Introduction to Machine Learning (ML)

Definition:

Machine learning (ML) uses algorithms trained on data to create models that can perform tasks typically requiring human intelligence. These tasks include recognizing patterns, making decisions, and predictions

How it Works:

ML algorithms use a dataset to train a model. The training involves an iterative process where the model makes predictions, compares them to the actual outcomes, and adjusts its parameters to improve accuracy.

Types of Machine Learning:

- Supervised Learning:

  The algorithm learns from a labeled dataset, meaning the correct answer is included in the training data.

- Unsupervised Learning:

  The algorithm must find patterns and relationships in datasets without any labels.

- Reinforcement Learning:

  The algorithm learns by making a sequence of decisions and receiving feedback in the form of rewards or penalties.

  ### Applications:

  Machine learning powers many applications we use daily, such as recommendation systems, search engines, spam filters, and speech recognition systems.Machine learning is a rapidly evolving field that continues to push the boundaries of what's possible with technology, making our interactions with digital systems more intuitive and efficient.

∨ Linear Regression

Definition:

Linear Regression is a statistical method that models the relationship between a dependent variable and one or more independent variables by fitting a linear equation to observed data. The goal is to find the best-fit line or the linear equation that can be used to predict outcomes.

Equation

The linear equation for a simple linear regression, which involves a single independent variable, is:

$$y = \beta_0 + \beta_1 x + \epsilon$$

where:

- y is the dependent variable
- x is the independent variable
- β_0 is the y-intercept
- β_1 is the slope of the line
- ϵ is the error term.

Purpose:

The primary use of Linear Regression is to predict values within a continuous range, rather than trying to classify them into categories. For example, predicting prices of houses, sales of a product, or temperature forecasts.

Assumptions:

Linear Regression assumes that there is a linear relationship between the independent and dependent variables. Other assumptions include homoscedasticity, independence of errors, and normal distribution of errors

Applications:

It's widely used in economics, business, engineering, and the natural and social sciences as it provides a clear understanding of the relationships between variables. Linear Regression is valued for its ease of interpretation and understanding, making it a staple for many predictive modeling tasks

Example:

⌄ Import

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_regression
```

⌄ Generate a random regression problem

```
lrX, lry = make_regression(n_samples=100, n_features=1, noise=10)
```

⌄ Split

```
lrX_train, lrX_test, lry_train, lry_test = train_test_split(lrX, lry, test_size=0.2)
```

⌄ Create linear regression object

```
lr = LinearRegression()
```

⌄ Train the model using the training sets

```
lr.fit(lrX_train, lry_train)
```

```
    ▾  LinearRegression ⓘ ⍰
  LinearRegression()
```

⌄ Make predictions using the testing set
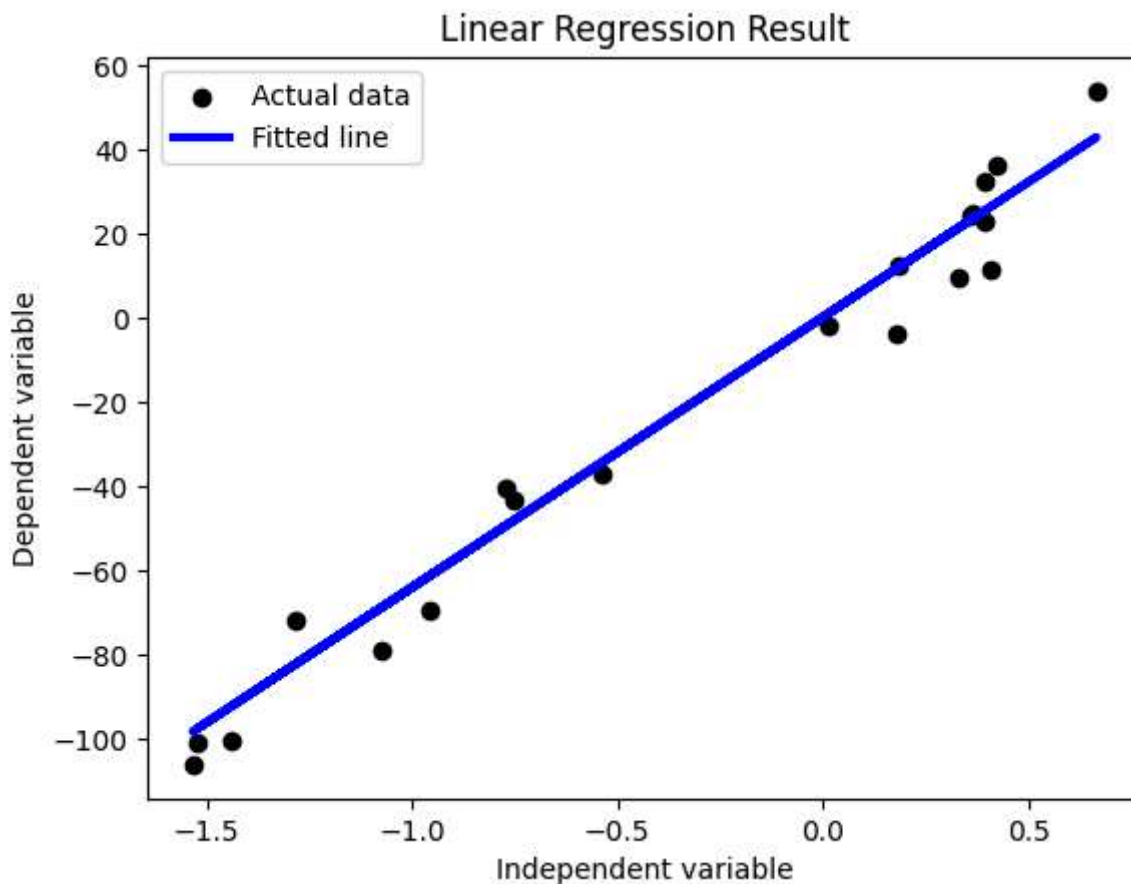
```
lry_pred = lr.predict(lrX_test)
```

⌄ Plot outputs

```
plt.scatter(lrX_test, lry_test, color='black', label='Actual data')
plt.plot(lrX_test, lry_pred, color='blue', linewidth=3, label='Fitted line')

plt.title('Linear Regression Result')
plt.xlabel('Independent variable')
plt.ylabel('Dependent variable')

plt.legend()

plt.show()
```


Validation

To validate your Linear Regression model, you can use several statistical metrics to assess its performance. Here are some common methods:

- R-squared (Coefficient of Determination):

  This metric indicates the proportion of the variance in the dependent variable that is predictable from the independent variables. It ranges from 0 to 1, Generally, a higher R-squared value means that the model fits the data better.

```
from sklearn import metrics

lrr_squared = metrics.r2_score(lry_test, lry_pred)
print('R-squared: ', lrr_squared)
```

�   R-squared: 0.9721095523164569

- ∨   Mean Absolute Error (MAE):

  This is the average of the absolute differences between the predicted values and the actual values. It gives an idea of how big the errors are on average. The Mean Absolute Error (MAE) is considered better when it is closer to 0.

```
lrmae = metrics.mean_absolute_error(lry_test, lry_pred)
print('Mean Absolute Error: ', lrmae)
```

⤴   Mean Absolute Error: 7.148104288854194

- ∨   Mean Squared Error (MSE):

  This is the average of the squares of the errors. It penalizes larger errors more than MAE does.The Mean Squared Error (MSE) is considered better when it is closer to 0.

```
lrmse = metrics.mean_squared_error(lry_test, lry_pred)
print('Mean Squared Error: ', lrmse)
```

⤴   Mean Squared Error: 71.12663664494545

- ∨   Root Mean Squared Error (RMSE):

  This is the square root of MSE. It's in the same units as the dependent variable and is often used to compare different regression models. The Root Mean Squared Error (RMSE) is better when it is as close to 0 as possible.

```
lrrmse = np.sqrt(lrmse)
print('Root Mean Squared Error: ', lrrmse)
```

⤴   Root Mean Squared Error: 8.433660927790816

∨   At the End make it better

```
import pandas as pd


lr_results = pd.DataFrame(['Simple Linear Regression', lrrmse, lrmse, lrmae, lrr_squared]).t
lr_results.columns = ['Method', 'RMSE', 'MSE', 'MAE', 'R2']
lr_results
```

	Method	RMSE	MSE	MAE	R2
0	Simple Linear Regression	8.433661	71.126637	7.148104	0.97211

✓ Residuals plots

Additionally, you can use residual plots to visually check for the validity of your model. Residuals are the differences between the actual values and the predicted values. A well-fitted model will have residuals that are randomly scattered around zero.

```
residuals = lry_test - lry_pred

plt.scatter(lry_pred, residuals, color='red', label='Residuals')
plt.axhline(y=0, color='blue', linestyle='--')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.title('Residual Plot')
plt.legend()
plt.show()
```

Residual Plot
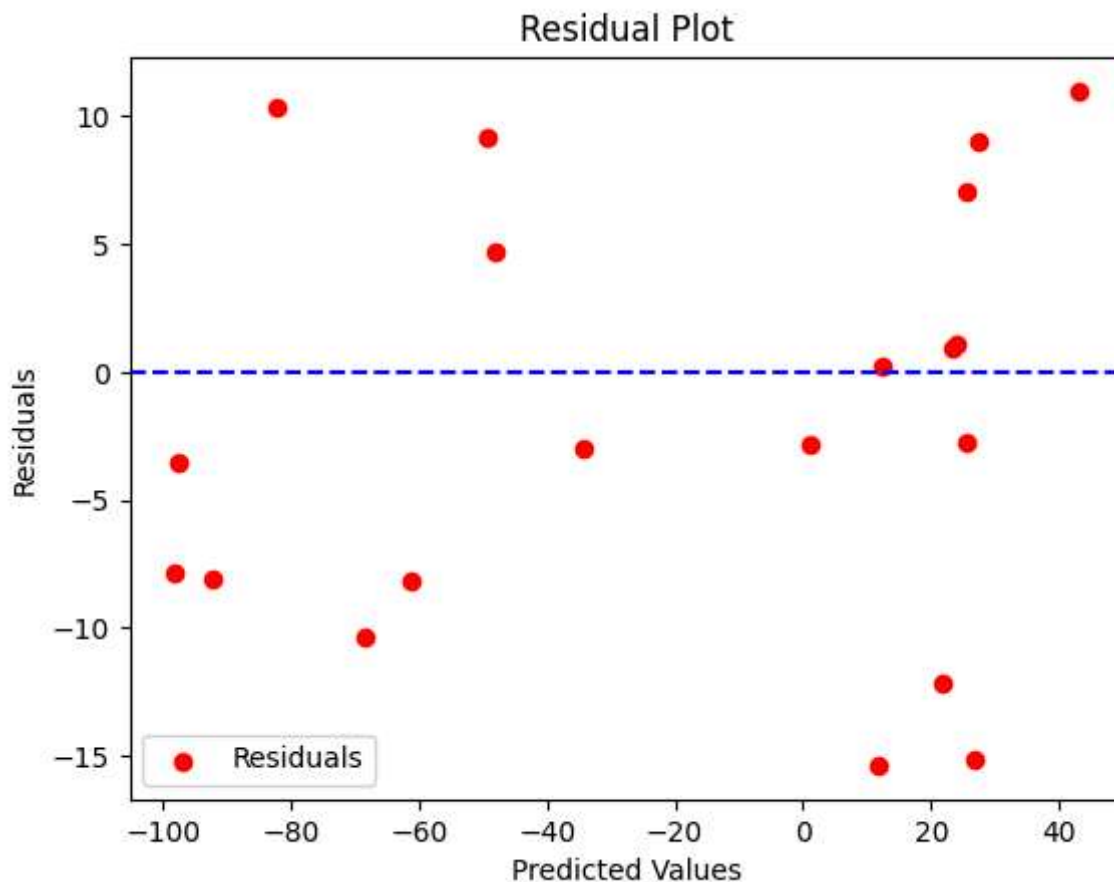
⌄ Support Vector Machine (SVM)

Support Vector Machine (SVM) is a powerful and versatile supervised machine learning algorithm used for both classification and regression tasks, though it is primarily known for classification. Here's an overview of SVM: Objective: The main goal of SVM is to find the optimal hyperplane that separates different classes in the feature space with the maximum margin possible1.

Core Idea:

SVM classifies data by finding the optimal hyperplane which maximizes the margin between different classes in an N-dimensional space.

Hyperplane:

This is the decision boundary that separates different classes. The dimension of the hyperplane depends on the number of input features.

Support Vectors:

These are the data points nearest to the hyperplane, which influence its position. SVM aims to maximize the distance (margin) from these points to the hyperplane

Margin:

It's the distance between the hyperplane and the nearest data point from either class. A larger margin is associated with a lower generalization error of the classifier

Kernels:

SVM can handle linear and non-linear data. For non-linear data, it uses kernel functions to transform the input space into a higher-dimensional space where a linear separator can be used.

Robustness to Outliers:

SVM is known for its robustness to outliers. It focuses on the support vectors and the margin, which helps in ignoring the outliers while finding the hyperplane1.

Applications:

SVM is used in various domains like text classification, image recognition, spam detection, and more due to its effectiveness in high-dimensional spaces and its ability to handle nonlinear relationships.

SVM's ability to find the maximum separating hyperplane by considering the closest points of different classes makes it a strong model, especially when dealing with complex datasets.

Example:

⌄ Import

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
```

⌄ Load The Iris Dataset from `sklearn`

This data sets consists of 3 different types of irises' (Setosa, Versicolour, and Virginica) petal and sepal length, stored in a 150x4 numpy.ndarray

The rows being the samples and the columns being: Sepal Length, Sepal Width, Petal Length and Petal Width.
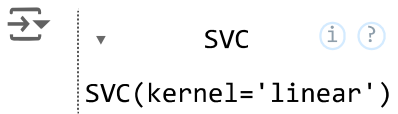
```
iris = datasets.load_iris()
svmX = iris.data[:, :2]
svmy = iris.target
```

⌄ Split

```
svmX_train, svmX_test, svmy_train, svmy_test = train_test_split(svmX, svmy, test_size=0.2, r
```

⌄ Create an SVM classifier with a linear kernel

```
clf = SVC(kernel='linear')
clf.fit(svmX_train, svmy_train)
```

```
        ▾        SVC        ⓘ ⓘ
    SVC(kernel='linear')
```

⌄ Predict using the test set

```
svmy_pred = clf.predict(svmX_test)
```

⌄ Create a mesh to plot the decision boundaries

```
h = .02  # step size in the mesh
svmx_min, svmx_max = svmX[:, 0].min() - 1, svmX[:, 0].max() + 1
svmy_min, svmy_max = svmX[:, 1].min() - 1, svmX[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(svmx_min, svmx_max, h), np.arange(svmy_min, svmy_max, h))
```

⌄ Plot outputs

```
# Plot the decision boundary by assigning a color to each point in the mesh
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.8)

# Plot also the training points
plt.scatter(svmX_train[:, 0], svmX_train[:, 1], c=svmy_train, cmap=plt.cm.coolwarm, edgecolc
# and testing points
plt.scatter(svmX_test[:, 0], svmX_test[:, 1], c=svmy_test, cmap=plt.cm.coolwarm, edgecolors=

plt.xlabel('Sepal length')
plt.ylabel('Sepal width')
plt.title('SVM on Iris Dataset')

# Show the plot
plt.show()
```
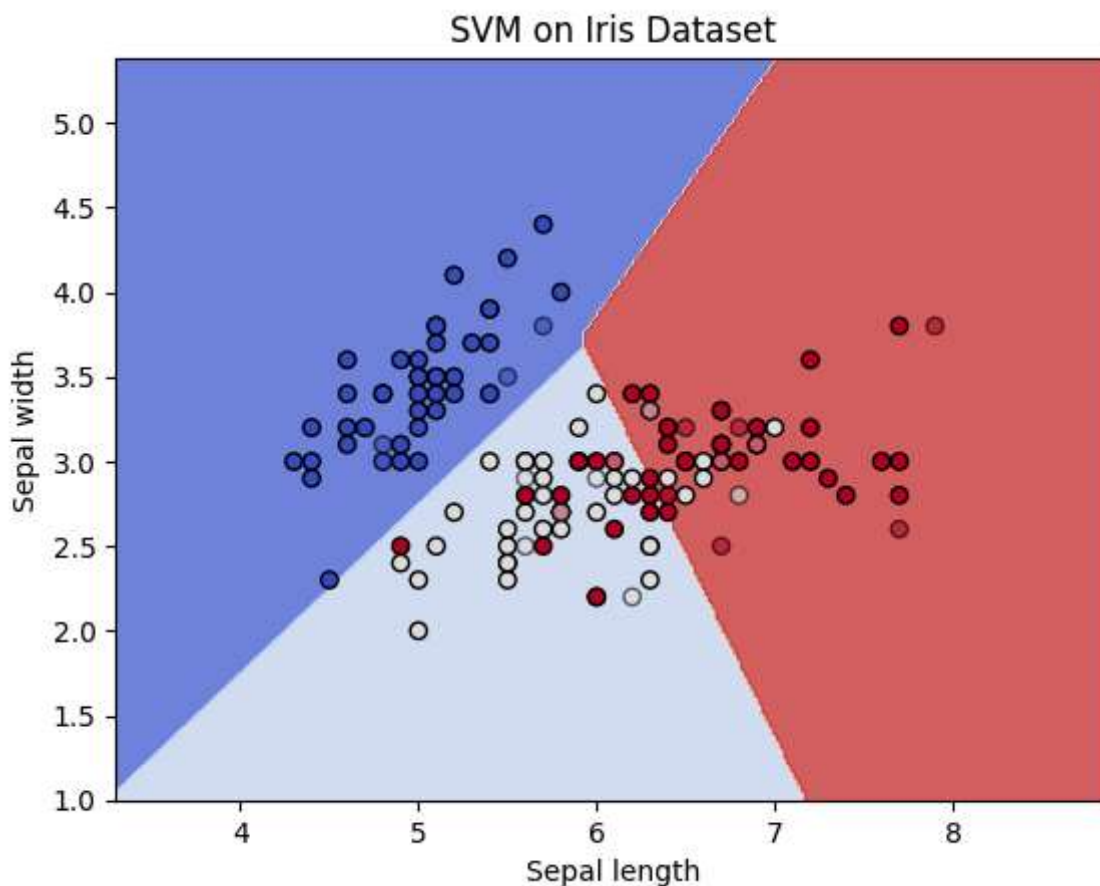

Validation

Confusion Matrix

A Confusion Matrix is a table used in machine learning to evaluate the performance of a classification model on a set of test data. It's particularly useful for visualizing the accuracy of a

model and understanding its errors.

```
cm = confusion_matrix(svmy_test, svmy_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=clf.classes_)
disp.plot()

plt.show()
```
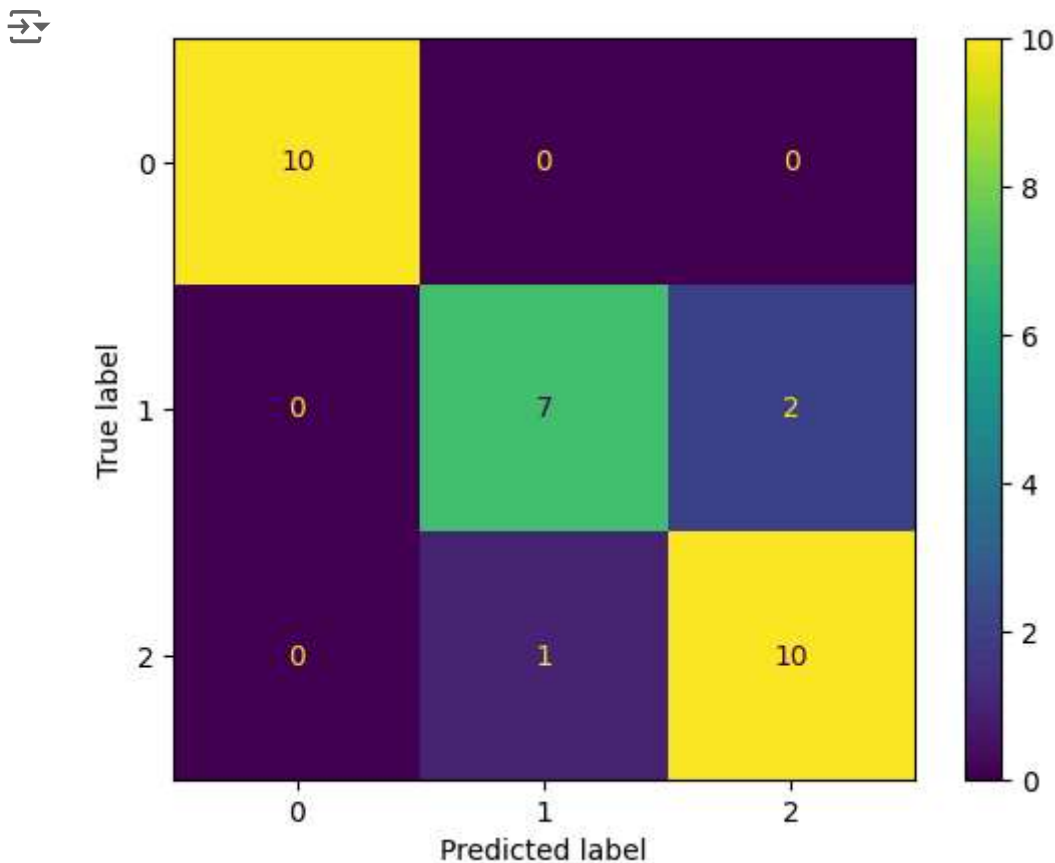

K Means Clustering

K-means is an unsupervised learning algorithm used in machine learning for clustering analysis. In unsupervised learning, algorithms are used to group unlabelled datasets, and k-means is particularly effective for partitioning a dataset into distinct clusters based on similarity. Here's a high-level overview of how the k-means algorithm works:

Initialization:

Choose the number of clusters, (K), and select (K) random points as cluster centers or centroids.

Assignment:

Assign each data point to the nearest centroid, forming (K) clusters.

Update:

Calculate the new centroids (mean) of the clusters by taking the average of all data points in the cluster.

Iteration:

Repeat the assignment and update steps until the centroids no longer change significantly, indicating that the clusters are stable.

The goal of k-means is to minimize the within-cluster variance, which is the sum of the squared distances between each data point and its corresponding centroid. This process iteratively refines the positions of the centroids, resulting in a set of clusters where each data point is closer to its own cluster's centroid than to any other.

K-means is widely used because it's simple, efficient, and can handle large datasets. However, it does have some limitations, such as sensitivity to the initial placement of centroids and difficulty in clustering data with varying shapes and densiti. It's also important to predefine the value of (K), which may not be straightforward and often requires additional methods like the elbow method to estimate the optimal number of clusters.es

⌄ Example:

⌄ Import

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
```

⌄ Generate synthetic data

```
X, _ = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)
```

The line `X, _ = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)` is a function call to `make_blobs`, which is a convenient tool from the `scikit-learn` library used to generate synthetic datasets, particularly for clustering algorithms like K-Means.

- `n_samples=300`: This specifies the total number of data points to generate. In this case, it's set to 300.

- `centers=4`: This determines the number of centers to generate, or in other words, the number of clusters. Here, we are generating data that should ideally be grouped into 4 clusters

- `cluster_std=0.60`: This parameter sets the standard deviation of the clusters. The standard deviation controls the spread of the data points around the cluster center. A smaller value will make the clusters tighter and more distinct, while a larger value will make them more spread out and possibly overlapping. Here, it's set to 0.6

- `random_state=0`: This is a seed for the random number generator. Using the same seed ensures that the random numbers are generated in the same order, which means that the function will produce the same results every time it's run with that seed. This is useful for reproducibility.

The function returns two values:

- `X`: This is the array of shape [n_samples, n_features] containing the generated data points.
- `_`: This is a placeholder for the second return value, which is the true integer labels for cluster membership of each sample. The underscore is used because we don't need these labels for the K-Means algorithm, as it is an unsupervised learning method.

```
plt.scatter(X[:, 0], X[:, 1], s=50, cmap='jet')
plt.title('Raw Data')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```

C:\Users\Arad\AppData\Local\Temp\ipykernel_16620\523021443.py:1: UserWarning: No data fo
    plt.scatter(X[:, 0], X[:, 1], s=50, cmap='jet')

Raw Data
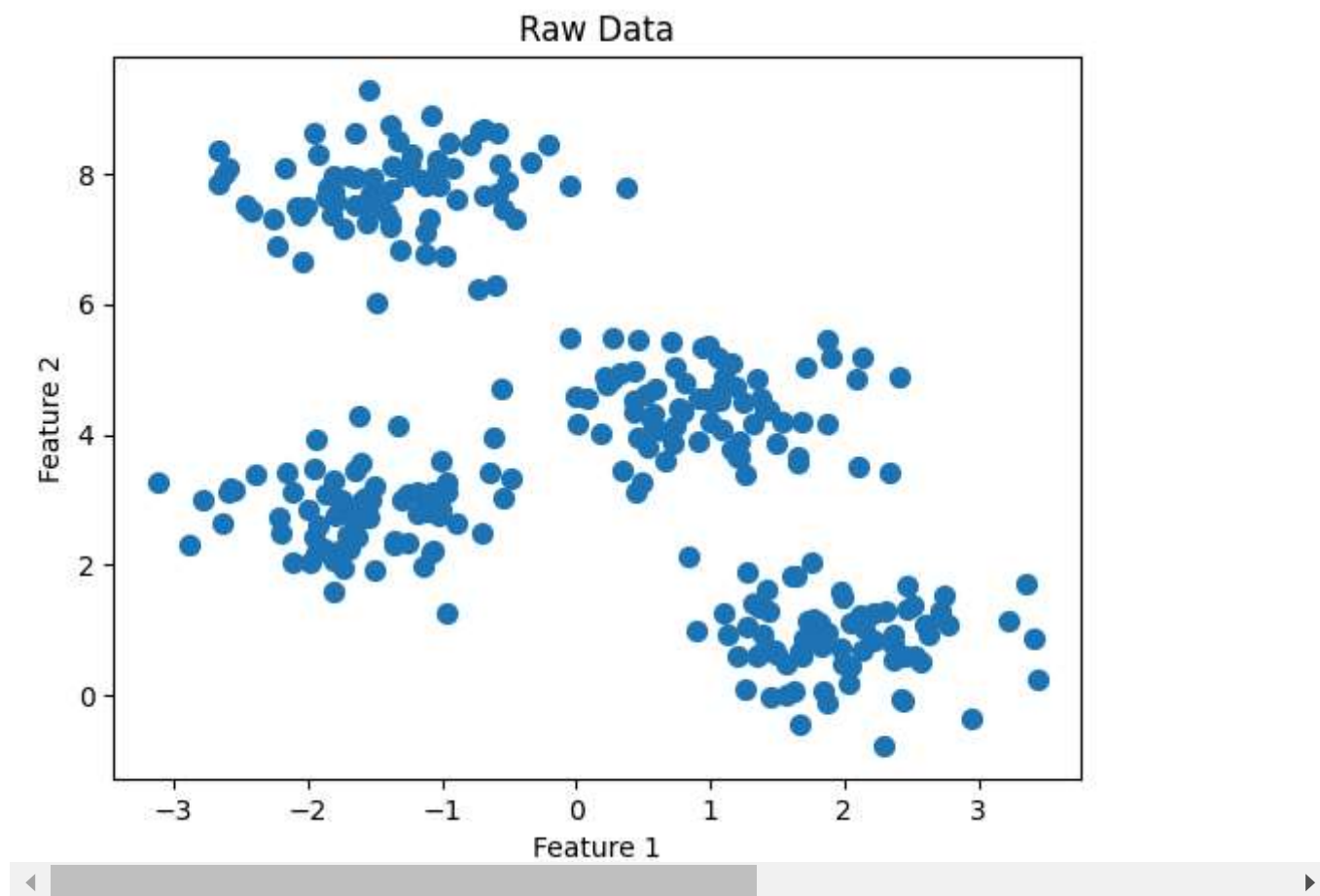
✓ Apply K-Means clustering

```
kmeans = KMeans(n_clusters=4)
kmeans.fit(X)
```
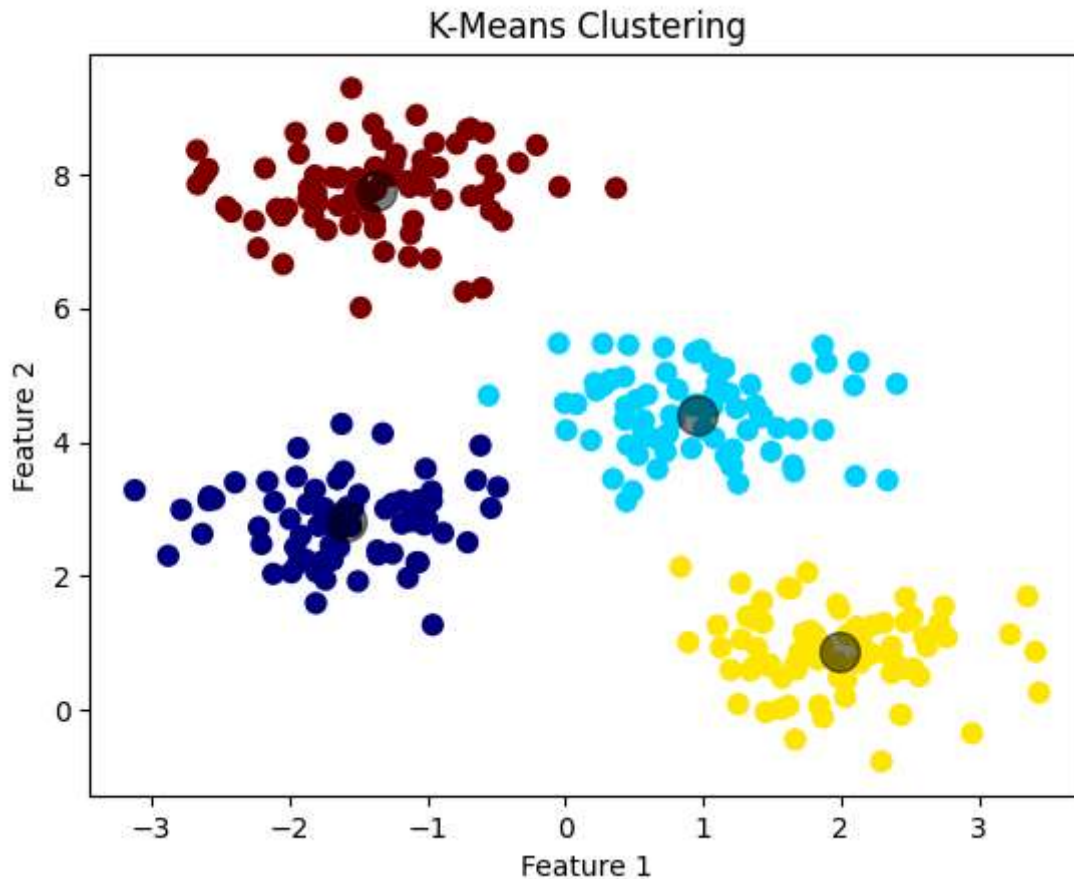
```
▼        KMeans        ⓘ ⓘ

KMeans(n_clusters=4)
```

✓ Predict

```
y_kmeans = kmeans.predict(X)
```

✓ Plot the Outputs

```
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='jet')
centers = kmeans.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], c='black', s=200, alpha=0.5)
plt.title('K-Means Clustering')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```


﹀ Validation

﹀ Silhouette Score

he silhouette score is a metric used to evaluate the quality of clusters in a clustering algorithm. It measures how similar an object is to its own cluster (cohesion) compared to other clusters (separation). The silhouette score ranges from -1 to +1, where:

- A high value close to +1 indicates that the object is well matched to its own cluster and poorly matched to neighboring clusters, suggesting the object is in the appropriate cluster.
- A value of 0 indicates that the object is on or very close to the decision boundary between two neighboring clusters, which may suggest overlapping clusters.
- Negative values indicate that the object might have been assigned to the wrong cluster.

Equation

The silhouette score for the data point (i) is then calculated as:

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}$$

Where:

- (a(i)) is the mean distance between (i) and all other data points in the same cluster.
- (b(i)) is the smallest mean distance from (i) to all points in any other cluster, of which (i) is not a member.

```
silhouette_avg = silhouette_score(X, y_kmeans)
print(f'Silhouette Score: {silhouette_avg}')
```

➔▾ Silhouette Score: 0.6819938690643478

```
print(f'Silhouette Score: {silhouette_avg:.2f}')
```

➔▾ Silhouette Score: 0.68

Principal Component Analysis (PCA)

PCA, or Principal Component Analysis, is a statistical technique used for dimensionality reduction in data analysis. It transforms a large set of variables, which may be correlated, into a smaller set of uncorrelated variables called principal components. These principal components are obtained by orthogonal transformation and capture the most significant variance in the data, making them useful for exploratory data analysis, visualization, and preprocessing data for machine learning algorithms.

The process involves the following steps:

Standardization:

The data is standardized to have a mean of zero and a standard deviation of one.

Covariance Matrix Computation:

A covariance matrix is computed to understand how variables in the data are varying from the mean with respect to each other.

Eigenvalue and Eigenvector Calculation:

Eigenvalues and eigenvectors of the covariance matrix are calculated. Eigenvectors determine the directions of the new feature space, and eigenvalues determine their magnitude.

Sorting and Selecting Principal Components:

Eigenvectors are sorted by their eigenvalues in descending order to rank the principal components. The top principal components are selected based on the amount of variance they capture from the data.

Transformation:

The original data is transformed into the new subspace using the selected principal components.

PCA is widely used in fields like finance, genetics, computer vision, and many others where data dimensionality poses a challenge. It helps to simplify the complexity in high-dimensional data while retaining trends and patterns.

⌄ Example:

⌄ Import

```
import numpy as np
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

⌄ Hypothetical dataset (features X and target y)

```
pcX = np.random.rand(100, 10)  # 100 samples with 10 features each
pcy = np.random.rand(100)      # 100 target values
```

⌄ Split

```
pcX_train, pcX_test, pcy_train, pcy_test = train_test_split(pcX, pcy, test_size=0.2, random_
```

˅ Standardizing the features

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(pcX_train)
X_test_scaled = scaler.transform(pcX_test)
```

˅ Applying PCA

```
pca = PCA(n_components=0.95)  # Keep 95% of variance
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)
```

˅ Plot the Outputs

```
import matplotlib.pyplot as plt
plt.figure(figsize=(8, 6))
plt.scatter(X_train_pca[:, 0], X_train_pca[:, 1], c=y_train, cmap='viridis', edgecolor='k',
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('2D PCA Result')
plt.colorbar()
plt.show()
```

2D PCA Result
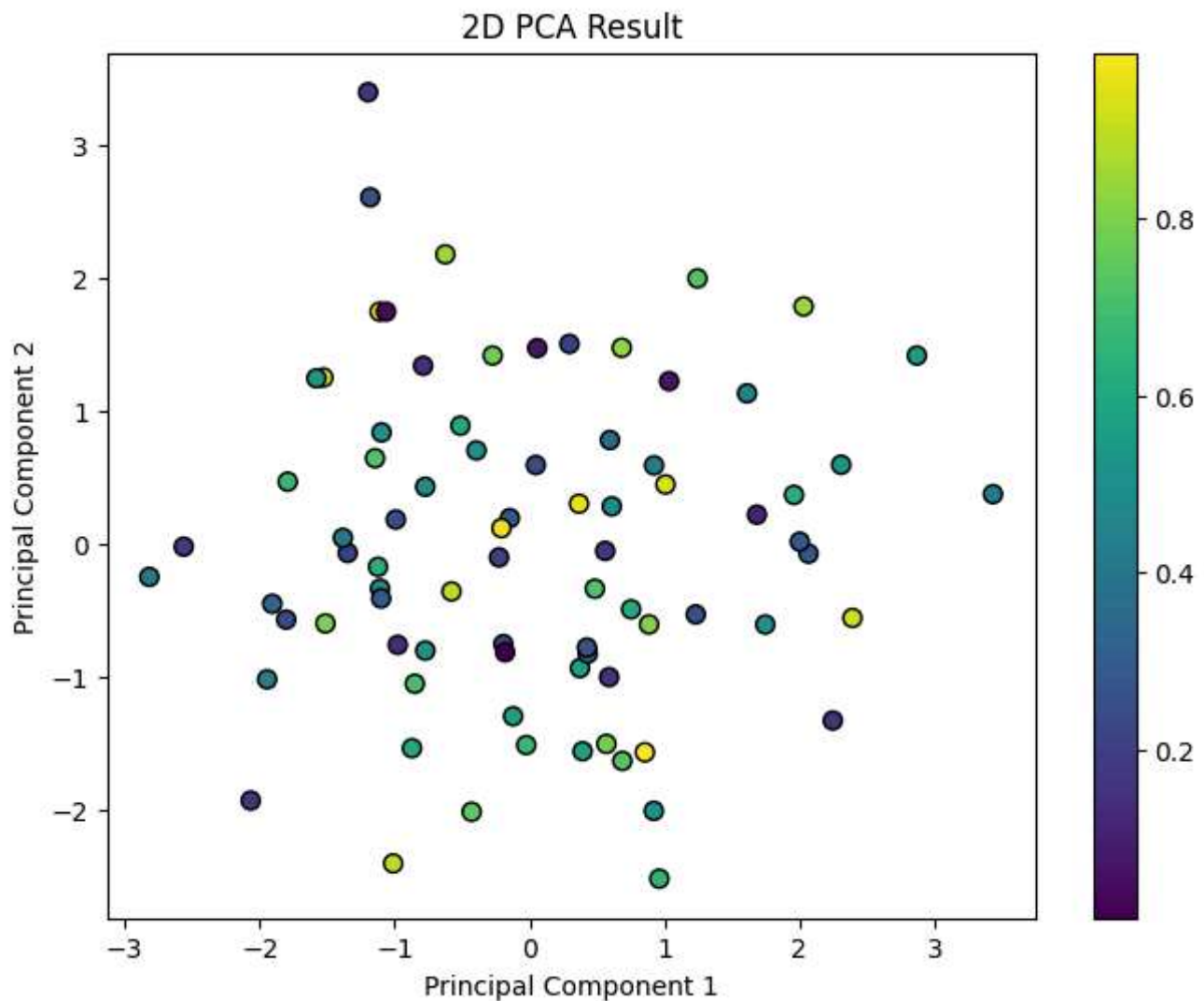
⌄ Validation

⌄ Explained Variance Ratio

The explained variance ratio in PCA (Principal Component Analysis) represents the proportion of the dataset's total variance that is explained by each principal component. It is calculated by dividing the variance captured by a principal component by the total variance in the dataset.

$$\text{Explained Variance Ratio}_i = \frac{\text{Variance of } i^{th} \text{ component}}{\text{Total Variance}}$$

```
explained_variance = pca.explained_variance_ratio_
print(f'Explained Variance Ratio: {explained_variance}')
```

```
Explained Variance Ratio: [0.17144806 0.14064212 0.11766196 0.10744755 0.09574796 0.0909⟨
 0.07899548 0.07084091 0.06829907 0.05800297]
```

Cumulative Explained Variance

The cumulative explained variance in PCA (Principal Component Analysis) refers to the cumulative sum of the explained variances of the principal components. It indicates the total variance explained by the first (n) components. This is useful for determining how many components are needed to capture a certain percentage of the total variance in the dataset.

$$\text{Cumulative Explained Variance}_n = \sum_{i=1}^{n} \text{Explained Variance Ratio}_i$$

```python
cumulative_explained_variance = np.cumsum(explained_variance)
print(f'Cumulative Explained Variance: {cumulative_explained_variance}')
```

```
Cumulative Explained Variance: [0.17144806 0.31209018 0.42975214 0.53719969 0.63294765 6
 0.80285705 0.87369796 0.94199703 1.          ]
```

Numeric Narratives

```python
plt.figure(figsize=(10, 5))
plt.bar(range(1, len(explained_variance) + 1), explained_variance, alpha=0.5, align='center'
plt.step(range(1, len(cumulative_explained_variance) + 1), cumulative_explained_variance, wh
plt.ylabel('Explained variance ratio')
plt.xlabel('Principal component index')
plt.legend(loc='best')
plt.tight layout()
```