Machine Learning

Machine learning is a subset of artificial intelligence that focuses on the development of algorithms and models that allow computers to learn from data and make predictions or decisions without being explicitly programmed

Supervised Learning:

Supervised learning is a machine learning approach in which a model learns from labeled training data to make predictions or decisions about unseen or future data. In supervised learning, the algorithm is provided with a dataset where each example is associated with a corresponding target variable or label.

The goal of supervised learning is to train a model that can generalize patterns and relationships observed in the training data to make accurate predictions or classifications on new, unseen data. The training process involves adjusting the model's internal parameters based on the input features and the corresponding known target values.

Supervised learning can be further categorized into two main types:

Classification: In classification tasks, the target variable is a categorical or discrete variable. The goal is to classify or assign input examples to predefined classes or categories. For example, spam detection, sentiment analysis, or image classification.

Regression: In regression tasks, the target variable is a continuous variable. The goal is to predict a numerical value or estimate a continuous output. For example, predicting house prices, stock market forecasting, or demand prediction.

In [1]:
```
import numpy as np
import pandas as pd
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score

# Generate a regression dataset
X, y = make_regression(n_samples=100, n_features=5, noise=0.1)
```

Data Preprocessing Pipeline

In [2]:
```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

# Define the preprocessing steps
preprocessing_steps = [
    ('imputation', SimpleImputer(strategy='mean')),
```

```
    ('scaling', StandardScaler())
]

# Create the preprocessing pipeline
preprocessing_pipeline = Pipeline(preprocessing_steps)

# Fit and transform the data
X_preprocessed = preprocessing_pipeline.fit_transform(X)
```

Creating Sample Dataset for regression problem

In [3]:
```
# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X_preprocessed, y, test_size=0.2, r

# Print the shape of the dataset
print("Dataset Shape:", X.shape, y.shape)
```

```
Dataset Shape: (100, 5) (100,)
```

In [4]:
```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

# Define the preprocessing steps
preprocessing_steps = [
    ('imputation', SimpleImputer(strategy='mean')),
    ('scaling', StandardScaler())
]

# Create the preprocessing pipeline
preprocessing_pipeline = Pipeline(preprocessing_steps)

# Fit and transform the data
X_preprocessed = preprocessing_pipeline.fit_transform(X)
```

Linear Regression

Ordinary least squares Linear Regression: LinearRegression fits a linear model with coefficients w = (w1, ..., wp) to minimize the residual sum of squares between the observed targets in the dataset, and the targets predicted by the linear approximation.

Linear regression is a simple yet powerful algorithm for regression tasks. However, it doesn't have many hyperparameters to tune compared to other complex algorithms. The main hyperparameters for linear regression are:

No specific hyperparameters:

Linear regression doesn't have many hyperparameters to tune. The model itself learns the coefficients and intercept from the data during the training process.

There are some variations of linear regression, such as Ridge regression and Lasso regression, that introduce regularization to handle multicollinearity and overfitting. These variations have additional hyperparameters to control the regularization strength:

alpha (or lambda):

- It controls the regularization strength. Higher values of alpha increase the amount of regularization, which helps prevent overfitting by shrinking the coefficients towards zero. The optimal value of alpha needs to be determined through cross-validation or other techniques.

solver:

- It specifies the solver algorithm used to optimize the objective function in Ridge or Lasso regression. Popular options include "auto", "svd", "cholesky", "lsqr", and "sparse_cg". The default value is usually suitable for most cases.

normalize:

- It determines whether to normalize the input features before fitting the model. Normalization can be useful when the features have different scales. By default, it is set to False.

random_state:

- It sets the seed for random number generation, ensuring reproducibility of results.

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

In [57]:
```
from sklearn.linear_model import LinearRegression

# Create a Linear Regressor
regression_model = LinearRegression()

# Fit the model on the training data
regression_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = regression_model.predict(X_test)

# Training score
print("Training accuracy : ",regression_model.score(X_train, y_train))

# Evaluate the accuracy of the regressor
accuracy = r2_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

```
Training accuracy :  0.40534430629888885
Accuracy: 0.45011526518117273
```

DecisionTreeRegressor (regression)

Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation.

The DecisionTreeRegressor in scikit-learn is a regression algorithm based on decision trees. It has several hyperparameters that can be tuned to improve its performance. Here are some important hyperparameters for the DecisionTreeRegressor:

criterion:

- It determines the function to measure the quality of a split. The two commonly used options are "mse" (mean squared error) and "mae" (mean absolute error).

max_depth:

- It controls the maximum depth of the decision tree. A larger value can lead to overfitting, while a smaller value can result in underfitting. It is important to tune this hyperparameter carefully based on the complexity of the data.

min_samples_split:

- It sets the minimum number of samples required to split an internal node. Higher values can prevent overfitting by requiring a larger number of samples for a split.

min_samples_leaf:

- It specifies the minimum number of samples required to be at a leaf node. Similar to min_samples_split, higher values can prevent overfitting by ensuring a minimum number of samples in leaf nodes.

max_features:

- It determines the maximum number of features to consider when looking for the best split. Setting it to "auto" uses all features, while "sqrt" uses the square root of the total number of features. Other options include "log2" and a specific integer value.

random_state:

- It sets the seed for random number generation to ensure reproducibility.

https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html

In [6]:
```
from sklearn.tree import DecisionTreeRegressor

# Create a Decision Tree Regressor
regression_model = DecisionTreeRegressor()

# Fit the model on the training data
regression_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = regression_model.predict(X_test)

# Training score
print("Training accuracy : ",regression_model.score(X_train, y_train))
```

```
# Evaluate the accuracy of the regressor
accuracy = r2_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

```
Training accuracy :  1.0
Accuracy: 0.6818562146722973
```

RandomForestRegressor

A random forest is a meta estimator that fits a number of classifying decision trees on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is controlled with the max_samples parameter if bootstrap=True (default), otherwise the whole dataset is used to build each tree.

The RandomForestRegressor in scikit-learn is an ensemble regression algorithm based on random forests. It combines multiple decision trees to make predictions and has several hyperparameters that can be tuned to optimize its performance. Here are some important hyperparameters for the RandomForestRegressor:

n_estimators:

- It specifies the number of decision trees in the random forest. Increasing the number of estimators can improve the model's performance, but it also increases the training time.

criterion:

- It determines the function to measure the quality of a split. The two commonly used options are "mse" (mean squared error) and "mae" (mean absolute error).

max_depth:

- It controls the maximum depth of each decision tree in the random forest. Setting a higher value can result in overfitting, while a smaller value can lead to underfitting.

min_samples_split:

- It sets the minimum number of samples required to split an internal node. Higher values can prevent overfitting by requiring a larger number of samples for a split.

min_samples_leaf:

- It specifies the minimum number of samples required to be at a leaf node. Similar to min_samples_split, higher values can prevent overfitting by ensuring a minimum number of samples in leaf nodes.

max_features:

- It determines the maximum number of features to consider when looking for the best split. Setting it to "auto" uses all features, while "sqrt" uses the square root of the total number of features. Other options include "log2" and a specific integer value.

random_state:

- It sets the seed for random number generation to ensure reproducibility.

https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html

In [7]:
```
from sklearn.ensemble import RandomForestRegressor

# Create a Random Forest Regressor
regression_model = RandomForestRegressor()

# Fit the model on the training data
regression_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = regression_model.predict(X_test)

# Training score
print("Training accuracy : ",regression_model.score(X_train, y_train))

# Evaluate the accuracy of the regressor
accuracy = r2_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

```
Training accuracy :  0.9842654913434831
Accuracy: 0.8729499348949598
```

Support Vector Regressor

The Support Vector Regressor (SVR) in scikit-learn is a regression algorithm based on support vector machines. It uses support vectors to define a hyperplane that minimizes the regression error. Here are some important hyperparameters for the SVR:

kernel:

- It determines the type of kernel function to be used in the SVR. Common choices include "linear", "poly" (polynomial), "rbf" (radial basis function), and "sigmoid". Each kernel has its own set of parameters that can be tuned.

C:

- It controls the regularization parameter. A smaller value of C allows more errors in the training data, while a larger value makes the model focus on minimizing errors. Choosing the appropriate value of C is important to avoid overfitting or underfitting.

epsilon:

- It defines the width of the epsilon-tube, which is the margin of tolerance for errors. Points within this margin are considered as fitting the model. Larger values of epsilon allow more errors within the margin.

gamma:

- It defines the kernel coefficient for "rbf", "poly", and "sigmoid" kernels. It determines the influence of each training example on the decision boundary. A smaller gamma value leads to a smoother decision boundary, while a larger value can result in a more complex boundary.

degree:

- It is the degree of the polynomial kernel function and is only applicable when the kernel is set to "poly".

shrinking:

- It determines whether to use the shrinking heuristic during model training. When set to True, it speeds up the training process by shrinking the number of support vectors.

tol:

- It sets the tolerance for stopping criterion. If the change in the objective function is smaller than the tolerance, the optimization process stops.

  https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html

In [8]:
```
from sklearn.svm import SVR

# Create a Support Vector Regressor
regression_model = SVR()

# Fit the model on the training data
regression_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = regression_model.predict(X_test)

# Training score
print("Training accuracy : ",regression_model.score(X_train, y_train))

# Evaluate the accuracy of the regressor
accuracy = r2_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

```
Training accuracy :  0.07074455027985149
Accuracy: 0.0583273822627709
```

GradientBoostingRegressor

The GradientBoostingRegressor in scikit-learn is an ensemble learning method that combines multiple weak regression models, typically decision trees, to create a stronger predictive model. Here are some important hyperparameters for the GradientBoostingRegressor:

n_estimators:

- It specifies the number of boosting stages (weak models) to be performed. Increasing the number of estimators can improve the performance of the model, but it also increases the training time.

learning_rate:

- It controls the contribution of each weak model to the overall model. A smaller learning rate requires more estimators to achieve the same performance as a higher learning rate. It is important to tune the learning rate in conjunction with the number of estimators.

max_depth:

- It determines the maximum depth of each weak model (decision tree). Increasing the max depth can make the model more complex and prone to overfitting. It is important to find an optimal balance by tuning this parameter.

min_samples_split:

- It sets the minimum number of samples required to split an internal node. Increasing this parameter can prevent overfitting by enforcing more samples in each split.

min_samples_leaf:

- It sets the minimum number of samples required to be at a leaf node. Similar to min_samples_split, increasing this parameter can help prevent overfitting by ensuring a minimum number of samples in each leaf.

max_features:

- It determines the number of features to consider when looking for the best split at each node. You can specify an integer value or a fraction of the total number of features. Setting max_features to a smaller value can reduce the complexity of the model and improve generalization.

subsample:

- It controls the fraction of samples to be used for training each weak model. A value less than 1.0 means that only a fraction of the training data is used, which can help reduce overfitting.

loss:

- It specifies the loss function to be optimized during training. Common options include "ls" (least squares regression), "lad" (least absolute deviation), and "huber" (a combination of squared and absolute loss).

https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html

In [9]:
```
from sklearn.ensemble import GradientBoostingRegressor

# Create a Gradient Boosting Regressor
regression_model = GradientBoostingRegressor()

# Fit the model on the training data
regression_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = regression_model.predict(X_test)

# Training score
print("Training accuracy : ",regression_model.score(X_train, y_train))

# Evaluate the accuracy of the regressor
accuracy = r2_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

```
Training accuracy :  0.9995910329673865
Accuracy: 0.9417841404948906
```

K-Nearest Neighbors (KNN) Regressor

Neighbors-based regression can be used in cases where the data labels are continuous rather than discrete variables. The label assigned to a query point is computed based on the mean of the labels of its nearest neighbors.

The KNeighborsRegressor class in Scikit-learn has several important parameters that control the behavior of the KNN regressor model. Here are the main parameters and their descriptions:

n_neighbors:

- The number of neighbors to consider when making predictions. This parameter affects the model's complexity and ability to capture local patterns. It is typically set to an odd value to avoid ties. Default value is 5.

weights:

- The weight function used in prediction. It can be set to "uniform" (where all neighbors have equal weight) or "distance" (where closer neighbors have higher influence). You can also define a custom weighting function. Default value is "uniform".

algorithm:

- The algorithm used to compute nearest neighbors. It can be set to "auto" (automatically selects the most appropriate algorithm based on the input data), "ball_tree", "kd_tree", or "brute". Default value is "auto".

leaf_size:

- The leaf size passed to the ball_tree or kd_tree algorithms. It affects the speed and memory usage of the algorithm. Default value is 30.

p:

- The power parameter for the Minkowski distance metric. When p=1, it corresponds to the Manhattan distance. When p=2, it corresponds to the Euclidean distance. You can also use other values for custom distance metrics. Default value is 2.

metric:

- The distance metric used to compute the distance between instances. It can be set to a valid string identifier or a callable function that takes two arrays and returns a distance value. Default value is "minkowski" with p=2.

n_jobs:

- The number of parallel jobs to run for neighbor search. It can speed up the computation for large datasets. Setting it to -1 uses all available processors. Default value is 1.

https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html

In [10]:
```
from sklearn.neighbors import KNeighborsRegressor

# Create a KNeighbors Regressor
neigh = KNeighborsRegressor(n_neighbors=2)

# Fit the model on the training data
neigh.fit(X_train, y_train)

# Make predictions on the test set
y_pred = neigh.predict(X_test)

# Training score
print("Training accuracy : ",neigh.score(X_train, y_train))

# Evaluate the accuracy of the regressor
accuracy = r2_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

```
Training accuracy :  0.9520707727390478
Accuracy: 0.6995899801621652
```

Neural Networks (Multi-layer Perceptron) Regressor

The Neural Networks (Multi-layer Perceptron) Regressor, also known as MLP Regressor, is a supervised machine learning algorithm used for regression tasks. It is based on the concept of artificial neural networks, specifically multilayer perceptrons. The MLPRegressor class in Scikit-learn has several important parameters that control the behavior of the MLP Regressor model. Here are the main parameters and their descriptions:

hidden_layer_sizes:

- Tuple or list specifying the number of neurons in each hidden layer. By default, a single hidden layer with 100 neurons is used. You can specify multiple hidden layers by providing a

tuple or list of integers.

activation:

- Activation function to be used in the hidden layers. It can be set to "identity", "logistic", "tanh", or "relu". The default is "relu".

solver:

- Optimization algorithm used to train the model. It can be set to "adam", "lbfgs", or "sgd". The default is "adam".

alpha:

- Regularization parameter that controls the amount of regularization applied to the weights. Higher values result in stronger regularization. The default is 0.0001.

batch_size:

- Number of samples per mini-batch during training. It can be an integer or "auto" (which selects a batch size of min(200, n_samples)). The default is "auto".

learning_rate:

- Learning rate schedule for weight updates. It can be set to "constant", "invscaling", or "adaptive". The default is "constant".

learning_rate_init:

- Initial learning rate used. The default is 0.001.

max_iter:

- Maximum number of iterations (epochs) for training the model. The default is 200.

https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html

In [11]:
```
from sklearn.neural_network import MLPRegressor

# Create a Multi-Layer Regressor
regr = MLPRegressor(random_state=1, max_iter=500)

# Fit the model on the training data
regr.fit(X_train, y_train)

# Make predictions on the test set
y_pred = regr.predict(X_test)

# Training score
print("Training accuracy : ",regr.score(X_test, y_test))

# Evaluate the accuracy of the regressor
accuracy = r2_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

```
Training accuracy :  0.7563595787984869
Accuracy: 0.7563595787984869
```

```
C:\Users\RACHIT\anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_perceptr
on.py:686: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (500) reached a
nd the optimization hasn't converged yet.
  warnings.warn(
```

Creating Sample Dataset for classification problem

In [39]:
```
import pandas as pd
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# Create a synthetic classification dataset
X, y = make_classification(n_samples=100, n_features=10, n_informative=5, n_classes=2, r
```

In [40]:
```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

# Define the preprocessing steps
preprocessing_steps = [
    ('imputation', SimpleImputer(strategy='mean')),
    ('scaling', StandardScaler())
]

# Create the preprocessing pipeline
preprocessing_pipeline = Pipeline(preprocessing_steps)

# Fit and transform the data
X_preprocessed = preprocessing_pipeline.fit_transform(X)
```

In [41]:
```
# Split the dataset into training and test sets
X_train1, X_test1, y_train1, y_test1 = train_test_split(X_preprocessed, y, test_size=0.
```

Logistic Regression

Logistic regression has several hyperparameters that can be tuned to improve the model's performance. Here are some common hyperparameters for logistic regression:

penalty:

- It determines the type of regularization to be applied. It can be set to 'l1' for L1 regularization (Lasso), 'l2' for L2 regularization (Ridge), or 'none' for no regularization.

C:

- It controls the inverse of the regularization strength. Smaller values of C increase the regularization strength, while larger values reduce it. It is typically specified as a positive float.

solver:

- It specifies the algorithm to be used for optimization. Different solvers have different characteristics and may perform better in different scenarios. Common choices include 'liblinear', 'newton-cg', 'lbfgs', 'sag', and 'saga'.

max_iter:

- It sets the maximum number of iterations for the solver to converge. If the solver does not converge within the specified number of iterations, it stops even if the optimization problem is not fully solved. It is an integer value.

class_weight:

- It is used to handle class imbalance in the dataset. By default, all classes have equal weight, but you can set it to 'balanced' to automatically adjust the weights based on the class frequencies.

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

In [44]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
, X_test1, y_train1, y_test1
# Create a Logistic Regression classifier
clf = LogisticRegression()

# Fit the model on the training data
clf.fit(X_train1, y_train1)

# Make predictions on the test set
y_pred1 = clf.predict(X_test1)

# Training score
print("Training accuracy : ",clf.score(X_train1, y_train1))

# Evaluate the accuracy of the classifier
accuracy = accuracy_score(y_test1, y_pred1)
print("Accuracy:", accuracy)
```

```
Training accuracy :  0.8625
Accuracy: 0.9
```

Decision Trees Classifier

Decision trees have several hyperparameters that can be tuned to optimize the model's performance. Here are some commonly used hyperparameters for decision trees:

criterion:

- It specifies the function to measure the quality of a split. Common choices are 'gini' for the Gini impurity and 'entropy' for information gain. The default is 'gini'.

max_depth:

- It controls the maximum depth of the decision tree. Setting a maximum depth can prevent overfitting. If not specified, the tree will expand until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split:

- It specifies the minimum number of samples required to split an internal node. If a node has fewer samples than min_samples_split, it will not be split further.

min_samples_leaf:

- It sets the minimum number of samples required to be at a leaf node. Nodes that would create leaf nodes with fewer samples than min_samples_leaf will be disregarded as potential splits.

max_features:

- It determines the number of features to consider when looking for the best split. It can be an integer, float, or 'auto'. The default value is 'auto', which considers all features.

class_weight:

- It is used to handle class imbalance in the dataset. By default, all classes have equal weight, but you can set it to 'balanced' to automatically adjust the weights based on the class frequencies.

https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

In [16]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Create a Decision Tree classifier
clf = DecisionTreeClassifier()

# Fit the model on the training data
clf.fit(X_train1, y_train1)

# Make predictions on the test set
y_pred1 = clf.predict(X_test1)

# Training score
print("Training accuracy : ",clf.score(X_train1, y_train1))

# Evaluate the accuracy of the classifier
accuracy = accuracy_score(y_test1, y_pred1)
print("Accuracy:", accuracy)
```

```
Training accuracy :  1.0
Accuracy: 0.9
```

Random Forests Classifier

Random Forest is an ensemble learning method that combines multiple decision trees to make predictions. Here are some commonly used hyperparameters for the Random Forest Classifier:

n_estimators:

- It determines the number of decision trees in the random forest. Increasing the number of trees generally improves performance, but it also increases the computational cost. A higher number of estimators reduces the variance but may lead to overfitting.

criterion:

- It specifies the function to measure the quality of a split in each decision tree. Common choices are 'gini' for the Gini impurity and 'entropy' for information gain. The default is 'gini'.

max_depth:

- It controls the maximum depth of each decision tree in the random forest. Setting a maximum depth can prevent overfitting. If not specified, the tree will expand until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split:

- It specifies the minimum number of samples required to split an internal node in each decision tree. If a node has fewer samples than min_samples_split, it will not be split further.

min_samples_leaf:

- It sets the minimum number of samples required to be at a leaf node in each decision tree. Nodes that would create leaf nodes with fewer samples than min_samples_leaf will be disregarded as potential splits.

max_features:

- It determines the number of features to consider when looking for the best split in each decision tree. It can be an integer, float, or 'auto'. The default value is 'auto', which considers the square root of the total number of features.

class_weight:

- It is used to handle class imbalance in the dataset. By default, all classes have equal weight, but you can set it to 'balanced' to automatically adjust the weights based on the class frequencies.

https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

In [17]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score


# Create a Random Forest classifier
```

```
clf = RandomForestClassifier()

# Fit the model on the training data
clf.fit(X_train1, y_train1)

# Make predictions on the test set
y_pred1 = clf.predict(X_test1)

# Training score
print("Training accuracy : ",clf.score(X_train1, y_train1))

# Evaluate the accuracy of the classifier
accuracy = accuracy_score(y_test1, y_pred1)
print("Accuracy:", accuracy)
```

```
Training accuracy :  1.0
Accuracy: 0.955
```

Support Vector Machines (SVM) classifier

Support Vector Machines (SVM) have several hyperparameters that can be tuned to optimize the model's performance. Here are some commonly used hyperparameters for SVM (specifically, the SVC class in scikit-learn):

C:

- It controls the trade-off between maximizing the margin and minimizing the classification error. A smaller C value leads to a larger margin but may allow for more misclassifications, while a larger C value aims to classify all training examples correctly but may result in a narrower margin.

kernel:

- It specifies the kernel function to be used for mapping the input data into higher-dimensional feature space. Common choices include 'linear', 'poly', 'rbf' (Radial Basis Function), and 'sigmoid'. Each kernel has its own set of parameters to be tuned.

gamma:

- It defines the influence of each training example. A smaller gamma value indicates a larger influence and results in a more complex decision boundary, while a larger gamma value reduces the influence and leads to a smoother decision boundary.

degree:

- It is only applicable when the poly kernel is used. It defines the degree of the polynomial function in the kernel.

class_weight:

- It is used to handle class imbalance in the dataset. By default, all classes have equal weight, but you can set it to 'balanced' to automatically adjust the weights based on the class

frequencies.

https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

In [18]:
```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score


# Create an SVM classifier
clf = SVC()

# Fit the model on the training data
clf.fit(X_train1, y_train1)

# Make predictions on the test set
y_pred1 = clf.predict(X_test1)

# Training score
print("Training accuracy : ",clf.score(X_train1, y_train1))

# Evaluate the accuracy of the classifier
accuracy = accuracy_score(y_test1, y_pred1)
print("Accuracy:", accuracy)
```

```
Training accuracy :  0.93
Accuracy: 0.94
```

Gradient Boosting Classifier

Gradient Boosting is a popular ensemble learning method that combines multiple weak learners (typically decision trees) to create a strong predictive model.

Some commonly used hyperparameters for Gradient Boosting:

learning_rate:

- It controls the contribution of each tree to the final prediction. A lower learning rate shrinks the contribution of each tree, leading to a slower learning process but potentially better generalization.

n_estimators:

- It determines the number of weak learners (decision trees) in the gradient boosting model. Increasing the number of estimators can improve performance but also increases computational cost. However, adding more estimators can lead to overfitting if not properly regularized.

max_depth:

- It controls the maximum depth of each decision tree in the gradient boosting model. Setting a maximum depth can prevent overfitting. If not specified, the trees will expand until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split:

- It specifies the minimum number of samples required to split an internal node in each decision tree. If a node has fewer samples than min_samples_split, it will not be split further.

min_samples_leaf:

- It sets the minimum number of samples required to be at a leaf node in each decision tree. Nodes that would create leaf nodes with fewer samples than min_samples_leaf will be disregarded as potential splits.

subsample:

- It controls the fraction of samples used for fitting each tree. A value less than 1.0 results in Stochastic Gradient Boosting, where each tree is trained on a random subset of the training data.

max_features:

- It determines the number of features to consider when looking for the best split in each decision tree. It can be an integer, float, or 'auto'. The default value is 'auto', which considers the square root of the total number of features.

https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html

In [19]:
```
from sklearn.ensemble import GradientBoostingClassifier

# Create a Gradient Boosting Classifier object
clf = GradientBoostingClassifier()

# Fit the model on the training data
clf.fit(X_train1, y_train1)

# Make predictions on the test data
y_pred1 = clf.predict(X_test1)

# Training score
print("Training accuracy : ",clf.score(X_train1, y_train1))

# Evaluate the model performance
accuracy = accuracy_score(y_test1, y_pred1)
print("Accuracy:", accuracy)
```

```
Training accuracy :  0.99625
Accuracy: 0.96
```

K-Nearest Neighbors (KNN) Classifier

The K-Nearest Neighbors (KNN) Classifier in Scikit-learn has several important parameters that control the behavior of the KNN algorithm.

The main parameters and their descriptions:

n_neighbors:

- The number of neighbors to consider when making predictions. This parameter affects the model's complexity and its ability to capture local patterns. It is typically set to an odd value to avoid ties. The default value is 5.

weights:

- The weight function used in prediction. It can be set to "uniform" (where all neighbors have equal weight) or "distance" (where closer neighbors have higher influence). You can also define a custom weighting function. The default value is "uniform".

algorithm:

- The algorithm used to compute nearest neighbors. It can be set to "auto" (which automatically selects the most appropriate algorithm based on the input data), "ball_tree", "kd_tree", or "brute". The default value is "auto".

leaf_size:

- The leaf size passed to the ball_tree or kd_tree algorithms. It affects the speed and memory usage of the algorithm. The default value is 30.

p:

- The power parameter for the Minkowski distance metric. When p=1, it corresponds to the Manhattan distance. When p=2, it corresponds to the Euclidean distance. You can also use other values for custom distance metrics. The default value is 2.

metric:

- The distance metric used to compute the distance between instances. It can be set to a valid string identifier or a callable function that takes two arrays and returns a distance value. The default value is "minkowski" with p=2.

n_jobs:

- The number of parallel jobs to run for neighbor search. It can speed up the computation for large datasets. Setting it to -1 uses all available processors. The default value is 1.

https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html (classification)

In [20]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score


# Create a KNN classifier
clf = KNeighborsClassifier()
```

```
# Fit the model on the training data
clf.fit(X_train1, y_train1)

# Make predictions on the test set
y_pred1 = clf.predict(X_test1)

# Training score
print("Training accuracy : ",clf.score(X_train1, y_train1))

# Evaluate the accuracy of the classifier
accuracy = accuracy_score(y_test1, y_pred1)
print("Accuracy:", accuracy)
```

```
Training accuracy :  0.91375
Accuracy: 0.895
```

Naive Bayes classifier

The Naive Bayes Classifier in Scikit-learn has several important parameters that control the behavior of the Naive Bayes algorithm.

The main parameters and their descriptions:

alpha:

- This is the additive smoothing parameter (also known as Laplace smoothing) that accounts for unseen features in the training data. It helps avoid zero probabilities. The default value is 1.0, which provides a moderate smoothing effect.

fit_prior:

- This parameter specifies whether to learn class prior probabilities from the training data or to use uniform prior probabilities. If set to True, the class priors are learned from the data. If set to False, uniform class priors are assumed. The default value is True.

class_prior:

- This parameter allows you to explicitly set the prior probabilities for each class. It takes a list or array-like object specifying the prior probabilities of the classes. By default, if fit_prior=True, the prior probabilities are computed based on the training data. If fit_prior=False, uniform prior probabilities are assumed.

var_smoothing:

- This is the portion of the largest variance of all features that is added to variances for calculation stability. It helps prevent numerical instability when dealing with features that have zero variance. The default value is 1e-9.

https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html
(classification)

In [21]:
```
from sklearn.naive_bayes import GaussianNB

# Create a GaussianNB classifier
clf = GaussianNB()

# Fit the model on the training data
clf.fit(X_train1, y_train1)

# Make predictions on the test set
y_pred1 = clf.predict(X_test1)

# Training score
print("Training accuracy : ",clf.score(X_train1, y_train1))

# Evaluate the accuracy of the classifier
accuracy = accuracy_score(y_test1, y_pred1)
print("Accuracy:", accuracy)
```

```
Training accuracy :  0.84125
Accuracy: 0.89
```

Neural Networks (Multi-layer Perceptron) Classifier

The Neural Networks (Multi-layer Perceptron) Classifier, also known as MLP Classifier, is a supervised machine learning algorithm that is based on artificial neural networks, specifically multilayer perceptrons. It is widely used for classification tasks.

The main parameters and their descriptions:

hidden_layer_sizes:

- Tuple or list specifying the number of neurons in each hidden layer. By default, a single hidden layer with 100 neurons is used. You can specify multiple hidden layers by providing a tuple or list of integers.

activation:

- Activation function to be used in the hidden layers. It can be set to "identity", "logistic", "tanh", or "relu". The default is "relu".

solver:

- Optimization algorithm used to train the model. It can be set to "adam", "lbfgs", or "sgd". The default is "adam".

alpha:

- Regularization parameter that controls the amount of regularization applied to the weights. Higher values result in stronger regularization. The default is 0.0001.

batch_size:

- Number of samples per mini-batch during training. It can be an integer or "auto" (which selects a batch size of min(200, n_samples)). The default is "auto".

learning_rate:

- Learning rate schedule for weight updates. It can be set to "constant", "invscaling", or "adaptive". The default is "constant".

learning_rate_init:

- Initial learning rate used. The default is 0.001.

max_iter:

- Maximum number of iterations (epochs) for training the model. The default is 200.

https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html (classification)

In [22]:
```
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Create an MLP classifier
clf = MLPClassifier()

# Fit the model on the training data
clf.fit(X_train1, y_train1)

# Make predictions on the test set
y_pred1 = clf.predict(X_test1)

# Training score
print("Training accuracy : ",clf.score(X_train1, y_train1))

# Evaluate the accuracy of the classifier
accuracy = accuracy_score(y_test1, y_pred1)
print("Accuracy:", accuracy)
```

```
Training accuracy :  0.955
Accuracy: 0.95
C:\Users\RACHIT\anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:686: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and the optimization hasn't converged yet.
  warnings.warn(
```

Unsupervised Learning:

Unsupervised learning involves learning patterns and structures in unlabeled data. The algorithm explores the data and finds meaningful relationships or clusters without any predefined labels or targets.

- Popular unsupervised learning algorithms

K-Means Clustering:

K-Means Clustering is an unsupervised machine learning algorithm used for clustering and data segmentation. It aims to partition a given dataset into K clusters, where each data point belongs to the cluster with the nearest mean (centroid). The algorithm iteratively assigns data points to the nearest centroid and updates the centroids until convergence.

The key hyperparameters of the K-Means Clustering algorithm:

n_clusters:

- The number of clusters (K) to form. It determines the desired number of clusters in the dataset. This parameter must be specified before running the algorithm.

init:

- The method for initializing the initial centroids. It can be set to "k-means++" (smart initialization that improves convergence) or "random" (randomly selects initial centroids). The default is "k-means++".

n_init:

- The number of times the algorithm will be run with different centroid seeds. The final result will be the best output in terms of inertia. The default is 10.

max_iter:

- The maximum number of iterations for the algorithm to converge. If convergence is not reached within this limit, the algorithm stops. The default is 300.

tol:

- The tolerance for convergence. If the difference in the centroids' positions between iterations is less than this value, the algorithm is considered to have converged. The default is 1e-4.

algorithm:

- The algorithm used to compute the K-Means Clustering. It can be set to "auto", "full", or "elkan". The "auto" option automatically selects the best algorithm based on the dataset. The default is "auto".

https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html

In [23]:
```
from sklearn.datasets import make_blobs

# Create a synthetic dataset with 100 samples and 2 features
X, _ = make_blobs(n_samples=100, n_features=2, centers=3, random_state=42)
```

In [24]:
```
from sklearn.cluster import KMeans
```

```
# Create a K-Means Clustering object with desired number of clusters (K)
kmeans = KMeans(n_clusters=3)

# Fit the K-Means model on the data
kmeans.fit(X)

# Get the cluster labels for each data point
labels = kmeans.labels_

# Get the cluster centroids
centroids = kmeans.cluster_centers_

print("Cluster Labels:", labels)
print("Cluster Centroids:", centroids)
```

```
C:\Users\RACHIT\anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py:870: FutureWarni
ng: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value o
f `n_init` explicitly to suppress the warning
  warnings.warn(
C:\Users\RACHIT\anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py:1382: UserWarnin
g: KMeans is known to have a memory leak on Windows with MKL, when there are less chunk
s than available threads. You can avoid it by setting the environment variable OMP_NUM_
THREADS=1.
  warnings.warn(
Cluster Labels: [1 2 0 2 1 2 0 2 2 0 0 1 1 0 0 1 1 0 1 1 0 1 1 0 0 0 2 1 1 1 1 2 2 1 0
0 0
 0 2 2 1 0 2 0 0 2 1 1 1 2 2 2 0 1 1 1 0 0 2 0 1 2 1 2 1 1 2 1 2 2 2 1 1 0
 2 1 2 1 2 2 0 2 0 1 0 0 0 2 0 2 2 2 0 2 0 0 0 2 1 0]
Cluster Centroids: [[-2.66780392  8.93576069]
 [-6.95170962 -6.67621669]
 [ 4.49951001  1.93892013]]
```

Agglomerative Clustering (Hierarchical Clustering)

Hierarchical Clustering is an unsupervised machine learning algorithm used for clustering and data segmentation. It builds a hierarchy of clusters by iteratively merging or splitting clusters based on a distance or similarity metric. There are two main types of hierarchical clustering: Agglomerative and Divisive.

Here are the key hyperparameters for hierarchical clustering:

n_clusters:

- The desired number of clusters to form. This parameter is optional and specifies the number of clusters to be formed at the end of the clustering process. If not provided, the algorithm will create clusters based on a distance threshold or another stopping criterion.

affinity:

- The distance or similarity metric used to compute the proximity between data points. It can be set to "euclidean", "manhattan", "cosine", or other valid distance measures. The default is "euclidean".

linkage:

- The linkage criterion used to determine how to merge or split clusters. It can be set to "ward", "complete", "average", or "single". The default is "ward", which minimizes the variance of the clusters being merged.

distance_threshold:

- The threshold to use when forming flat clusters. This parameter is optional and allows you to specify a distance value at which clusters are formed. Clusters that have a distance less than or equal to this threshold are merged into a single cluster.

https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html

In [25]:
```
from sklearn.cluster import AgglomerativeClustering

# Create an Agglomerative Clustering object with desired number of clusters (n_clusters)
hc = AgglomerativeClustering(n_clusters=3, linkage='ward')

# Fit the hierarchical clustering model on the data
hc.fit(X)

# Get the cluster labels for each data point
labels = hc.labels_

print("Cluster Labels:", labels)
```

```
Cluster Labels: [1 2 0 2 1 2 0 2 2 0 0 1 1 0 0 1 1 0 1 1 0 1 1 0 0 0 2 1 1 1 1 2 2 1 0
0 0
 0 2 2 1 0 2 0 0 2 1 1 1 2 2 2 0 1 1 1 0 0 2 0 1 2 1 2 1 1 2 1 2 2 2 1 1 0
 2 1 2 1 2 2 0 2 0 1 0 0 0 2 0 2 2 2 0 2 0 0 0 2 1 0]
```

DBSCAN (Density-Based Spatial Clustering of Applications with Noise):

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is an unsupervised machine learning algorithm used for clustering and data segmentation. It groups together data points that are close to each other in dense regions, while labeling data points in sparser regions as noise or outliers. DBSCAN does not require the number of clusters to be specified in advance and can discover clusters of arbitrary shapes.

The key hyperparameters for DBSCAN:

eps:

- The maximum distance between two data points to be considered neighbors. It defines the radius of the neighborhood around each data point. Points within this distance are considered part of the same cluster. The default value is 0.5.

min_samples:

- The minimum number of data points required to form a dense region. Points that have at least min_samples neighbors within the eps radius are considered core points and are used to form clusters. The default value is 5.

metric:

- The distance metric used to compute the distance between data points. It can be set to a valid string identifier or a callable function that takes two arrays and returns a distance value. The default is "euclidean".

algorithm:

- The algorithm used to compute the DBSCAN clustering. It can be set to "auto", "ball_tree", "kd_tree", or "brute". The "auto" option automatically selects the best algorithm based on the dataset. The default is "auto".

leaf_size:

- The leaf size passed to the ball_tree or kd_tree algorithms. It affects the speed and memory usage of the algorithm. The default is 30.

n_jobs:

- The number of parallel jobs to run for neighbor search. It can speed up the computation for large datasets. Setting it to -1 uses all available processors. The default is 1.

https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html

In [26]:
```
from sklearn.cluster import DBSCAN

# Create a DBSCAN object with desired epsilon (eps) and minimum samples (min_samples)
dbscan = DBSCAN(eps=0.5, min_samples=5)

# Fit the DBSCAN model on the data
dbscan.fit(X)

# Get the cluster labels for each data point
labels = dbscan.labels_

print("Cluster Labels:", labels)
```

```
Cluster Labels: [-1 -1 -1 -1  0  1 -1  4 -1 -1  3  0  0 -1 -1  2  0 -1  2  0  3  0 -1 -
1
 -1 -1  1 -1  2 -1  2  4  4 -1 -1  3 -1  3 -1 -1 -1  3 -1 -1  3 -1 -1  0
 -1  1 -1  5 -1  0 -1  0 -1 -1 -1 -1 -1  1 -1 -1 -1 -1 -1 -1  4  5 -1  0
  2 -1  5  0 -1 -1 -1 -1 -1  1 -1 -1 -1 -1 -1 -1 -1 -1  5 -1 -1 -1 -1 -1
 -1  4 -1  3]
```

Dimensionality reduction

Principal Component Analysis (PCA)

In [27]:
```
from sklearn.decomposition import PCA

# Create a PCA object with desired number of components
```

```
pca = PCA(n_components=2)

# Fit the PCA model on the data
pca.fit(X)

# Transform the data to the lower-dimensional space
X_pca = pca.transform(X)

# print(X_pca)
```

t-SNE (t-Distributed Stochastic Neighbor Embedding)

In [28]:
```
from sklearn.manifold import TSNE

# Create a t-SNE object with desired number of components
tsne = TSNE(n_components=2)

# Fit the t-SNE model on the data
X_tsne = tsne.fit_transform(X)

# print(X_tsne)
```

Hyperparameter tunning for regression problem

In [29]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.svm import SVR
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_squared_error, r2_score


# Linear Regression hyperparameters
linear_params = {}

# Decision Tree Regression hyperparameters
dt_params = {
    'max_depth': [None, 5, 10],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Random Forest Regression hyperparameters
rf_params = {
    'n_estimators': [100, 200, 300],
    'max_depth': [None, 5, 10],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['auto', 'sqrt', 'log2']
}

# Gradient Boosting Regression hyperparameters
gb_params = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.01, 0.1, 1.0],
```

```
    'max_depth': [3, 5, 7],
    'subsample': [0.5, 0.8, 1.0]
}

# Support Vector Regression hyperparameters
svr_params = {
    'C': [0.1, 1.0, 10.0],
    'kernel': ['linear', 'rbf'],
    'epsilon': [0.1, 0.2, 0.3]
}

# Neural Network Regression hyperparameters
nn_params = {
    'hidden_layer_sizes': [(100,), (100, 50), (50, 50)],
    'activation': ['relu', 'tanh'],
    'learning_rate': ['constant', 'adaptive'],
    'alpha': [0.0001, 0.001, 0.01]
}

# Create the regression models
linear_regressor = LinearRegression()
dt_regressor = DecisionTreeRegressor()
rf_regressor = RandomForestRegressor()
gb_regressor = GradientBoostingRegressor()
svr_regressor = SVR()
nn_regressor = MLPRegressor()

# Perform grid search cross-validation for each algorithm
linear_grid_search = GridSearchCV(linear_regressor, linear_params, cv=5)
dt_grid_search = GridSearchCV(dt_regressor, dt_params, cv=5)
rf_grid_search = GridSearchCV(rf_regressor, rf_params, cv=5)
gb_grid_search = GridSearchCV(gb_regressor, gb_params, cv=5)
svr_grid_search = GridSearchCV(svr_regressor, svr_params, cv=5)
nn_grid_search = GridSearchCV(nn_regressor, nn_params, cv=5)

# Fit the models with training data
linear_grid_search.fit(X_train, y_train)
dt_grid_search.fit(X_train, y_train)
rf_grid_search.fit(X_train, y_train)
gb_grid_search.fit(X_train, y_train)
svr_grid_search.fit(X_train, y_train)
nn_grid_search.fit(X_train, y_train)

# Print the best hyperparameters for each algorithm
print("Linear Regression Best Hyperparameters:", linear_grid_search.best_params_)
print("Decision Tree Regression Best Hyperparameters:", dt_grid_search.best_params_)
print("Random Forest Regression Best Hyperparameters:", rf_grid_search.best_params_)
print("Gradient Boosting Regression Best Hyperparameters:", gb_grid_search.best_params_
print("Support Vector Regression Best Hyperparameters:", svr_grid_search.best_params_)
print("Neural Network Regression Best Hyperparameters:", nn_grid_search.best_params_)

print("Linear Regression Best Score:", linear_grid_search.best_score_*100)
print("Decision Tree Regression Best Score:", dt_grid_search.best_score_*100)
print("Random Forest Regression Best Score:", rf_grid_search.best_score_*100)
print("Gradient Boosting Regression Best Score:", gb_grid_search.best_score_*100)
print("Support Vector Regression Best Score:", svr_grid_search.best_score_*100)
print("Neural Network Regression Best Score:", nn_grid_search.best_score_*100)

pred_linear_grid_search = linear_grid_search.predict(X_test)
pred_dt_grid_search = dt_grid_search.predict(X_test)
```

```
pred_rf_grid_search = rf_grid_search.predict(X_test)
pred_gb_grid_search = gb_grid_search.predict(X_test)
pred_svr_grid_search = svr_grid_search.predict(X_test)
pred_nn_grid_search = nn_grid_search.predict(X_test)

linear_grid_search_mse =  mean_squared_error(y_test,pred_linear_grid_search )
dt_grid_search_mse = mean_squared_error(y_test,pred_dt_grid_search)
rf_grid_search_mse = mean_squared_error(y_test,pred_rf_grid_search)
gb_grid_search_mse = mean_squared_error(y_test, pred_gb_grid_search)
svr_grid_search_mse = mean_squared_error(y_test,pred_svr_grid_search)
nn_grid_search_mse = mean_squared_error(y_test,pred_nn_grid_search)

linear_grid_search_r2 = r2_score(y_test,pred_linear_grid_search)
dt_grid_search_r2 = r2_score(y_test, pred_dt_grid_search)
rf_grid_search_r2 = r2_score(y_test, pred_rf_grid_search)
gb_grid_search_r2 = r2_score(y_test, pred_gb_grid_search)
svr_grid_search_r2 = r2_score(y_test,pred_svr_grid_search)
nn_grid_search_r2 = r2_score(y_test, pred_nn_grid_search)

print("Linear Regression mse Score:", linear_grid_search_mse*100)
print("Decision Tree Regression mse Score:", dt_grid_search_mse*100)
print("Random Forest Regression mse Score:", rf_grid_search_mse*100)
print("Gradient Boosting Regression mse Score:", gb_grid_search_mse*100)
print("Support Vector Regression mse Score:", svr_grid_search_mse*100)
print("Neural Network Regression mse Score:", nn_grid_search_mse*100)

print("Linear Regression R2 Score:", linear_grid_search_r2*100)
print("Decision Tree Regression R2 Score:", dt_grid_search_r2*100)
print("Random Forest Regression R2 Score:", rf_grid_search_r2*100)
print("Gradient Boosting Regression R2 Score:", gb_grid_search_r2*100)
print("Support Vector Regression R2 Score:", svr_grid_search_r2*100)
print("Neural Network Regression R2 Score:", nn_grid_search_r2*100)
```

```
Linear Regression Best Hyperparameters: {}
Decision Tree Regression Best Hyperparameters: {'max_depth': 10, 'min_samples_leaf': 1,
'min_samples_split': 2}
Random Forest Regression Best Hyperparameters: {'max_depth': 5, 'max_features': 'auto',
'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}
Gradient Boosting Regression Best Hyperparameters: {'learning_rate': 0.1, 'max_depth':
3, 'n_estimators': 300, 'subsample': 0.5}
Support Vector Regression Best Hyperparameters: {'C': 10.0, 'epsilon': 0.1, 'kernel':
'linear'}
Neural Network Regression Best Hyperparameters: {'activation': 'relu', 'alpha': 0.01,
'hidden_layer_sizes': (100, 50), 'learning_rate': 'adaptive'}
Linear Regression Best Score: 99.99984846009961
Decision Tree Regression Best Score: 72.1210073196629
Random Forest Regression Best Score: 81.74597381927956
Gradient Boosting Regression Best Score: 93.57410895015555
Support Vector Regression Best Score: 99.9998444342866
Neural Network Regression Best Score: 95.22558537443238
Linear Regression mse Score: 0.8438040237743157
Decision Tree Regression mse Score: 201707.21642889347
Random Forest Regression mse Score: 74243.59332381857
Gradient Boosting Regression mse Score: 19998.896499366238
Support Vector Regression mse Score: 0.9537474885350984
Neural Network Regression mse Score: 31051.636233039717
Linear Regression R2 Score: 99.99985594793908
Decision Tree Regression R2 Score: 65.56506082934398
Random Forest Regression R2 Score: 87.32532397611135
Gradient Boosting Regression R2 Score: 96.585839631183
Support Vector Regression R2 Score: 99.99983717867246
Neural Network Regression R2 Score: 94.69894422339124
```

Hyperparameter tunning for classification problem

In [42]:

```
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import AdaBoostClassifier,GradientBoostingClassifier
from sklearn.metrics import classification_report

# Define the parameter grid for each algorithm
param_grid_logistic_regression = {'C': [0.1, 1, 10], 'penalty': ['l2']}
param_grid_decision_tree = {'max_depth': [3, 5, 7], 'min_samples_split': [2, 5, 10], 'm
param_grid_svm = {'C': [0.1, 1, 10], 'kernel': ['linear', 'rbf'], 'gamma': [0.1, 0.01, (
param_grid_random_forest = {'n_estimators': [100, 200, 300], 'max_depth': [5, 10, 15],
param_grid_gradient_boosting = {'learning_rate': [0.1, 0.01, 0.001], 'n_estimators': [10
param_grid_adaboost = {'learning_rate': [0.1, 0.01, 0.001], 'n_estimators': [100, 200, 1

# Perform Grid Search for each algorithm
grid_search_logistic_regression = GridSearchCV(LogisticRegression(), param_grid_logistic
grid_search_decision_tree = GridSearchCV(DecisionTreeClassifier(), param_grid_decision_t
grid_search_svm = GridSearchCV(SVC(), param_grid_svm, cv=5)
grid_search_random_forest = GridSearchCV(RandomForestClassifier(), param_grid_random_for
grid_search_gradient_boosting = GridSearchCV(GradientBoostingClassifier(), param_grid_gr
grid_search_adaboost = GridSearchCV(AdaBoostClassifier(), param_grid_adaboost, cv=5)

# Fit the models and find the best hyperparameters
grid_search_logistic_regression.fit(X_train1, y_train1)
grid_search_decision_tree.fit(X_train1, y_train1)
grid_search_svm.fit(X_train1, y_train1)
grid_search_random_forest.fit(X_train1, y_train1)
grid_search_gradient_boosting.fit(X_train1, y_train1)
```

```
grid_search_adaboost.fit(X_train1, y_train1)

# Get the best hyperparameters and their respective scores
best_params_logistic_regression = grid_search_logistic_regression.best_params_
best_params_decision_tree = grid_search_decision_tree.best_params_
best_params_svm = grid_search_svm.best_params_
best_params_random_forest = grid_search_random_forest.best_params_
best_params_gradient_boosting = grid_search_gradient_boosting.best_params_
best_params_adaboost = grid_search_adaboost.best_params_

print("Best hyperparameters for Logistic Regression:", best_params_logistic_regression)
print("Best hyperparameters for Decision Tree:", best_params_decision_tree)
print("Best hyperparameters for SVM:", best_params_svm)
print("Best hyperparameters for Random Forest:", best_params_random_forest)
print("Best hyperparameters for Gradient Boosting:", best_params_gradient_boosting)
print("Best hyperparameters for AdaBoost:", best_params_adaboost)

print("Best best score for Logistic Regression:", grid_search_logistic_regression.best_s
print("Best best score for Decision Tree:", grid_search_decision_tree.best_score_*100)
print("Best best score for SVM:", grid_search_svm.best_score_*100)
print("Best best score for Random Forest:", grid_search_random_forest.best_score_*100)
print("Best best score for Gradient Boosting:", grid_search_gradient_boosting.best_scor
print("Best best score for AdaBoost:", grid_search_adaboost.best_score_*100)

pred_grid_search_logistic_regression = grid_search_logistic_regression.predict(X_test1)
pred_grid_search_decision_tree = grid_search_decision_tree.predict(X_test1)
pred_grid_search_svm = grid_search_svm.predict(X_test1)
pred_grid_search_random_forest = grid_search_random_forest.predict(X_test1)
pred_grid_search_gradient_boosting = grid_search_gradient_boosting.predict(X_test1)
pred_grid_search_adaboost = grid_search_adaboost.predict(X_test1)

print(classification_report(y_test1,pred_grid_search_logistic_regression))
print(classification_report(y_test1,pred_grid_search_decision_tree))
print(classification_report(y_test1,pred_grid_search_svm))
print(classification_report(y_test1,pred_grid_search_random_forest))
print(classification_report(y_test1,pred_grid_search_gradient_boosting))
print(classification_report(y_test1,pred_grid_search_adaboost))
```

```
Best hyperparameters for Logistic Regression: {'C': 0.1, 'penalty': 'l2'}
Best hyperparameters for Decision Tree: {'max_depth': 5, 'min_samples_leaf': 1, 'min_sa
mples_split': 10}
Best hyperparameters for SVM: {'C': 1, 'gamma': 0.1, 'kernel': 'rbf'}
Best hyperparameters for Random Forest: {'max_depth': 5, 'min_samples_split': 2, 'n_est
imators': 300}
Best hyperparameters for Gradient Boosting: {'learning_rate': 0.001, 'max_depth': 3, 'n
_estimators': 100}
Best hyperparameters for AdaBoost: {'learning_rate': 0.1, 'n_estimators': 100}
Best best score for Logistic Regression: 85.0
Best best score for Decision Tree: 86.25
Best best score for SVM: 88.75
Best best score for Random Forest: 91.25
Best best score for Gradient Boosting: 88.75
Best best score for AdaBoost: 85.0
              precision    recall  f1-score   support

           0       0.88      0.78      0.82         9
           1       0.83      0.91      0.87        11

    accuracy                           0.85        20
   macro avg       0.85      0.84      0.85        20
weighted avg       0.85      0.85      0.85        20

              precision    recall  f1-score   support

           0       0.86      0.67      0.75         9
           1       0.77      0.91      0.83        11

    accuracy                           0.80        20
   macro avg       0.81      0.79      0.79        20
weighted avg       0.81      0.80      0.80        20

              precision    recall  f1-score   support

           0       0.86      0.67      0.75         9
           1       0.77      0.91      0.83        11

    accuracy                           0.80        20
   macro avg       0.81      0.79      0.79        20
weighted avg       0.81      0.80      0.80        20

              precision    recall  f1-score   support

           0       0.86      0.67      0.75         9
           1       0.77      0.91      0.83        11

    accuracy                           0.80        20
   macro avg       0.81      0.79      0.79        20
weighted avg       0.81      0.80      0.80        20

              precision    recall  f1-score   support

           0       0.86      0.67      0.75         9
           1       0.77      0.91      0.83        11

    accuracy                           0.80        20
   macro avg       0.81      0.79      0.79        20
weighted avg       0.81      0.80      0.80        20
```

```
              precision    recall  f1-score   support

           0       0.67      0.67      0.67         9
           1       0.73      0.73      0.73        11

    accuracy                           0.70        20
   macro avg       0.70      0.70      0.70        20
weighted avg       0.70      0.70      0.70        20
```

Metrics Evaluation for Classification

Accuracy

The accuracy_score function works by comparing the predicted labels with the true labels and computing the proportion of correct predictions. Here's an overview of how it works:

Input:

- The function takes two arrays as input - y_true and y_pred.
  - y_true represents the true labels or target values.
  - y_pred represents the predicted labels or target values.

Comparison:

- The function compares each element of y_true with the corresponding element of y_pred. It checks if the predicted label matches the true label for each sample.

Counting:

- It counts the number of correct predictions, i.e., the number of times the predicted label matches the true label.

Accuracy Calculation:

- The function calculates the accuracy by dividing the count of correct predictions by the total number of samples.

accuracy = (number of correct predictions) / (total number of samples)

Output:

- The function returns the accuracy score, which is a value between 0 and 1. A higher accuracy score indicates a better performance of the classification model.

In [45]:
```
from sklearn.metrics import accuracy_score

accuracy = accuracy_score(y_test1, y_pred1)
```

Precision_score

The precision_score function works by comparing the predicted labels with the true labels and computing the precision of the classification model's predictions. Here's an overview of how it works:

Input:

- The function takes two arrays as input - y_true and y_pred.
  - y_true represents the true labels or target values.
  - y_pred represents the predicted labels or target values.

True Positive (TP) Calculation:

- The function counts the number of true positive predictions, which are the cases where the predicted label matches the true label and both are positive.

False Positive (FP) Calculation:

- The function counts the number of false positive predictions, which are the cases where the predicted label is positive, but the true label is negative.

Precision Calculation:

- The function calculates the precision by dividing the number of true positive predictions by the sum of true positive and false positive predictions.

precision = TP / (TP + FP)

Output:

- The function returns the precision score, which is a value between 0 and 1. A higher precision score indicates a lower rate of false positive predictions and better performance of the classification model in correctly predicting positive samples.

In [46]:
```
# Precision:
from sklearn.metrics import precision_score

precision = precision_score(y_test1, y_pred1)
```

Recall_score

The recall_score function works by comparing the predicted labels with the true labels and computing the recall of the classification model's predictions. Here's an overview of how it works:

Input:

- The function takes two arrays as input - y_true and y_pred.
    - y_true represents the true labels or target values.
    - y_pred represents the predicted labels or target values.

True Positive (TP) Calculation:

- The function counts the number of true positive predictions, which are the cases where the predicted label matches the true label and both are positive.

False Negative (FN) Calculation:

- The function counts the number of false negative predictions, which are the cases where the predicted label is negative, but the true label is positive.

Recall Calculation:

- The function calculates the recall by dividing the number of true positive predictions by the sum of true positive and false negative predictions.

recall = TP / (TP + FN)

Output:

- The function returns the recall score, which is a value between 0 and 1. A higher recall score indicates a lower rate of false negative predictions and better performance of the classification model in correctly identifying positive samples.

In [47]:
```
# Recall (Sensitivity or True Positive Rate):
from sklearn.metrics import recall_score

recall = recall_score(y_test1, y_pred1)
```

F1 score

The F1 score is a measure of a classification model's accuracy that considers both precision and recall. It provides a single metric that balances the trade-off between precision and recall, making it a useful evaluation metric for imbalanced datasets or situations where both precision and recall are important.

Here's how the F1 score works:

Precision Calculation:

- Precision is the ratio of true positive predictions to the sum of true positive and false positive predictions. It represents the accuracy of positive predictions.

precision = TP / (TP + FP)

Recall Calculation:

- Recall is the ratio of true positive predictions to the sum of true positive and false negative predictions. It represents the sensitivity or the ability to correctly identify positive samples.

recall = TP / (TP + FN)

F1 Score Calculation:

- The F1 score is the harmonic mean of precision and recall. It combines both measures into a single score.

F1 score = 2 *(precision* recall) / (precision + recall)

The harmonic mean is used instead of a simple average to give more weight to lower values. This means that the F1 score penalizes models that have a large difference between precision and recall.

Output:

- The F1 score is a value between 0 and 1. A higher F1 score indicates better overall performance of the classification model in terms of balancing precision and recall.

In [48]:
```
# F1-Score:
from sklearn.metrics import f1_score

f1 = f1_score(y_test1, y_pred1)
```

Specificity

Specificity, also known as the True Negative Rate (TNR), is a performance metric used in binary classification tasks. It measures the ability of a model to correctly identify negative samples.

Here's how specificity works:

True Negative (TN) Calculation:

- The model correctly predicts negative samples as negative.

False Positive (FP) Calculation:

- The model incorrectly predicts negative samples as positive.

Specificity Calculation:

- Specificity is calculated as the ratio of true negative predictions to the sum of true negative and false positive predictions.

specificity = TN / (TN + FP)

Specificity represents the proportion of actual negative samples that are correctly identified as negative by the model.

Output:

- Specificity is a value between 0 and 1. A higher specificity score indicates better performance of the classification model in correctly identifying negative samples.

In [49]:
```
# Specificity (True Negative Rate):
from sklearn.metrics import confusion_matrix

tn, fp, fn, tp = confusion_matrix(y_test1, y_pred1).ravel()
specificity = tn / (tn + fp)
```

Confusion Matrix

A Confusion Matrix is a table that summarizes the performance of a classification model by showing the counts of true positive, true negative, false positive, and false negative predictions. It provides a detailed breakdown of how well the model is performing in terms of classifying instances correctly or incorrectly.

Confusion Matrix works:

True Positives (TP):

- These are the instances that are actually positive (belong to the positive class) and are correctly predicted as positive by the model.

True Negatives (TN):

- These are the instances that are actually negative (belong to the negative class) and are correctly predicted as negative by the model.

False Positives (FP):

- These are the instances that are actually negative but are incorrectly predicted as positive by the model. Also known as Type I errors.

False Negatives (FN):

- These are the instances that are actually positive but are incorrectly predicted as negative by the model. Also known as Type II errors.

The Confusion Matrix is typically presented in a table format, where the true class labels are represented by rows and the predicted class labels are represented by columns. The table provides a comprehensive view of the model's performance across different classes.

The Confusion Matrix can be used to compute various evaluation metrics for the model, such as accuracy, precision, recall, F1-score, and more. These metrics provide insights into different aspects of the model's performance, such as its ability to correctly identify positive instances (precision) or its ability to capture all positive instances (recall).

The Confusion Matrix is a valuable tool for evaluating the performance of classification models, especially in scenarios where the classes are imbalanced or the cost of false predictions varies for different classes. It allows for a more nuanced understanding of the model's strengths and weaknesses and can guide further model improvements or decision-making processes.

In [50]:
```
# Confusion Matrix:

from sklearn.metrics import confusion_matrix

conf_matrix = confusion_matrix(y_test1, y_pred1)
```

The classification_report

The classification_report is a utility function in Scikit-learn that provides a comprehensive report of various evaluation metrics for a classification model. It summarizes the performance of a classifier by calculating metrics such as precision, recall, F1-score, and support for each class in the target variable.

Here's an explanation of the metrics included in the classification_report:

Precision: Precision is the ratio of true positives to the sum of true positives and false positives. It measures the accuracy of positive predictions. A high precision indicates a low false positive rate.

Recall (also known as Sensitivity or True Positive Rate): Recall is the ratio of true positives to the sum of true positives and false negatives. It measures the ability of the classifier to correctly identify positive instances. A high recall indicates a low false negative rate.

F1-score: The F1-score is the harmonic mean of precision and recall. It provides a balance between precision and recall. A high F1-score indicates a good trade-off between precision and recall.

Support: Support is the number of occurrences of each class in the true dataset. It represents the number of samples in each class and can help identify class imbalances.

In [51]:
```
from sklearn.metrics import classification_report

# Calculate the classification report
report = classification_report(y_test1, y_pred1)

print(report)
```

```
              precision    recall  f1-score   support

           0       1.00      0.78      0.88         9
           1       0.85      1.00      0.92        11

    accuracy                           0.90        20
   macro avg       0.92      0.89      0.90        20
weighted avg       0.92      0.90      0.90        20
```

Metrics Evaluation for Regression

Mean Squared Error (MSE)

Mean Squared Error (MSE) is a commonly used loss function in regression algorithms. It measures the average squared difference between the predicted and actual values of the target variable.

Here's how MSE works:

For each data point in the dataset, the regression algorithm predicts a continuous value based on the input features.

The predicted values are compared to the actual values of the target variable. The difference between the predicted and actual values is calculated.

The differences are squared to ensure positive values and to give more weight to larger errors. Squaring also penalizes larger errors more than smaller errors.

The squared differences are then averaged across all data points to compute the MSE. It represents the average squared error between the predicted and actual values.

Mathematically, the MSE can be calculated using the following formula:

MSE = (1 / n) * Σ(y - ŷ)^2

where:

n is the number of data points in the dataset y is the actual value of the target variable ŷ is the predicted value of the target variable A lower MSE indicates a better fit of the regression model to the data. It quantifies the overall quality of the predictions made by the model. When performing model training, the goal is to minimize the MSE by adjusting the model's parameters or hyperparameters.

MSE is a popular choice as a loss function in regression algorithms due to its simplicity and convex nature, which makes it amenable to optimization techniques. However, it can be sensitive to outliers since the squared differences magnify their impact on the overall error.

In [58]:
```
# Mean Squared Error (MSE)
from sklearn.metrics import mean_squared_error

mse = mean_squared_error(y_test, y_pred)

print(mse)
```

```
0.13549159865935906
```

Root Mean Squared Error (RMSE)

Root Mean Squared Error (RMSE) is a commonly used evaluation metric in regression algorithms. It is derived from the Mean Squared Error (MSE) and provides a more interpretable measure of the average prediction error.

Here's how RMSE works:

For each data point in the dataset, the regression algorithm predicts a continuous value based on the input features.

The predicted values are compared to the actual values of the target variable. The difference between the predicted and actual values is calculated.

The differences are squared to ensure positive values and to give more weight to larger errors.

The squared differences are then averaged across all data points to compute the MSE.

Finally, the square root of the MSE is taken to calculate the RMSE. It represents the average magnitude of the prediction errors in the same units as the target variable.

Mathematically, the RMSE can be calculated using the following formula:

RMSE = sqrt(MSE)

RMSE is a popular metric for evaluating regression models because it has the same scale as the target variable, making it easier to interpret. A lower RMSE indicates a better fit of the regression model to the data, with smaller prediction errors.

Similar to MSE, RMSE is sensitive to outliers since the squared differences magnify their impact on the overall error. It is commonly used in cross-validation and hyperparameter tuning to assess and compare the performance of different regression models.

In [59]:
```
# Root Mean Squared Error (RMSE)
import numpy as np

rmse = np.sqrt(mean_squared_error(y_test, y_pred))

print(rmse)
```

```
0.3680918345458903
```

Mean Absolute Error (MAE)

Mean Absolute Error (MAE) is an evaluation metric commonly used in regression algorithms. It provides a measure of the average absolute difference between the predicted values and the actual values of the target variable.

Here's how MAE works:

For each data point in the dataset, the regression algorithm predicts a continuous value based on the input features.

The predicted values are compared to the actual values of the target variable. The absolute difference between the predicted and actual values is calculated.

The absolute differences are then averaged across all data points to compute the MAE. It represents the average magnitude of the prediction errors.

Mathematically, the MAE can be calculated using the following formula:

MAE = (1 / n) * Σ |y_pred - y_actual|

where n is the number of data points, y_pred is the predicted value, and y_actual is the actual value of the target variable.

MAE is advantageous because it is simple to understand and interpret. It represents the average absolute deviation of the predictions from the actual values. MAE is less sensitive to outliers compared to metrics like Mean Squared Error (MSE) or Root Mean Squared Error (RMSE), as it does not square the differences.

MAE is commonly used in regression tasks where the magnitude of the errors is important and needs to be evaluated. It is often used in combination with other evaluation metrics to comprehensively assess the performance of regression models and compare them against each other.

In [60]:
```
# Mean Absolute Error (MAE)
from sklearn.metrics import mean_absolute_error

mae = mean_absolute_error(y_test, y_pred)

print(mae)
```

```
0.308540405833592
```

R-squared (Coefficient of Determination)

R-squared, also known as the coefficient of determination, is a statistical measure used to evaluate the goodness-of-fit of a regression model. It indicates the proportion of the variance in the dependent variable that can be explained by the independent variables in the model.

Here's how R-squared works:

The regression model is fitted to the data using a specific algorithm, such as linear regression or any other regression algorithm.

The model predicts the values of the dependent variable based on the input features.

The actual values of the dependent variable are compared to the predicted values.

R-squared is calculated as the ratio of the sum of squares of the differences between the predicted values and the mean of the dependent variable, divided by the total sum of squares of the differences between the actual values and the mean of the dependent variable.

Mathematically, R-squared is computed using the formula:

R-squared = 1 - (SSR / SST)

where SSR is the sum of squares of residuals (differences between predicted and actual values), and SST is the total sum of squares (variance of the actual values).

R-squared ranges from 0 to 1, where 0 indicates that the model explains none of the variance in the dependent variable, and 1 indicates that the model explains all of the variance. Higher R-squared values indicate a better fit of the model to the data.

R-squared is a widely used metric for evaluating the performance of regression models. However, it has limitations. R-squared alone does not provide information about the significance or accuracy of the model's predictions, and it can be misleading when applied to complex or overfit models. Therefore, it is often used in conjunction with other evaluation metrics and considerations to assess the overall performance of the model.

In [61]:
```
# R-squared (Coefficient of Determination)
from sklearn.metrics import r2_score

r2 = r2_score(y_test, y_pred)

print(r2)
```

```
0.45011526518117273
```

Variance Score

Variance Score, also known as explained variance ratio or explained variance, is a metric used to measure the proportion of the variance in the dependent variable that is explained by the regression model. It assesses how well the model captures the variability in the target variable.

Here's how Explained Variance Score works:

The regression model is fitted to the data using a specific algorithm, such as linear regression or any other regression algorithm.

The model predicts the values of the dependent variable based on the input features.

The actual values of the dependent variable are compared to the predicted values.

Explained Variance Score is calculated as the ratio of the variance of the predicted values to the variance of the actual values.

Mathematically, Explained Variance Score is computed using the formula:

Explained Variance Score = 1 - (Var(y - y_pred) / Var(y))

where Var(y - y_pred) represents the variance of the difference between the actual and predicted values, and Var(y) represents the variance of the actual values.

Explained Variance Score ranges from 0 to 1, where 0 indicates that the model explains none of the variance in the dependent variable, and 1 indicates that the model explains all of the variance. Higher values indicate a better fit of the model to the data.

Explained Variance Score is particularly useful when comparing different models or assessing the improvement of a model over a baseline. It provides an indication of how much variability in the dependent variable can be accounted for by the model's predictions. However, like R-squared, it should be used in conjunction with other evaluation metrics and considerations to fully evaluate the model's performance.

In [62]:
```
# Variance Score
from sklearn.metrics import explained_variance_score

evs = explained_variance_score(y_test, y_pred)

print(evs)
```

```
0.45045107008245455
```