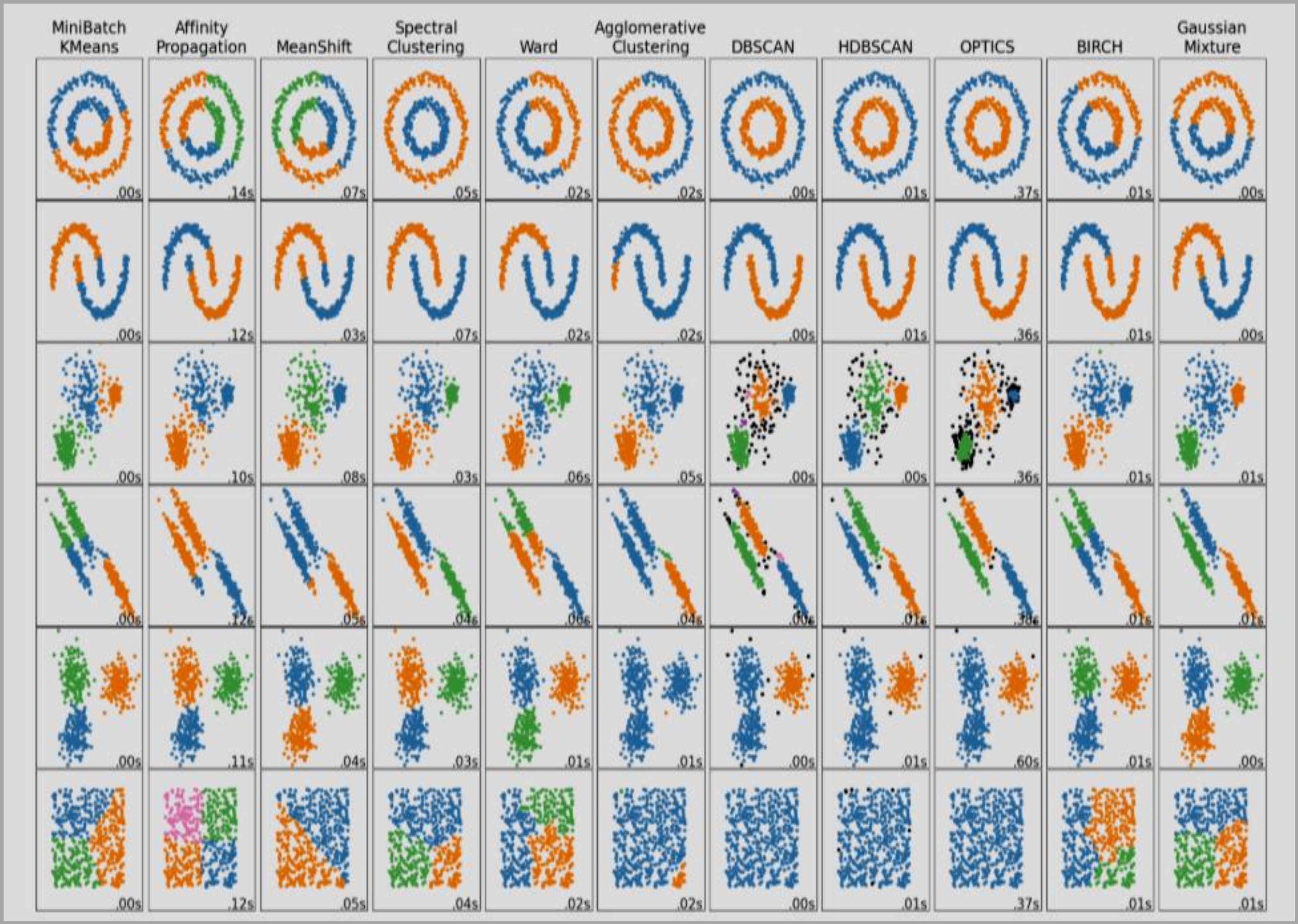
- Cut the dendrogram which max distance between horizontal line like given figures







Feature	K-Means	Hierarchical Clustering
Cluster Count	Requires predefined K	No need to define K initially
Speed	Faster on large datasets	Slower due to pairwise distance calculations
Flexibility	Suitable for spherical clusters	Suitable for clusters of any shape
Result Interpretation	Assigns hard clusters	Generates a tree structure (dendrogram)
Performance	Scalable and efficient	Computationally expensive for large datasets

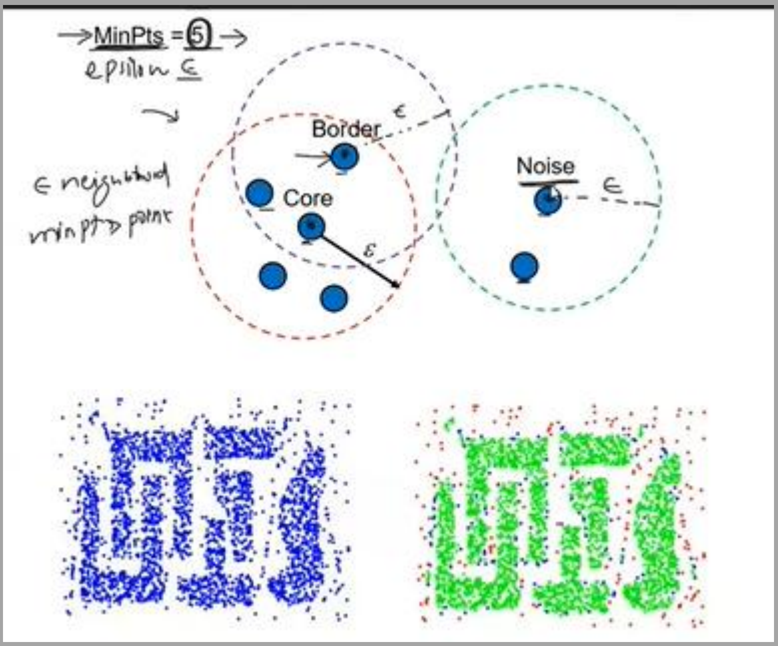


3. DBSCAN Clustering : Viz

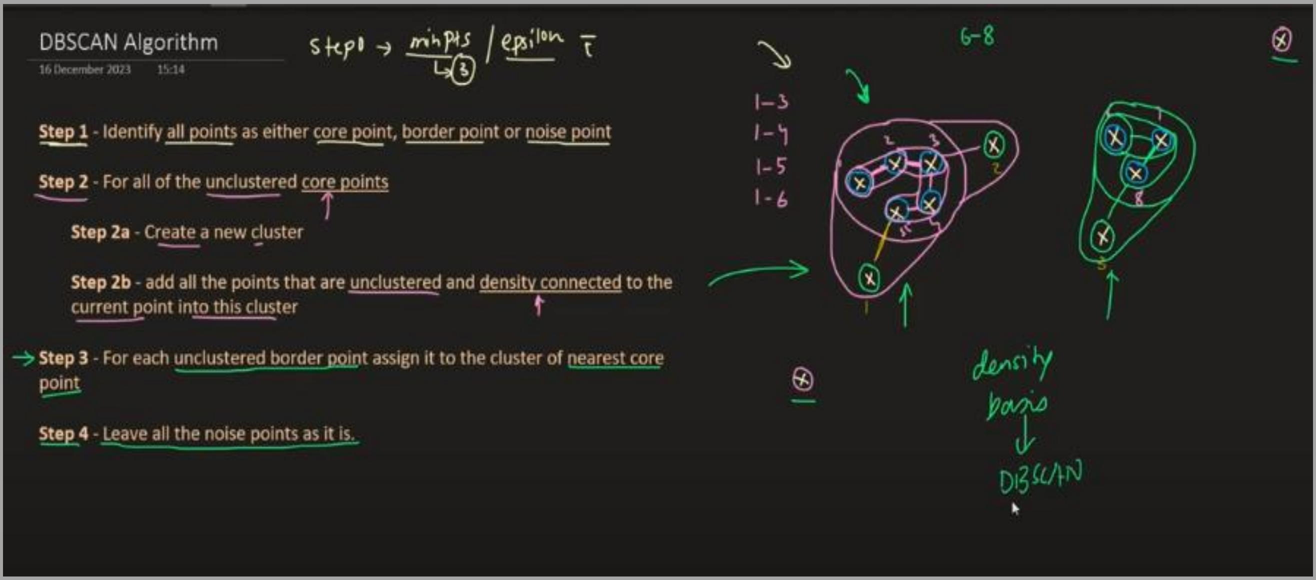
- DBSCAN stands for Density-Based Spatial Clustering of Applications with Noise. It is a powerful unsupervised clustering algorithm especially good at finding clusters of arbitrary shape and detecting outliers.9

➢ Types of points :

I. Core points
II. Boder points
III. Noise points



❖ **Algorithm :**



DBSCAN Algorithm

16 December 2023 15:14

Step 1 - Identify all points as either core point, border point or noise point

Step 2 - For all of the unclustered core points

Step 2a - Create a new cluster

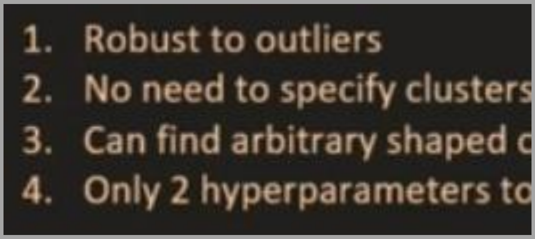
Step 2b - add all the points that are unclustered and density connected to the current point into this cluster

→ **Step 3** - For each unclustered border point assign it to the cluster of nearest core point

Step 4 - Leave all the noise points as it is.

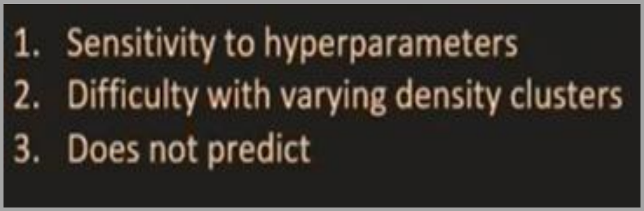
- **Limitation :**

Advantage



1. Robust to outliers
2. No need to specify clusters
3. Can find arbitrary shaped c
4. Only 2 hyperparameters to

Disadvantag



1. Sensitivity to hyperparameters
2. Difficulty with varying density clusters
3. Does not predict

🚀 Categories of Anomaly Detection Methods :

A. Isolation Forest :

- It works on the principle that **anomalies (outliers) are data points that are few and different** — and hence can be **isolated quickly**.
- It does this by **randomly selecting a feature** and then **randomly selecting a split value** between the minimum and maximum of that feature.
  - **Normal points** require **more splits** to isolate.
  - **Anomalies** are **isolated faster** (fewer splits), because they're far from other points

How It Works (Step-by-Step):

1. **Build isolation trees** (random trees):
   - For each tree:
     - Randomly pick a feature.
     - Randomly pick a split value between min and max of that feature.
     - Split the data recursively until:
       - Only one instance is left, or
       - Max depth is reached.
2. **Compute average path length** from root to the leaf for each data point across all trees.

3. **Anomaly score** is calculated:

$$s(x,n) = 2^{-\frac{E(h(x))}{c(n)}}$$

Where:

   - h(x) = path length of point xxx
   - E(h(x)) = expected path length
   - c(n) = average path length for unsuccessful search in a Binary Search Tree

4. **Interpret the score**:
   - **E(h(x)) << h(x)** → **S(x,n)** ≈ Closer to **1** → **outlier**
   - **E(h(x)) >> h(x)** → **S(x,n)** ≈ Closer to **0** → **normal**.

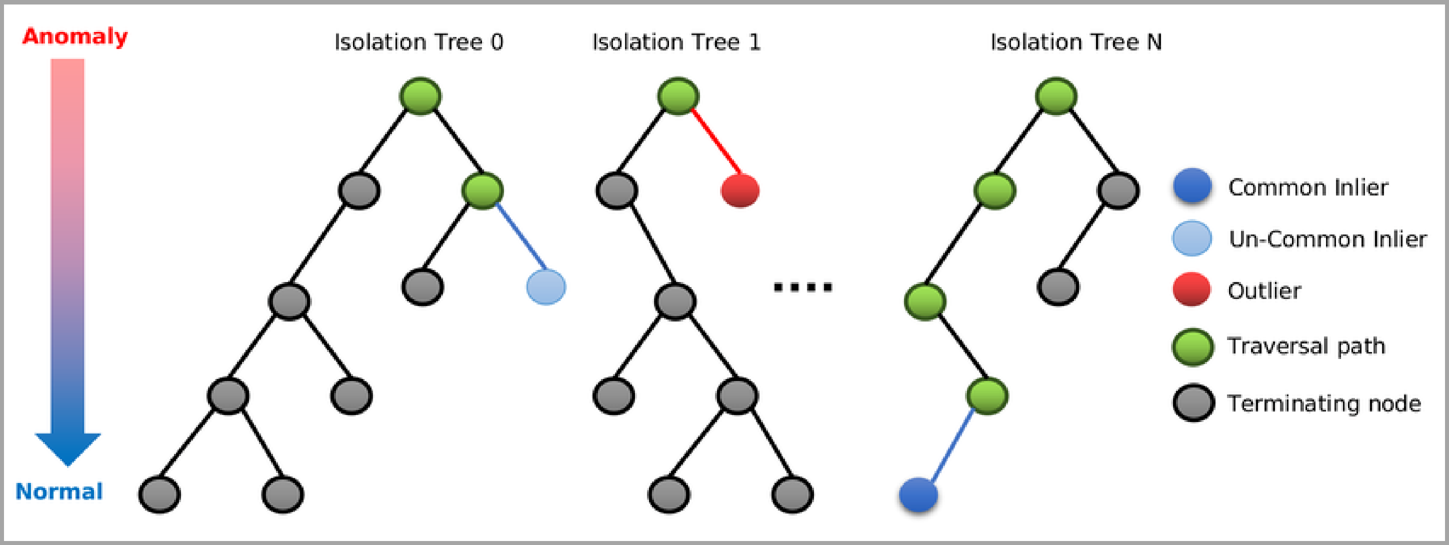
Advantages:

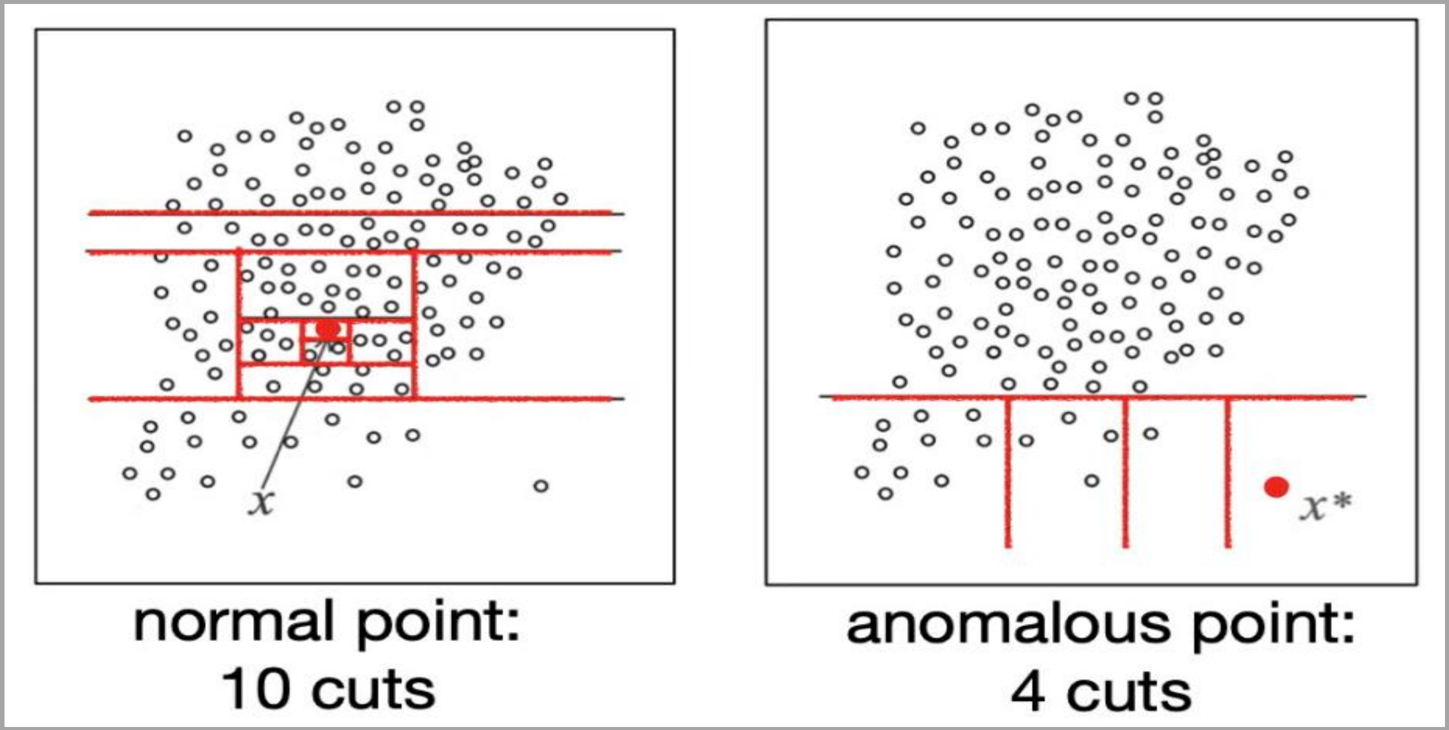
- Works well on **high-dimensional data**.
- **Fast** and scalable (linear time complexity).
- **No assumptions** about data distribution.

Use Case Examples:

- Fraud detection in finance.
- Intrusion detection in network traffic.
- Sensor fault detection in IoT.
- Health care

```python
from sklearn.ensemble import IsolationForest

model = IsolationForest(n_estimators=100, contamination=0.1, random_state=42)
model.fit(X_train)

# Predict: -1 = anomaly, 1 = normal
predictions = model.predict(X_test)
```


Anomaly
Normal
Isolation Tree 0
Isolation Tree 1
Isolation Tree N
Common Inlier
Un-Common Inlier
Outlier
Traversal path
Terminating node


x
x*
normal point:
10 cuts
anomalous point:
4 cuts

B. DBSCAN Anomaly Detection :

```
import numpy as np
from sklearn.datasets import make_blobs
from sklearn.cluster import DBSCAN
# Generate the data
X, y = make_blobs(n_samples=1000, centers=1, cluster_std=4,
random_state=123)
# Define the DBSCAN parameters
eps = 3
min_samples = 5
# Create the DBSCAN model
dbscan = DBSCAN(eps=eps, min_samples=min_samples)
# Fit the model to the data
dbscan.fit(X)
# Get the labels of the data points
labels = dbscan.labels_
# Identify the outliers
outliers = np.where(labels == -1)[0]
# Print the number of outliers
print("Number of outliers:", len(outliers))
# Plot the data with the outliers highlighted
plt.scatter(X[:, 0], X[:, 1], c=labels)
plt.scatter(X[outliers, 0], X[outliers, 1], c="red", marker="x")
plt.show()
```

- It is a clustering algorithm that can also be used for anomaly (outlier) detection by identifying low-density regions as anomalies.

- DBSCAN groups closely packed points into **clusters**, and labels points that lie alone in **low-density regions as outliers** (anomalies).

- Eps : Maximum distance between two samples to be considered as neighbors.
- min_samples : Minimum number of points to form a **dense region (cluster)**.

Steps:

1. Choose a point.
2. If it has at least **min_samples** within **eps** radius, it's a core point.
3. Expand the cluster from core points by adding all density-reachable points.
4. Points not reachable from any core point are labeled as noise (outliers).

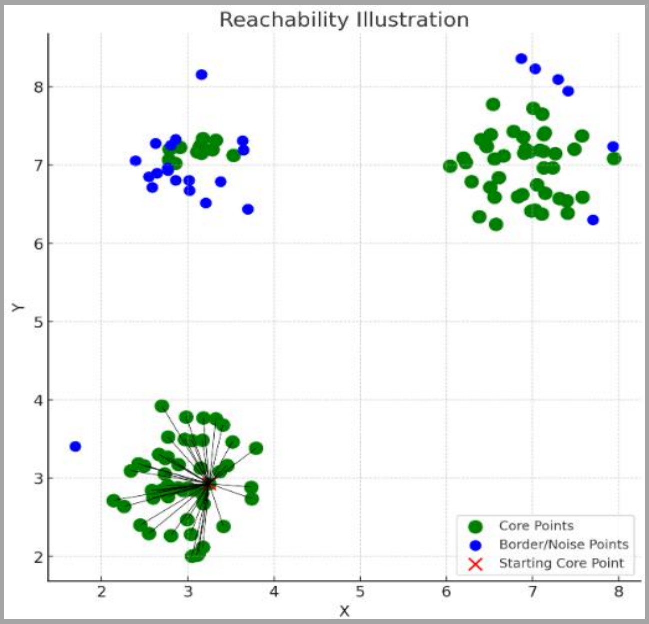
5.

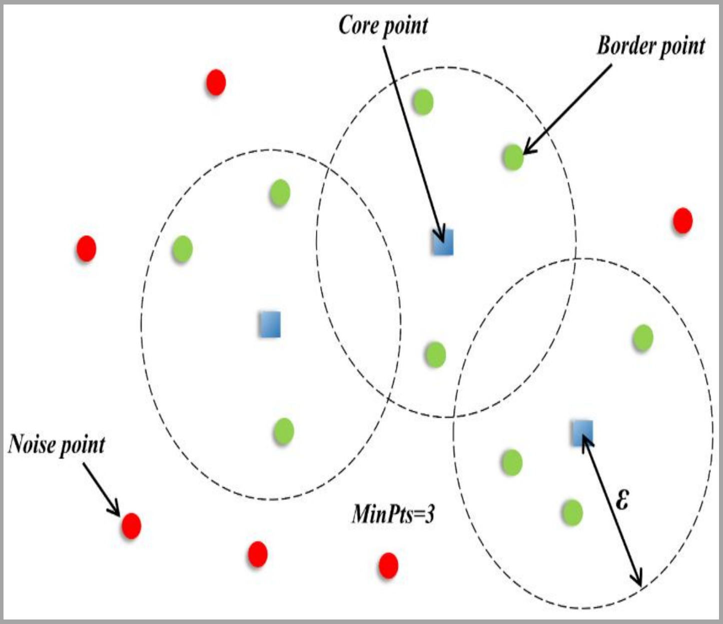
? Advantages:

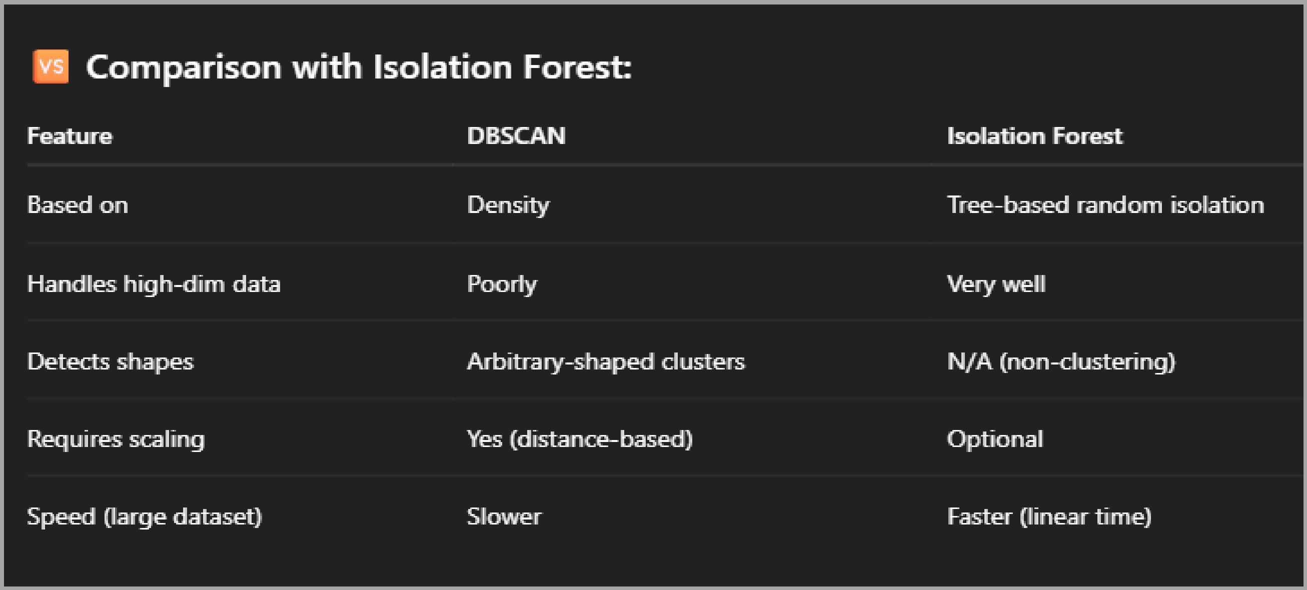
A. No need to specify number of clusters.
B. Can detect arbitrary shaped clusters.
C. Naturally detects outliers.

□ Limitations:

- Sensitive to choice of `eps` and `min_samples`.
- Does not scale well with high-dimensional data.







VS Comparison with Isolation Forest:

Feature	DBSCAN	Isolation Forest
Based on	Density	Tree-based random isolation
Handles high-dim data	Poorly	Very well
Detects shapes	Arbitrary-shaped clusters	N/A (non-clustering)
Requires scaling	Yes (distance-based)	Optional
Speed (large dataset)	Slower	Faster (linear time)

C. Local Outlier Factor Anamoly Detection :

```
from sklearn.neighbors import LocalOutlierFactor
clf = LocalOutlierFactor(n_neighbors=20, contamination=0.1)
```

- Local Outlier Factor (LOF) is a density-based anomaly detection algorithm. It identifies data points that are significantly less dense than their neighbors in other words, outliers.
- LOF compares the local density of a point to that of its k nearest neighbors.
  - If a point's density is much lower than that of its neighbors, it is likely an anomaly.
  - LOF assigns an anomaly score — higher means more likely to be an outlier.

How LOF Works:

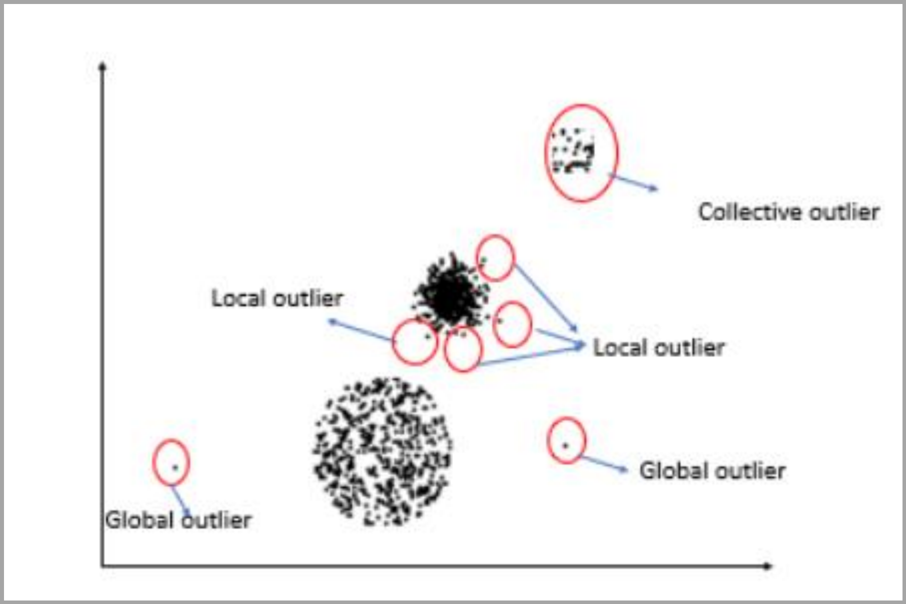
Given:

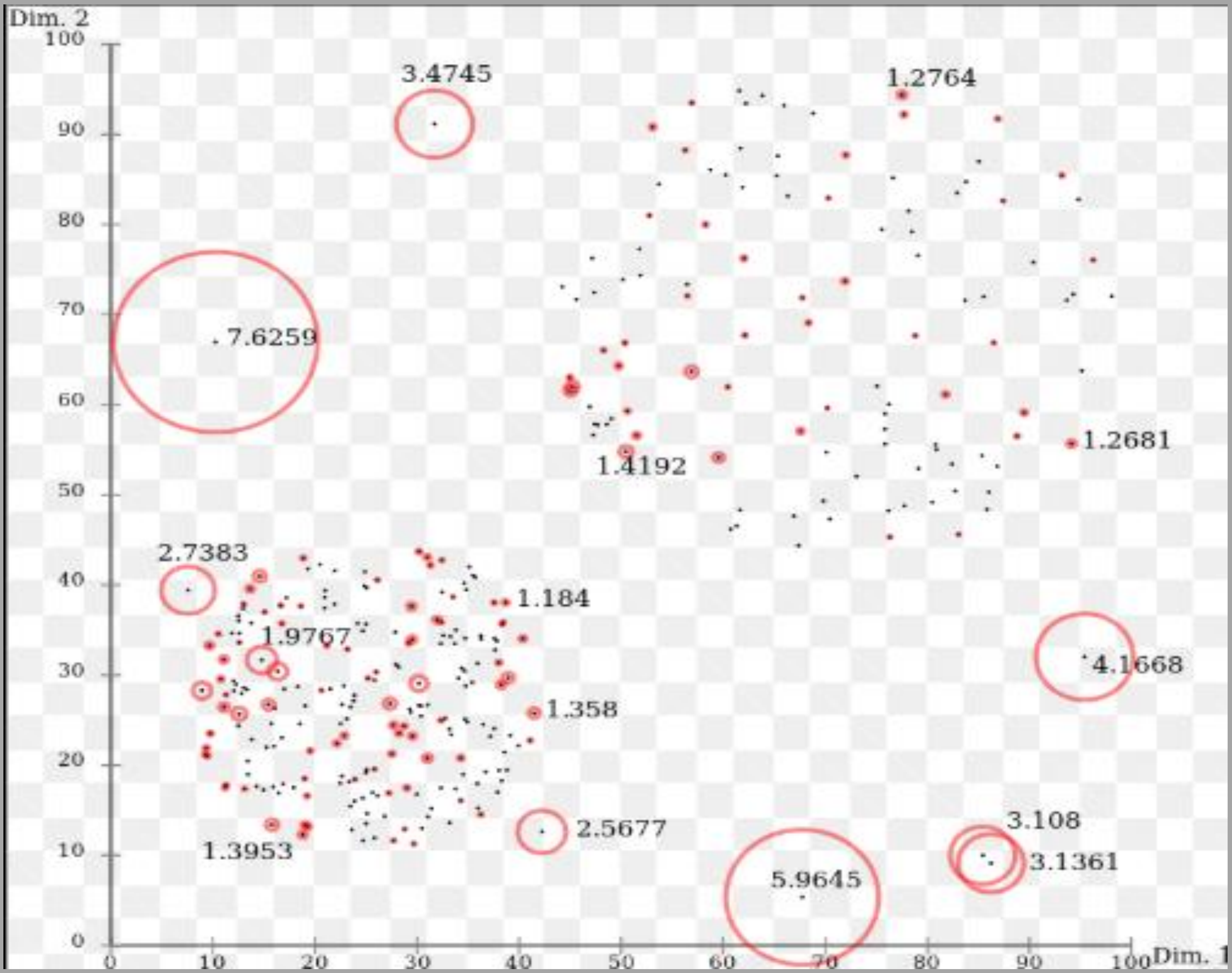
- kkk = number of nearest neighbors (a parameter)

1. Compute k-distance:
   - Distance from a point to its kthk^\text{th}kth nearest neighbor.
2. Find k-neighbors:
   - All points within the k-distance.

Advantages:

- Captures local density variations .
- Works well when data has clusters of different densities.


Collective outlier
Local outlier
Local outlier
Global outlier
Global outlier


Dim. 2
Dim. 1
3.4745
1.2764
7.6259
1.4192
1.2681
2.7383
1.184
1.9767
1.358
4.1668
2.5677
1.3953
3.108
3.1361
5.9645

============ 🡸 Reusable code 🡺 ============

Finetune the Hyperparameter :

Step 1: Import necessary libraries

```
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.ensemble import RandomForestRegressor, AdaBoostRegressor --  or more
from xgboost import XGBRegressor
from catboost import CatBoostRegressor
import numpy as np
```

Step 2: Define Models & Hyperparameter Grids

```
params = {
    # LinearRegression rarely needs param tuning; can keep empty or add 'fit_intercept'
    "Linear Regression": {
        'fit_intercept': [True, False]
    },
    "Lasso": {
        'alpha': [0.001, 0.01, 0.1, 1, 10],
        'selection': ['cyclic', 'random']
    },
    "Ridge": {
        'alpha': [0.01, 0.1, 1, 10, 100],
        'solver': ['svd', 'cholesky', 'lsqr']
    },
    "K-Neighbors Regressor": {
        'n_neighbors': [3, 5, 7, 9],
        'weights': ['uniform', 'distance'],
        'p': [1, 2]  # p=1: Manhattan, p=2: Euclidean distance
    },
    "Decision Tree": {
        'max_depth': [None, 5, 10, 15],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4]
    },
    "AdaBoost Regressor": {
        'n_estimators': [50, 100, 200],
        'learning_rate': [0.01, 0.1, 1]
    },
    "GradientBoosting": {
        'n_estimators': [100, 200],
        'learning_rate': [0.01, 0.1],
        'max_depth': [3, 5, 7]
    },
    "Random Forest Regressor": {
        'n_estimators': [100, 200],
        'max_depth': [None, 10, 20],
        'min_samples_split': [2, 5],
        'min_samples_leaf': [1, 2]
    },
    "XGBRegressor": {
        'n_estimators': [100, 200],
        'learning_rate': [0.01, 0.1],
        'max_depth': [3, 5, 7]
    },
    "CatBoosting Regressor": {
        'depth': [4, 6, 8],
        'learning_rate': [0.01, 0.1],
        'iterations': [500, 1000],
        'l2_leaf_reg': [1, 3, 5]
    }
}
```

Step 3: Load and split your data

```
data = load_iris()
X = data.data
y = data.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

Step 4: Run GridSearchCV for Each Model

```
best_models = {}
for name, config in params.items():
    print(f"Running GridSearchCV for {name}...")
    grid = GridSearchCV(
        estimator=config['model'],
        param_grid=config['params'],
        cv=3,
        scoring='neg_mean_squared_error',
        n_jobs=-1,
        verbose=2
    )
    grid.fit(X_train, y_train)
    best_models[name] = {
        'best_model': grid.best_estimator_,
        'best_score': -grid.best_score_,
        'best_params': grid.best_params_
}
```

Or

Step 5:Use RandomizedSearchCV Instead of GridSearchCV (if dataset is large)

```
random_search = RandomizedSearchCV(
    estimator=model,
    param_distributions=param_grid,
    n_iter=20,   # Only test 20 combinations
    scoring='neg_mean_squared_error',
    cv=3,
    n_jobs=-1,
    random_state=42,
    verbose=2
)
```

Step 5: Evaluate Best Models on Test Set

```
for name, result in best_models.items():
    y_pred = result['best_model'].predict(X_test)
    test_mse = mean_squared_error(y_test, y_pred)
    print(f"\n{name} Regressor:")
    print(f"  Best Params: {result['best_params']}")
    print(f"  Train MSE (CV): {result['best_score']:.4f}")
    print(f"  Test MSE: {test_mse:.4f}")
```