**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 :**

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)

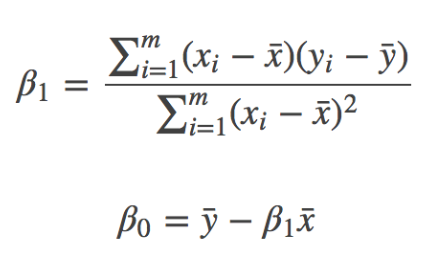
# Step 3: Predict

prediction = model.predict([[6]])

print("Prediction for x=6:", prediction)

* **Intuition :**

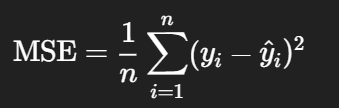
1. Imagine plotting points of data on a 2D graph.
2. You want to draw a straight line that best fits the data.
3. The line should minimize the vertical distance (error) between the actual data points and the predicted values on the line.
4. In multiple dimensions, this becomes a hyperplane instead of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

* **Formula :** 
  + **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

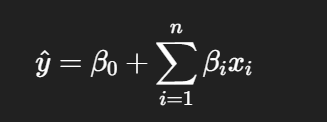
and actual value)

* **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):

****

1. After training, predictions are made using:

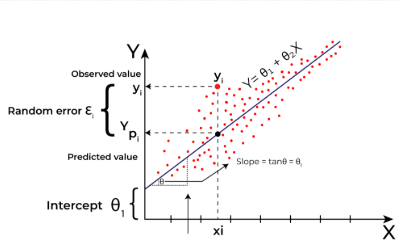
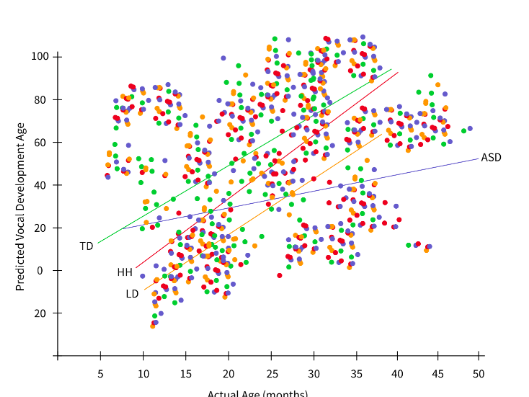
****

## ****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



Simple Multiple

* **From Scratch**

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)

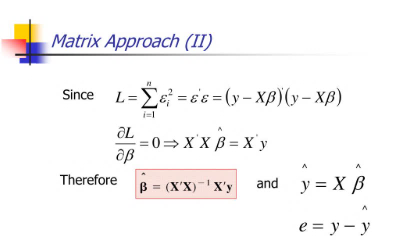
        self.intercept\_ = betas[0]

        self.coef\_ = betas[1:]

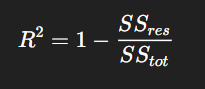
    def predict(self,X\_test):

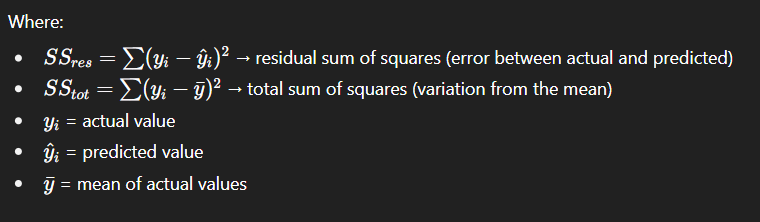
        y\_pred = np.dot(X\_test,self.coef\_) + self.intercept\_

        return y\_pred



* **R2 Score :**
* **Definition :** The R² 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 :**

****

1. **Intuition:**

* R² = 0.9 means 90% of the variation in the target can be explained by the model.
* R² = 0.0 means the model does no better than predicting the average.
* R² < 0.0 means the model is worse than just predicting the mean for every value.

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)

1. **Advantages:**

* Easy to interpret
* Widely used in regression evaluation
* Shows how well the model explains the variance

1. **Limitations:**

* Can bemisleading if used alone
* Doesn’t indicate whether predictions are biased
* Can be negative (confusing for beginners)
* Not useful for comparing different datasets

1. **When to Use :**

* Use R² 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.

1. **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

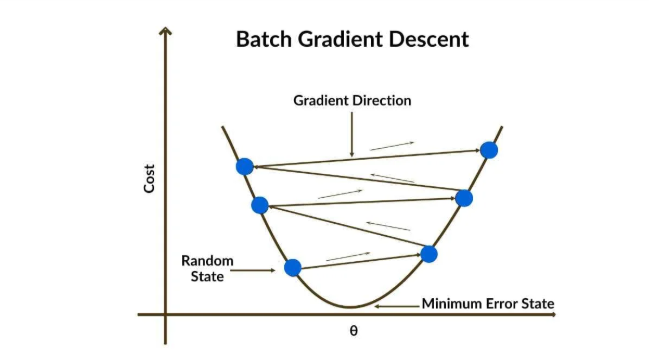
            self.m = self.m - self.lr \* slope\_m

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

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

    def predict(self, X\_test):

        return np.dot(X\_test, self.m) + self.b



1. 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

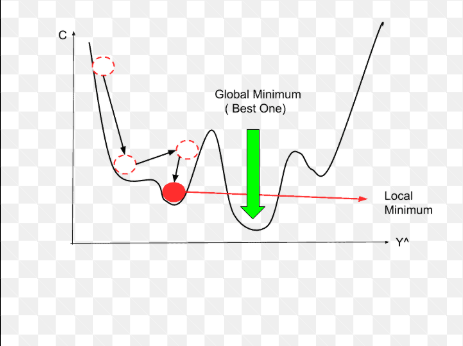
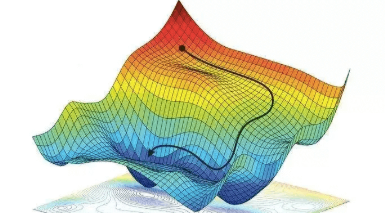
                self.m -= self.lr \* slope\_m

    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

** **

1. 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

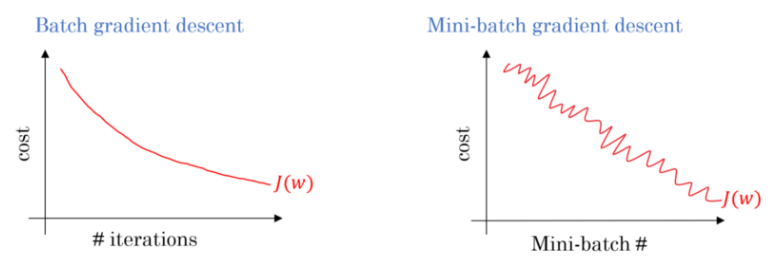
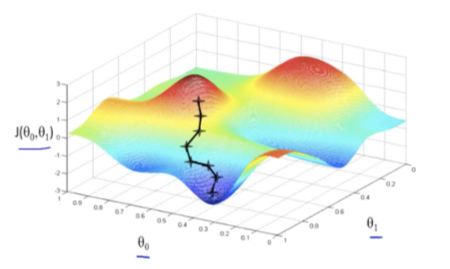
                self.m -= self.lr \* slope\_m

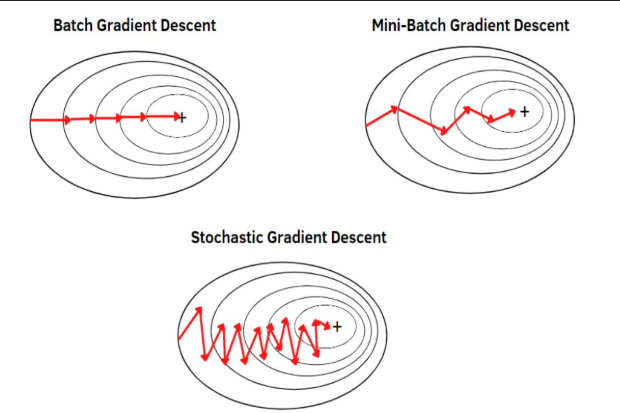
    def predict(self, X\_test):

        if X\_test.ndim == 1:

            X\_test = X\_test.reshape(-1, 1)

    return np.dot(X\_test, self.m) + self.b

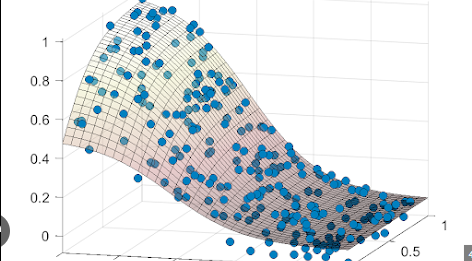
 

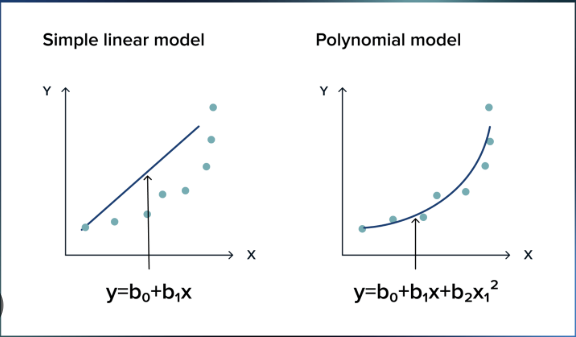




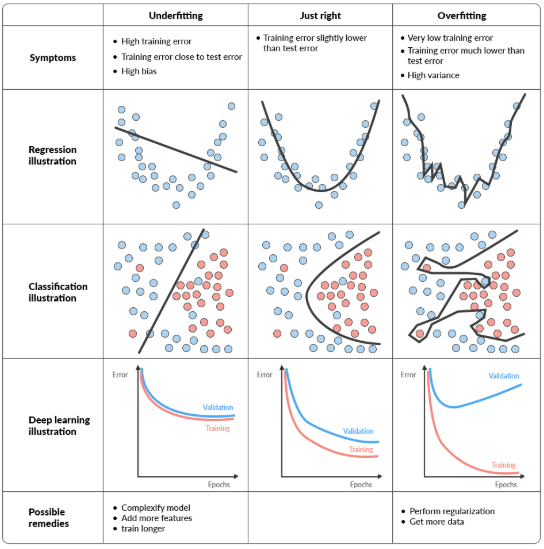
| **Type** | **Description** | **Use Case / When to Use** | **Pros** | **Cons** |
| --- | --- | --- | --- | --- |
| **Batch Gradient Descent** | Computes the gradient of the cost function using **entire dataset** for each step. | When dataset is **small to medium-sized** and can fit in memory. | - Converges 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) |

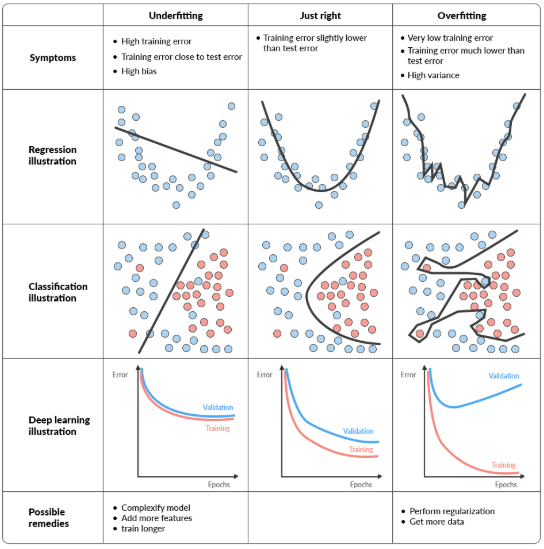
Polyniomial Regession **:**



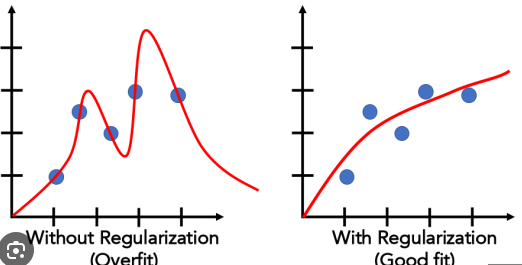


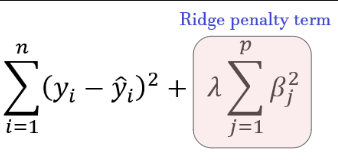
* **Bias Varience tradeoff :**

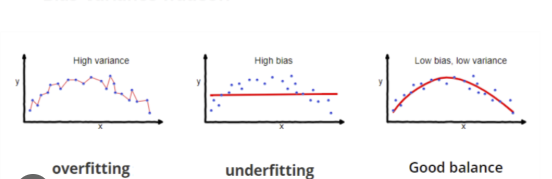


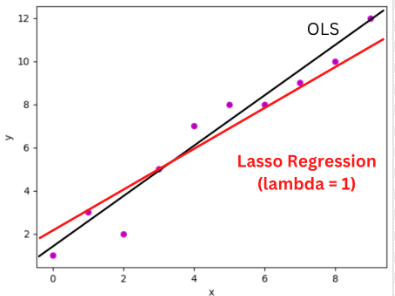
****

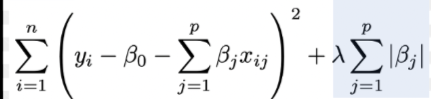
Regularization

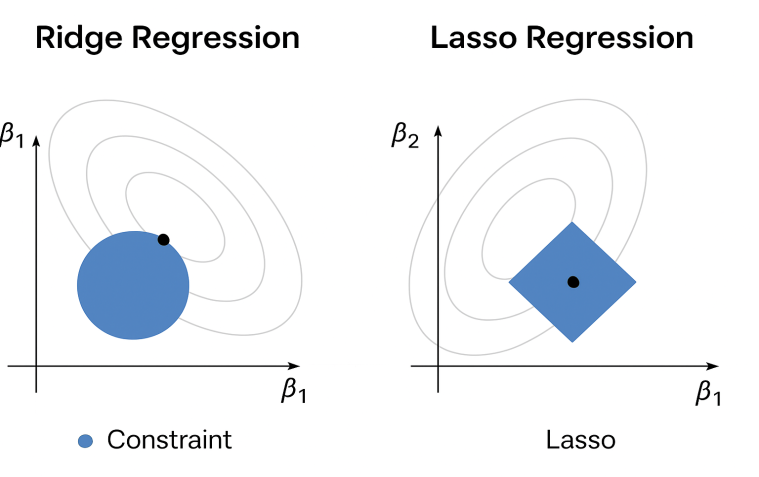
1. **Ridge Regression :**



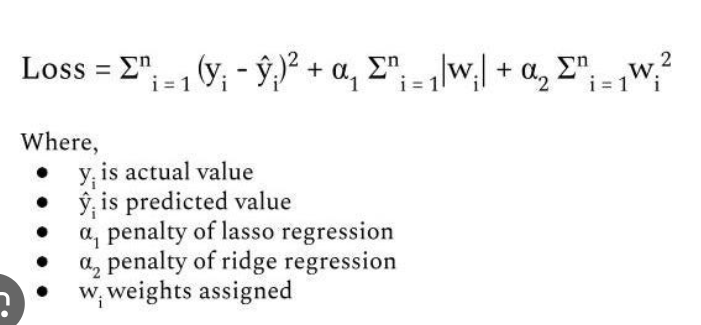
****

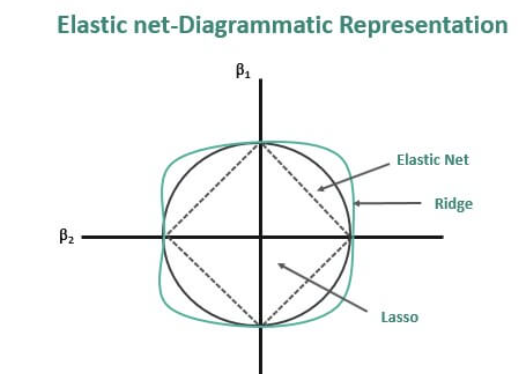
1. **Lasso Regression :** 



****

ElasticNet :





| **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

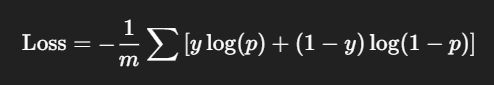


Z = w1​x1​ + w2​x2​ + ..... + wn​xn​ + b

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

1. **Steps in Logistic Regression :** 
   1. **Initialize weights** (**w**) and bias (**b**).
   2. **Linear combination**: Calculate **z** for each sample.
   3. **Sigmoid**: Convert **z** to probability p.
   4. **Loss Function**: Use **Binary Cross-Entropy Loss**:
   5. Gradient Descent: Adjust **w** and **b** to minimize the loss.

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

****

1. **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)

1. **Parameters :**

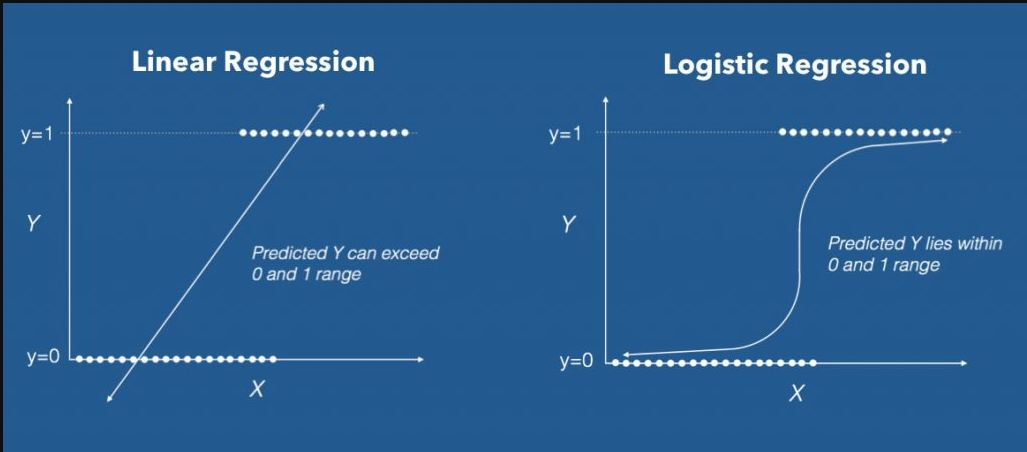
| **Parameter** | **Description** |
| --- | --- |
| penalty | Type of regularization ('l1', 'l2', 'elasticnet', 'none'). |
| C | Inverse of regularization strength. Smaller values = stronger regularization. |
| solver | Algorithm to use ('liblinear', 'saga', 'newton-cg', 'lbfgs'). |
| max\_iter | Maximum number of iterations to converge. |
| multi\_class | 'auto', 'ovr' (One-vs-Rest) or 'multinomial'. |
| random\_state | Controls randomness for reproducibility. |

1. **Advantage :**

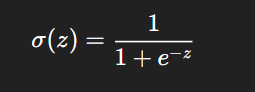
- Simple and interpretable.  
- Works well for binary classification.  
- Outputs probabilities (not just class labels).  
- Efficient on large datasets.

1. **Limitation :**

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



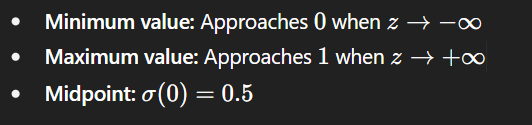
Sigmoid Function :

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 :**

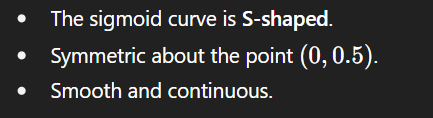
Where:

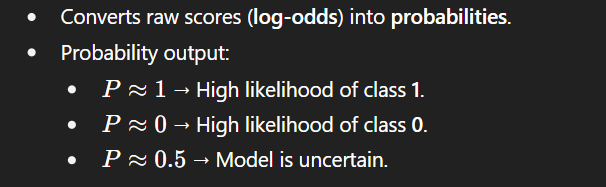
* z = input (can be any real number)
* e = Euler’s number (≈2.718\approx 2.718≈2.718)

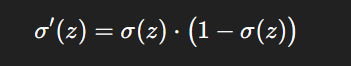
1. **Output Range :**

****

1. **Shape :**

****

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

****

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

1. **Advantage :**

✅ Smoothly maps inputs to (0,1).

✅ Useful for probabilistic interpretation.

✅ Differentiable everywhere (good for optimization).

1. **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).

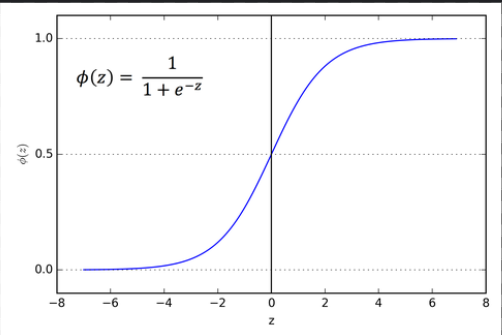
This can cause zig-zag movement in gradient descent, making optimization slower.

**** **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.



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 | R² 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])

1. Center the data (subtract the mean)

*Make sure each feature has a mean of zero by subtracting its mean from every value.*

1. Compute the covariance matrix

*Calculate how much the features vary together (covariance) to understand their relationships.*

1. 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).*

1. Sort eigen vectors by eigen values in descending order

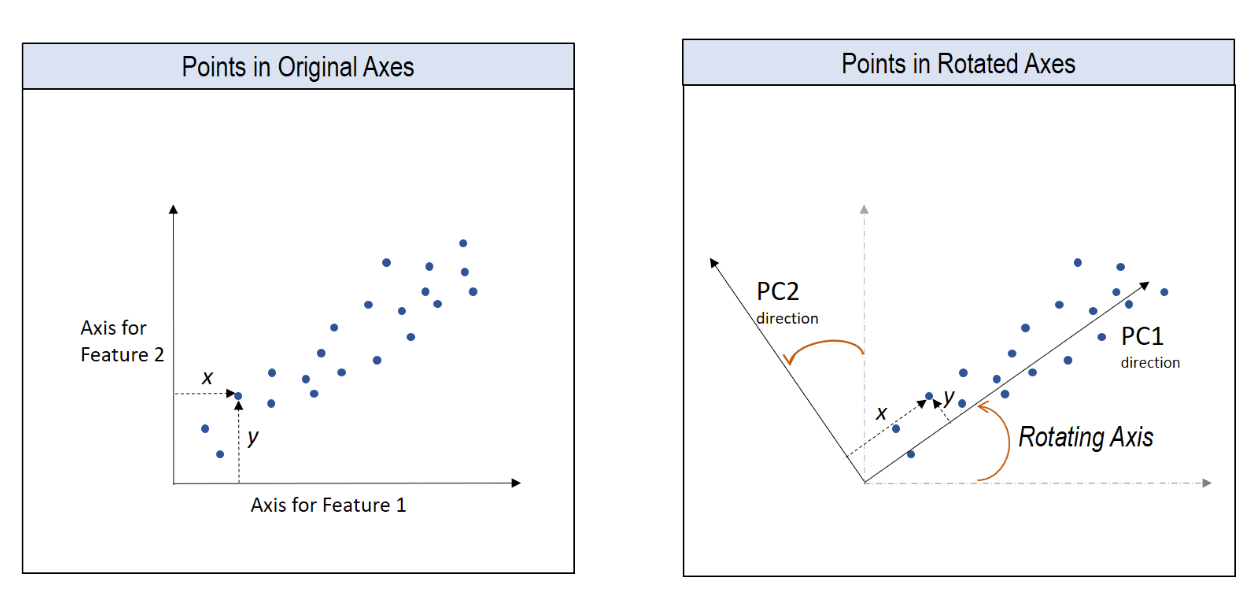
*Rank the components by the amount of variance they explain (most important first).*

1. Select the top k principal components

*Choose the number of components (k) that capture most of the variance (information).*

1. 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.*

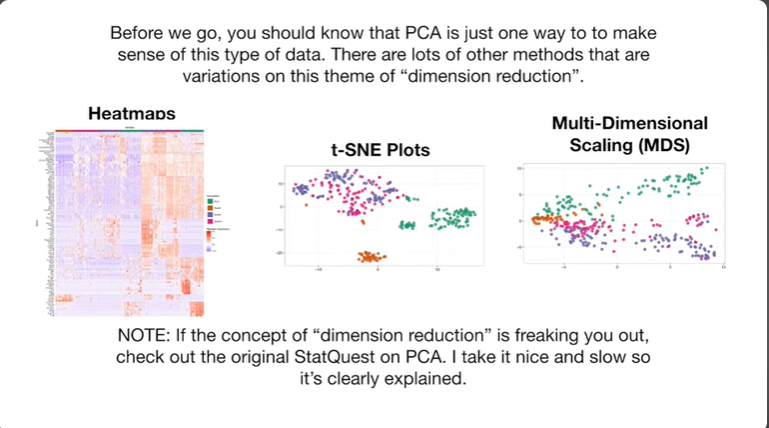


scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

pca = PCA(n\_components=2) # reduce to 2 dimensions

X\_pca = pca.fit\_transform(X\_scaled)



from sklearn.decomposition import PCA

pca = PCA(

    n\_components=None,

copy=True,

  whiten=False,

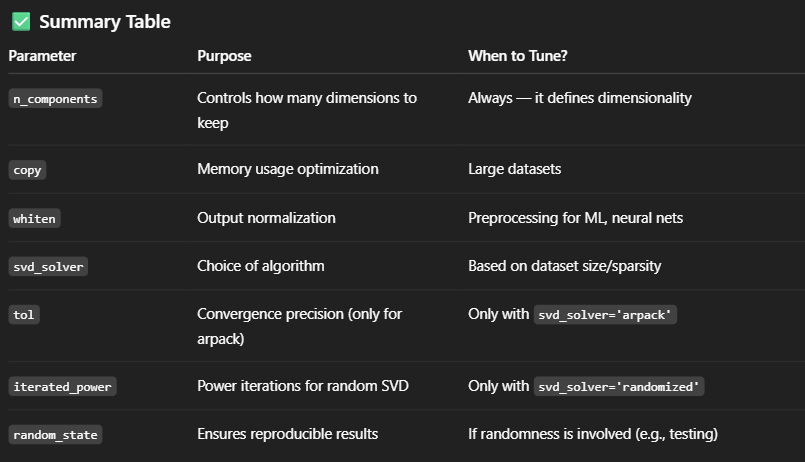
svd\_solver='auto',

    tol=0.0,

iterated\_power='auto',

    random\_state=None

)



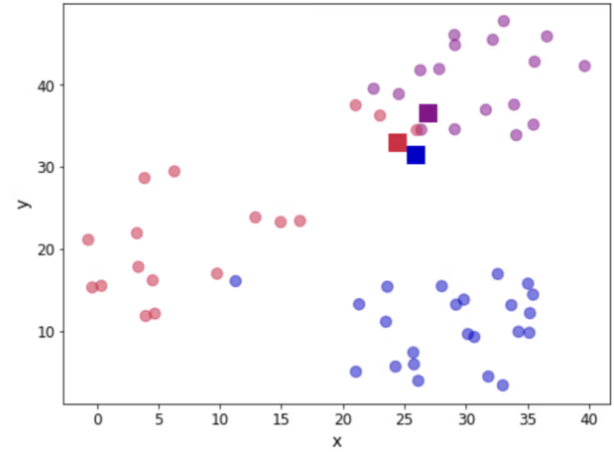
Check Column Importance :

| **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. |

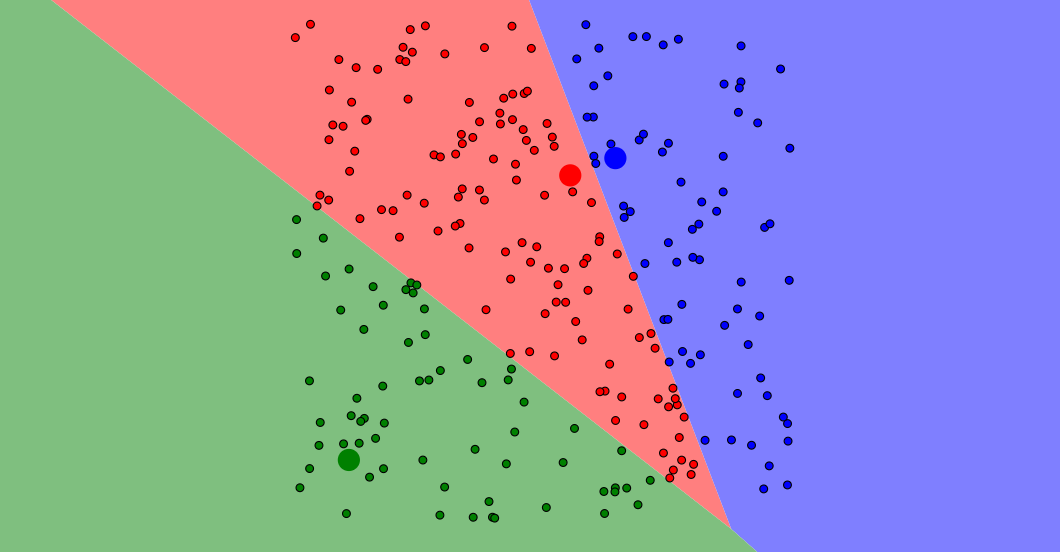
1. **K Means Clustering :** [Viz](https://www.naftaliharris.com/blog/visualizing-k-means-clustering/) , [video](https://www.youtube.com/watch?v=EItlUEPCIzM)

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:

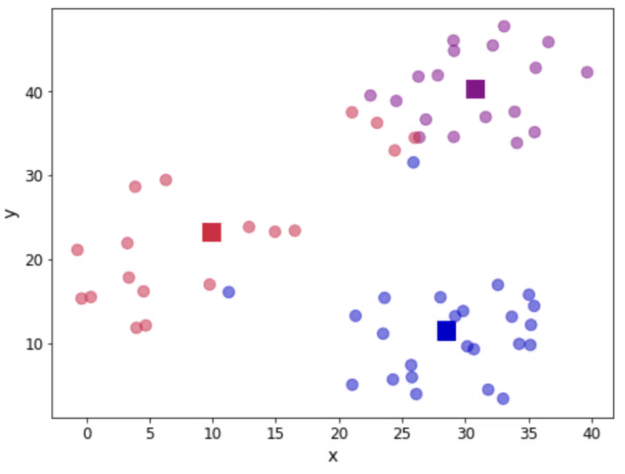
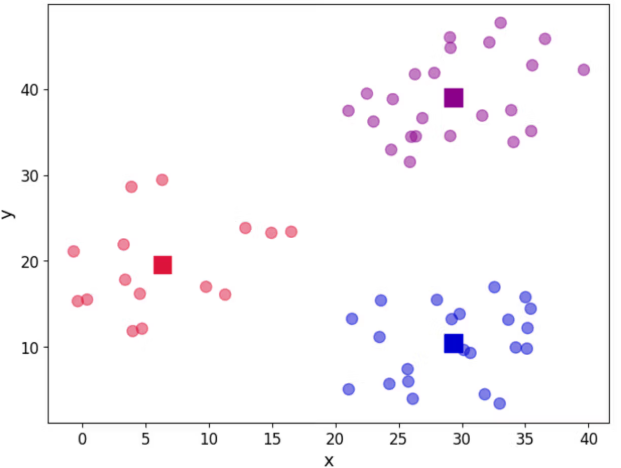
* Initializes k centroids: It randomly selects k central points (called centroids), which serve as the centers of the clusters.
* Assigns each data point to the nearest centroid: Each data point is assigned to the cluster whose centroid is closest (usually based on Euclidean distance).
* Updates centroids: 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.

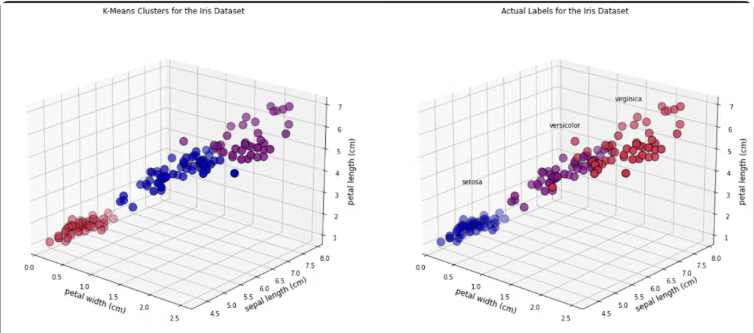
Data Define k and initiate the centroids



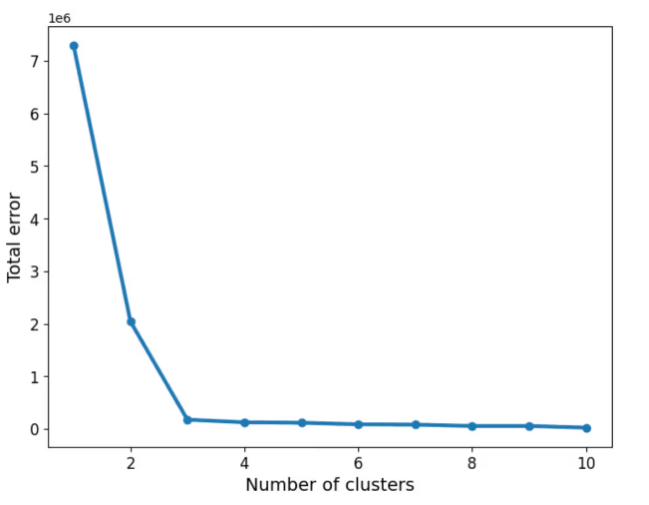
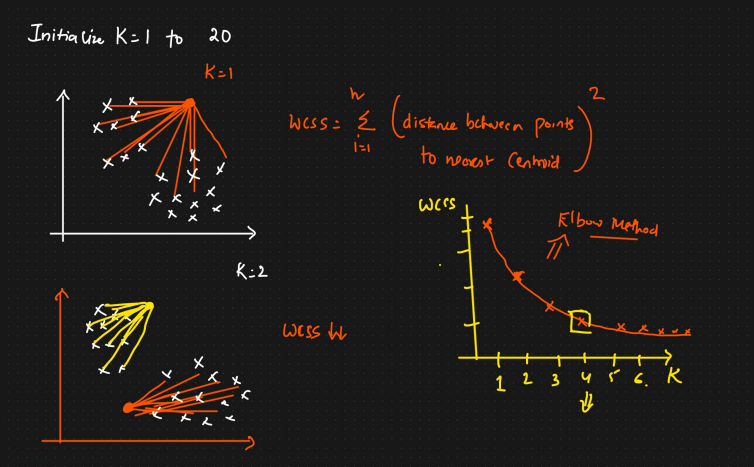
* How to Make cluster : calculate center between two centroid and draw perpendicular line.

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 )

1. **Hierarichal Clustering :**

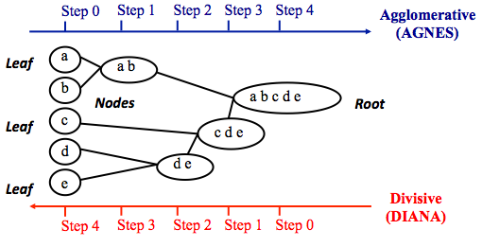
* need if hierarchal clustering
* k-Mean clustering is not work on given type of datsset,that why use this clustering.

****

1. **Agglomerative Hierarchical Clustering (Bottom-Up) :** [click](https://medium.com/analytics-vidhya/hierarchical-clustering-agglomerative-f6906d440981)

Input: Dataset D = {x₁, x₂, ..., xₙ}, 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 (Cᵢ, Cⱼ) with the smallest

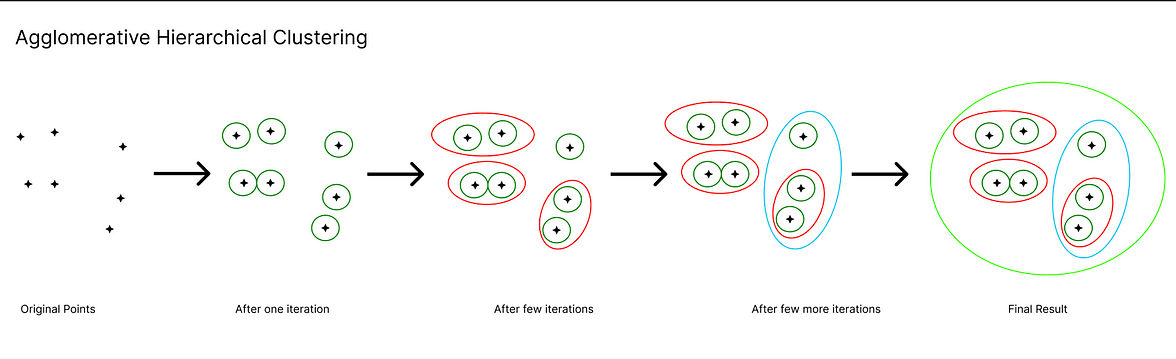
distance.

b. Merge Cᵢ and Cⱼ to form a new cluster Cₖ.

c. Update the distance matrix using the chosen

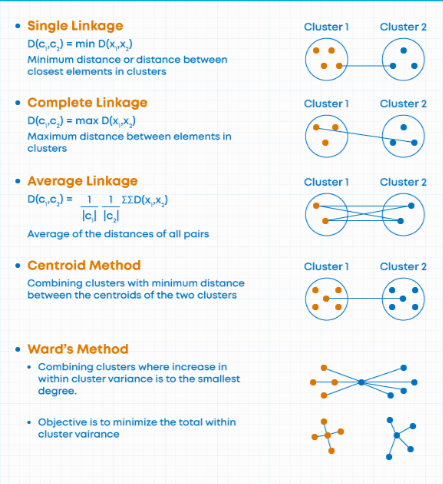
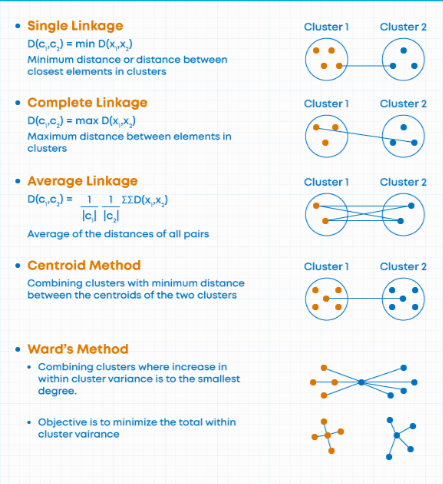
linkage method.

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 |



1. **Divisive Hierarchical Clustering (Top-Down)**

Input: Dataset D = {x₁, x₂, ..., xₙ}

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).

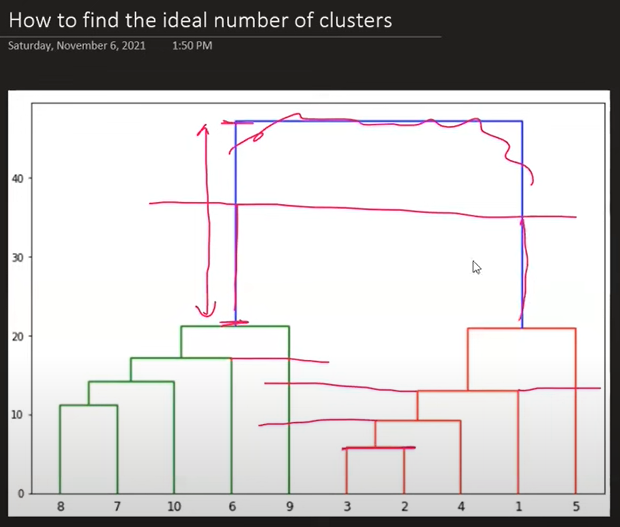
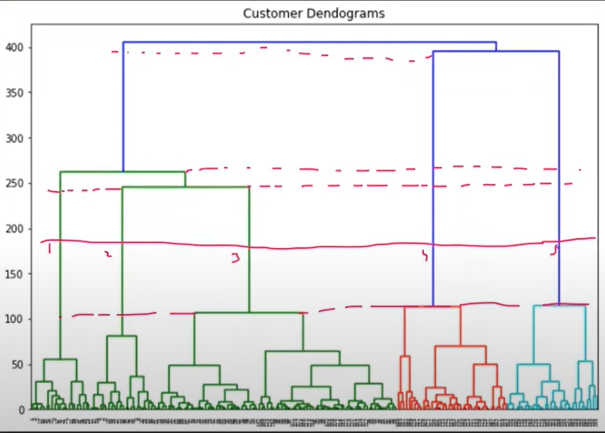
b. Apply a flat clustering algorithm (like K-Means with k=2) to split it.

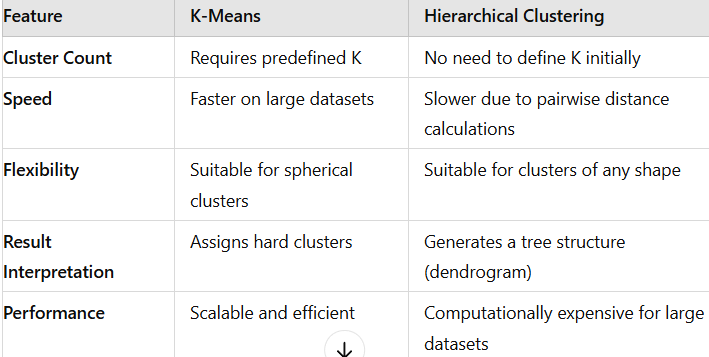
c. Replace the selected cluster with the two resulting clusters.

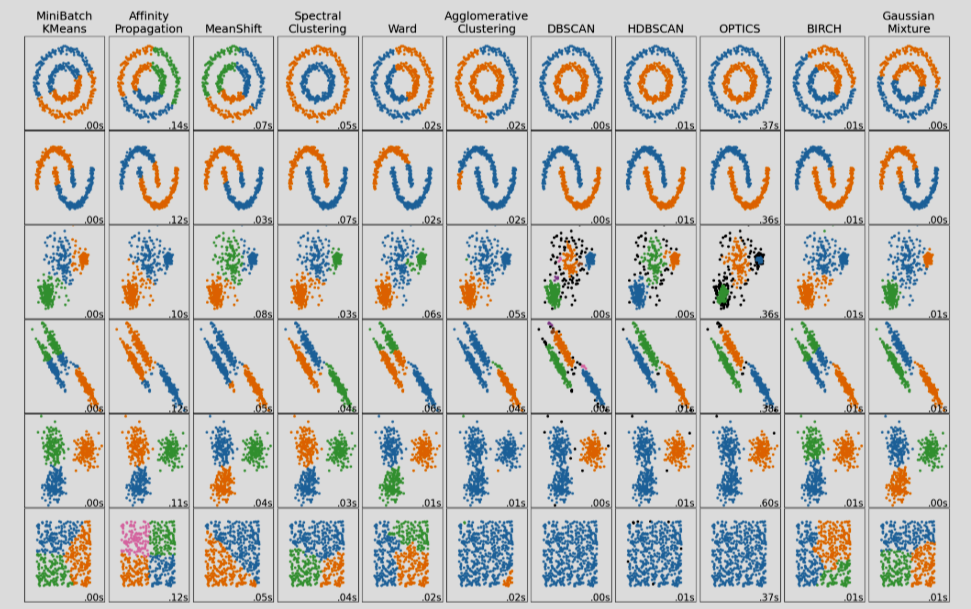
3. Return the hierarchy of clusters.

(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

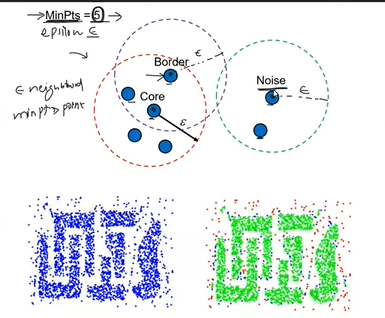






1. **DBSCAN Clustering :** [Viz](https://www.naftaliharris.com/blog/visualizing-dbscan-clustering/)

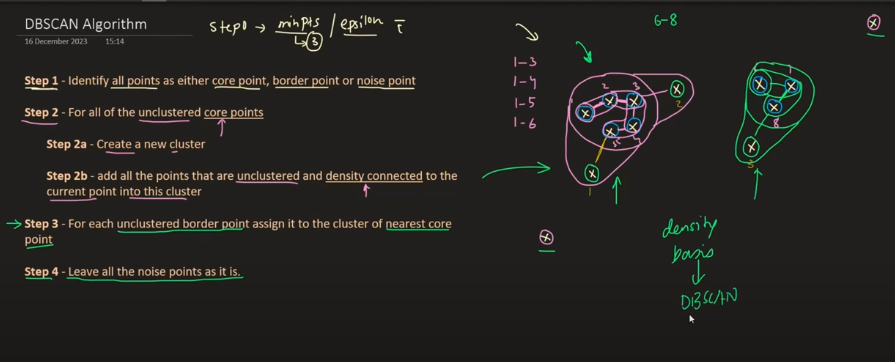
* **DBSCAN** stands for **Density-Based Spatial Clustering of Applications with Noise**.  
  It is a powerful **unsupervised clustering algorithm** especially good at **finding clusters of arbitrary shape** and **detecting outliers**.9



* Types of points :

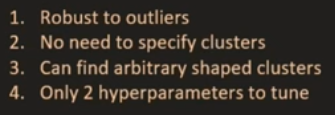
1. Core points
2. Boder points
3. Noise points

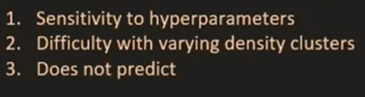
* **Algorithm :**



* **Limitation :**

Advantage

****

****Disadvantag

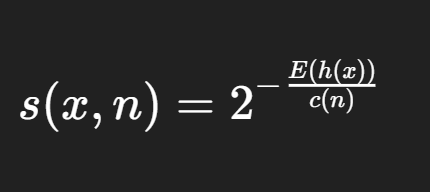
🚀 **Categories of Anomaly Detection Methods :**

1. 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:



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

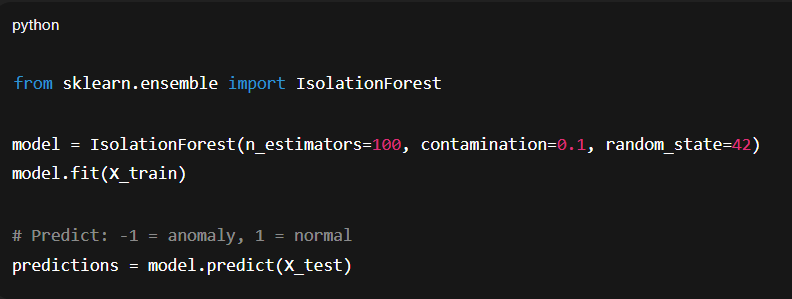
1. **Interpret the score**:
   * **E(h(x)) << h(x)** → **S(x,n)** ≈ Closer to **1** → **outlier**
   * **E(h(x)) >> h(x)** → **S(x,n)** ≈ Closer to **0** → **normal**.

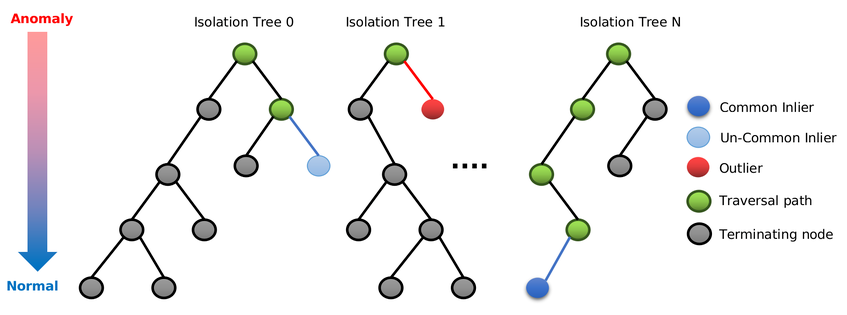
### ✅ ****Advantages:****

* 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







1. 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:**

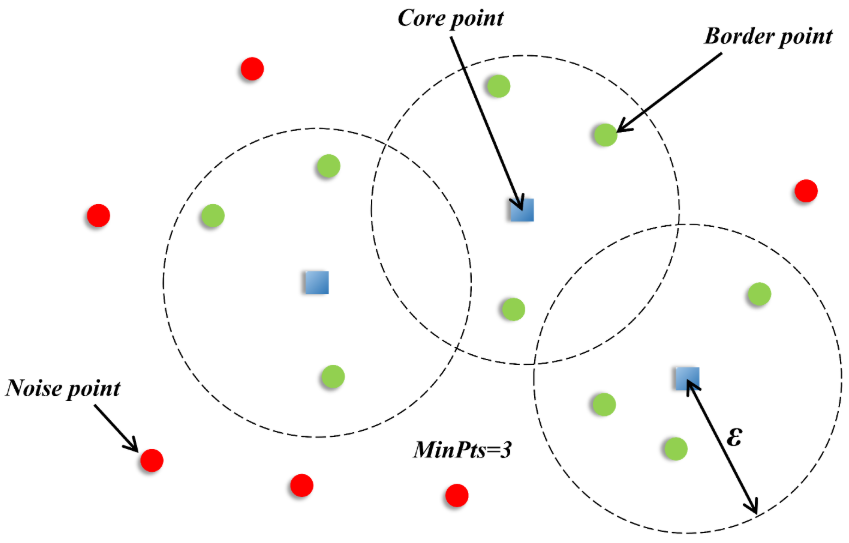
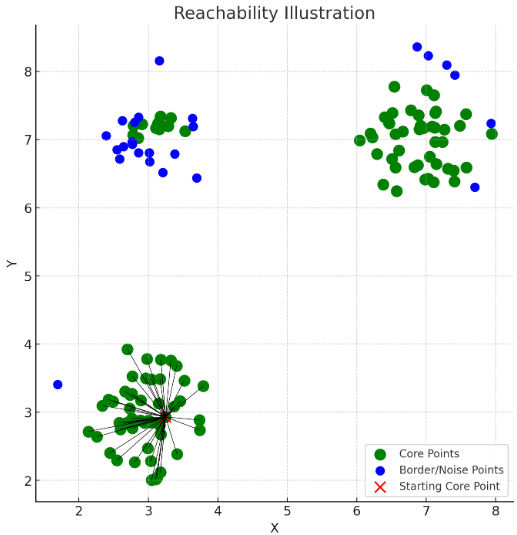
1. Choose a point.
2. If it has at least **min\_samples** within **eps** radius, it's a core point.
3. Expand the cluster from core points by adding all density-reachable points.
4. Points not reachable from any core point are labeled as noise (outliers).

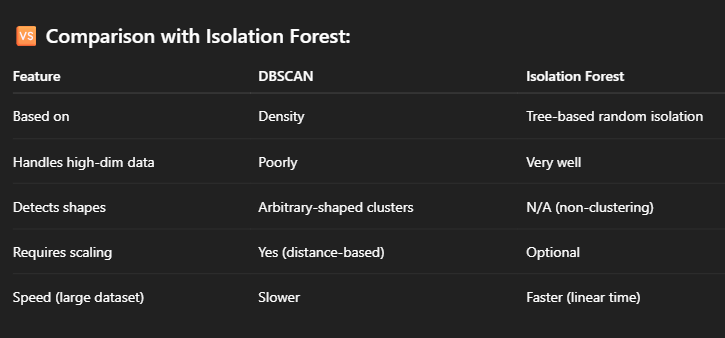
### ✅ ****Advantages:****

1. **No need to specify number of clusters**.
2. Can detect **arbitrary shaped clusters**.
3. Naturally detects **outliers**.

### ⚠️ ****Limitations:****

* Sensitive to choice of eps and min\_samples.
* Does **not scale well** with high-dimensional data.





1. Local Outlier Factor Anamoly Detection :

from sklearn.neighbors import [LocalOutlierFactor](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.LocalOutlierFactor.html#sklearn.neighbors.LocalOutlierFactor)

clf = [LocalOutlierFactor](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.LocalOutlierFactor.html#sklearn.neighbors.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:****

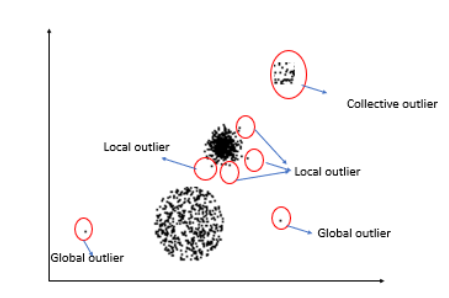
Given:

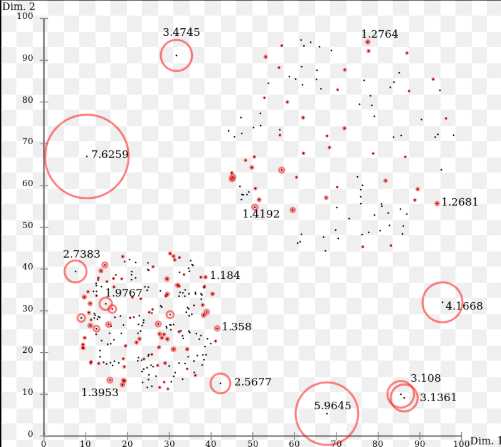
* kkk = number of nearest neighbors (a parameter)

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**.



****

**=========== 🡸** **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}")