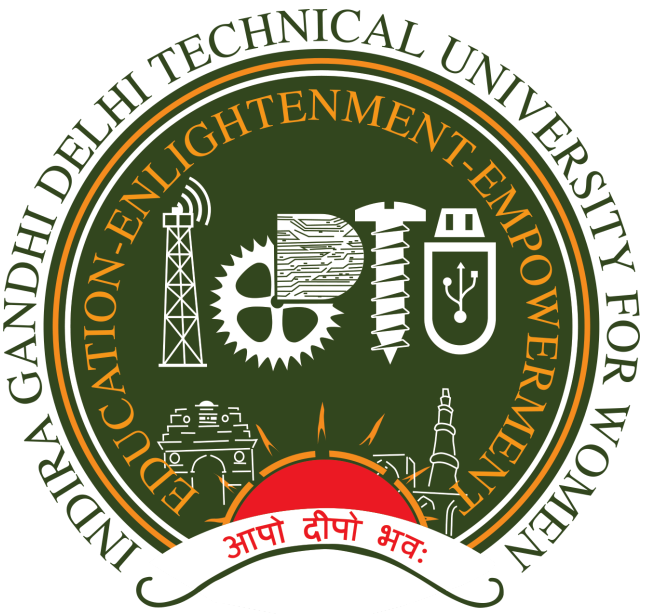
Indira Gandhi Delhi Technical University for Women



MACHINE LEARNING (LAB FILE)
(BAI-301)

BTech - 5th Semester
Department of Artificial Intelligence and Data Science
Academic Session: 2024-2025

Submitted To

Dr. Ritika Kumari

Submitted By:

Name: Tanisha Bansal
Roll No: 14301172022
Batch: B.Tech CSE-AI-2 (2026)

INDEX

S.NO	EXPERIMENTS	REMARKS
1	Write a program to perform visualization and interpretation of data using 1. Histogram 2. Scatter Plot 3. Bar Graph 4. Pie Chart	
2	Write a program to perform cleaning of the data 1. Drop a variable 2. Remove null values 3. Duplicates removal	
3	Write a program to perform data pre-processing 1. Take a data set that has to be either biased (majority - minority) 2. Standardization 3. Normalization 4. SMOTE technique(balance and clean the data)	
4	Write a program to perform data prediction through 1. Supervised (Random Forest) 2. Unsupervised (k means clustering)	
5	Write a program to implement reinforcement learning algorithm on a dataset.	
6	Write a program to perform the data sampling and estimation using Density-based clustering method.	
7	Write a program to perform 1. Model Regularization 2. PCA 3. Optimisation using feature selection methods	
8	Write a program to perform kernel based SVM model. 1. Gaussian (Radial Basis Function) 2. Sigmoid kernel	
9	Write a program to perform model regularisation and optimisation using ensemble methods.	
10	Write a program to perform the over-sampling and under sampling of a considered data in machine learning.	

EXPERIMENT - 1

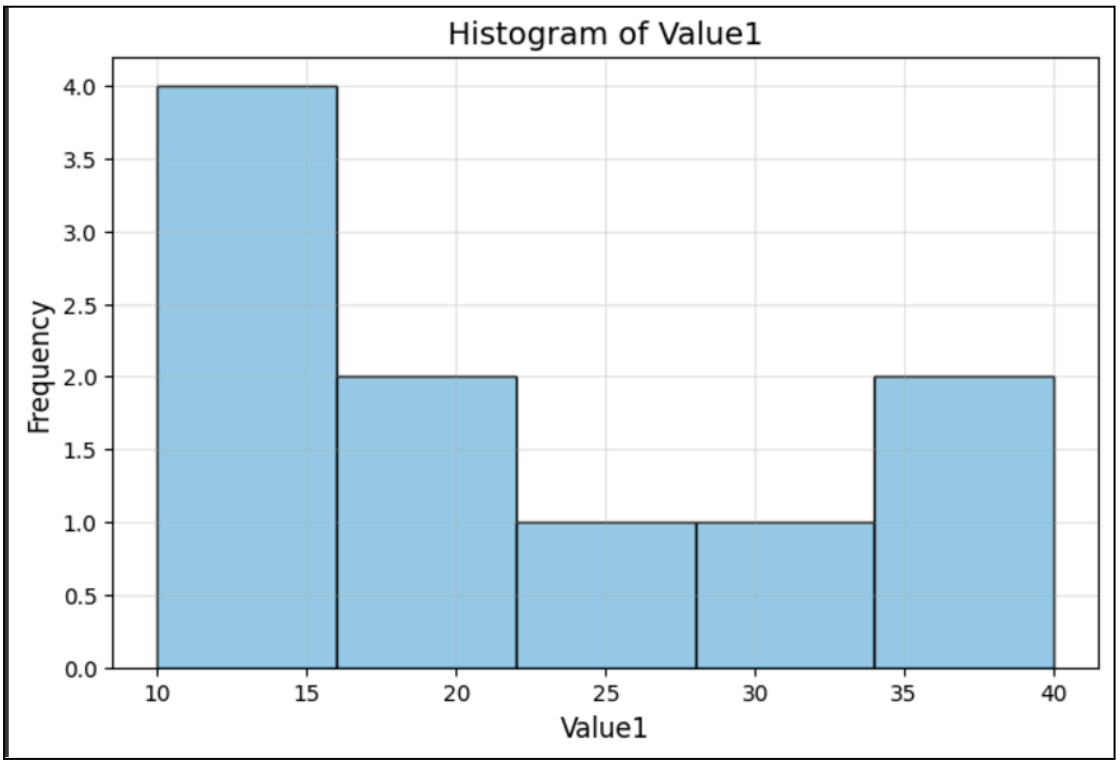
Aim: Write a program to perform visualization and interpretation of data using:-

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

data = {
    'Category': ['A', 'B', 'C', 'D', 'A', 'C', 'B', 'D', 'A', 'C'],
    'Value1': [10, 15, 10, 20, 30, 40, 20, 10, 25, 35],
    'Value2': [5, 7, 10, 15, 8, 12, 5, 6, 8, 10]
}
df = pd.DataFrame(data)
```

```
plt.figure(figsize=(8, 5))
plt.hist(df['Value1'], bins=5, color='skyblue', edgecolor='black')
plt.title('Histogram of Value1', fontsize=14)
plt.xlabel('Value1', fontsize=12)
plt.ylabel('Frequency', fontsize=12)
plt.grid(alpha=0.3)
plt.show()
```

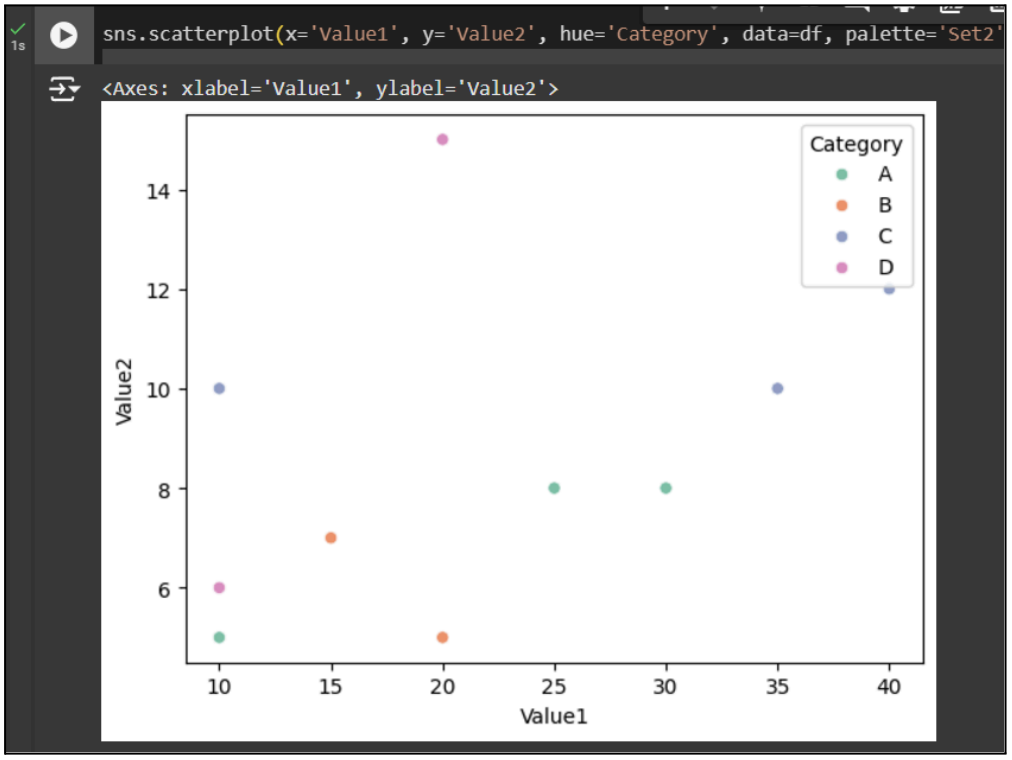
a) Histogram

A **histogram** is a graphical representation of the distribution of numerical data. It groups data into bins or intervals and displays the frequency (or count) of data points in each bin as bars. It is used to visualize the distribution, skewness, and spread of continuous data.



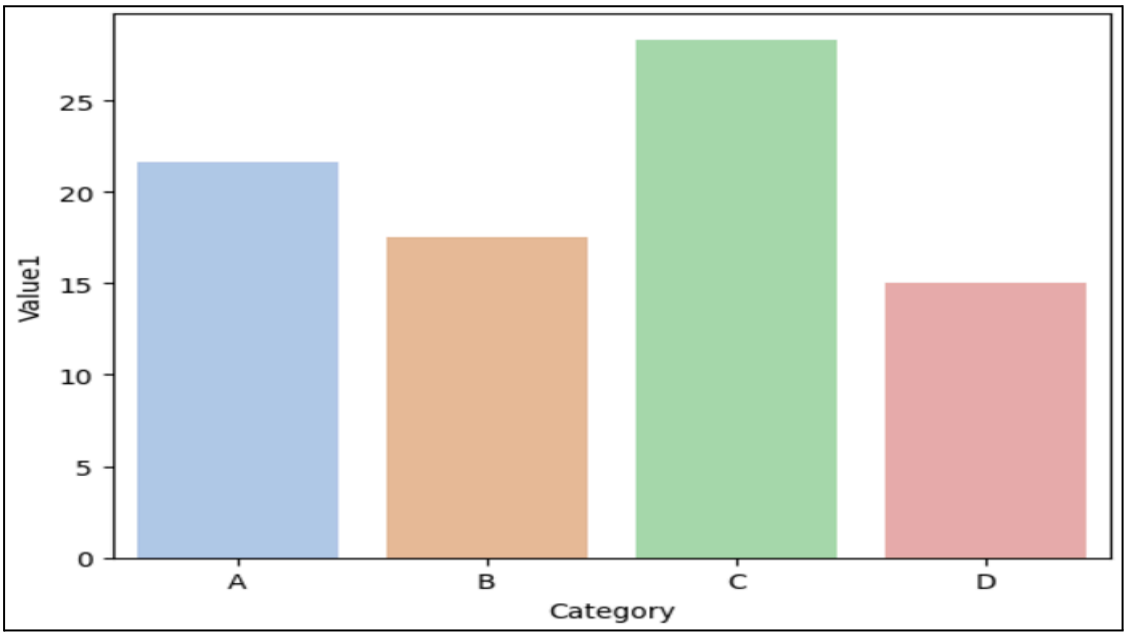
b) Scatter Plot

A **scatter plot** is a type of plot that uses dots to represent values for two continuous variables. Each point's position on the horizontal and vertical axes corresponds to the values of the two variables. It is used to visualize relationships, trends, or correlations between variables.



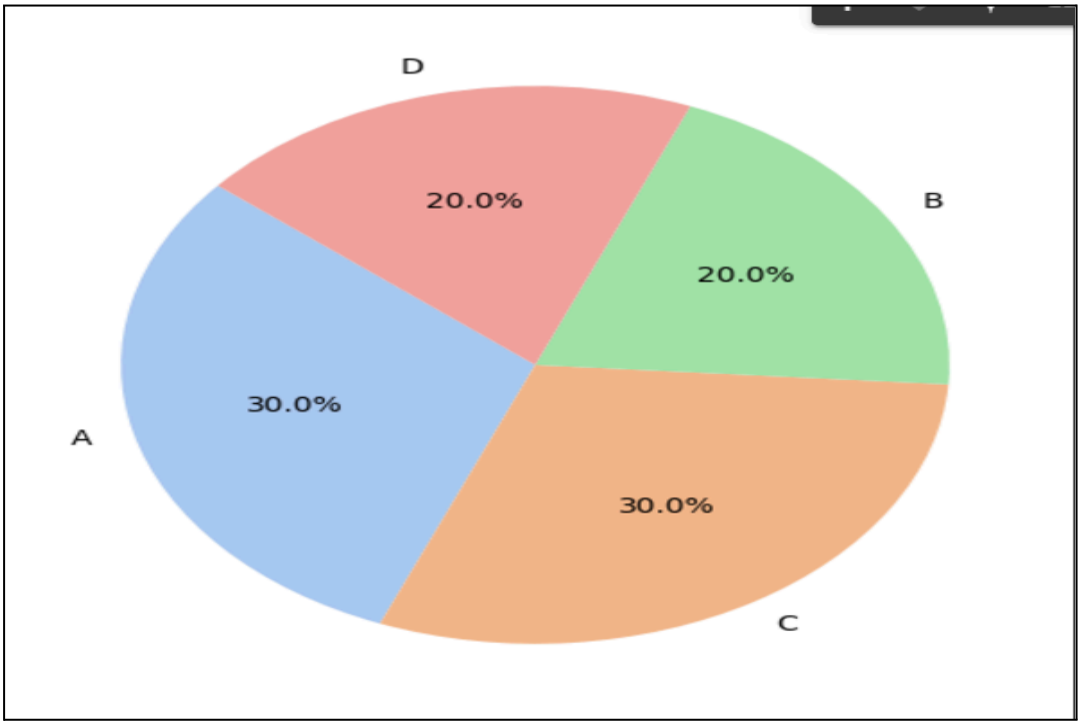
c) Bar Graph

A **bar plot** is a chart that represents categorical data with rectangular bars. The height or length of each bar corresponds to the value or frequency of the category. It is used to compare different categories or groups of data.



d) Pie Chart

A **pie chart** is a circular chart divided into slices to represent proportions of a whole. Each slice corresponds to a category and is sized according to its relative proportion or percentage of the total. It is often used to show parts of a whole in categorical data.



EXPERIMENT - 2

Aim: Write a program to perform cleaning of the data

1. Drop a variable

Removing a feature or column from the dataset that is not relevant or necessary for analysis or modeling.

```
import pandas as pd

data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', None, 'Alice'],
    'Age': [24, None, 22, 35, 29, 24],
    'City': ['New York', 'Los Angeles', None, 'Chicago', 'New York', 'New York'],
    'Score': [85, 90, None, 88, 92, 85]
}
df = pd.DataFrame(data)

column_to_drop = 'Score'
if column_to_drop in df.columns:
    df_dropped = df.drop(columns=[column_to_drop])
    print(f"\nDataFrame after dropping column '{column_to_drop}':")
    print(df_dropped)
```

```
DataFrame after dropping column 'Score':
      Name   Age         City
0    Alice  24.0     New York
1      Bob   NaN  Los Angeles
2  Charlie  22.0         None
3    David  35.0      Chicago
4     None  29.0     New York
5    Alice  24.0     New York
```

2. Remove null values

Deleting or filling missing values (nulls) in a dataset to ensure complete and clean data for analysis.

```
df_no_nulls = df.dropna()
print("\nDataFrame after removing rows with null values:")
print(df_no_nulls)
```

```
DataFrame after removing rows with null values:
    Name   Age      City  Score
0  Alice  24.0  New York   85.0
3  David  35.0   Chicago   88.0
5  Alice  24.0  New York   85.0
```

3. Duplicates removal

Identifying and removing repeated rows or entries in a dataset to avoid biased or inaccurate analysis.

```
df_no_duplicates = df.drop_duplicates()
print("\nDataFrame after removing duplicate rows:")
print(df_no_duplicates)
```

```
DataFrame after removing duplicate rows:
      Name   Age         City  Score
0    Alice  24.0     New York   85.0
1      Bob   NaN  Los Angeles   90.0
2  Charlie  22.0         None    NaN
3    David  35.0      Chicago   88.0
4     None  29.0     New York   92.0
```

EXPERIMENT - 3

Aim: Write a program to perform data pre-processing

1. Take a data set that has to be either biased (majority -minority)

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.datasets import make_classification
from imblearn.over_sampling import SMOTE
import matplotlib.pyplot as plt

X, y = make_classification(n_classes=2, class_sep=2, weights=[0.9, 0.1],
                           n_informative=3, n_redundant=1, flip_y=0,
                           n_features=5, n_clusters_per_class=1, n_samples=1000, r

df = pd.DataFrame(X, columns=[f'Feature_{i}' for i in range(1, 6)])
df['Target'] = y

print("Original Dataset Class Distribution:")
print(df['Target'].value_counts())
```

```
Original Dataset Class Distribution:
Target
0    900
1    100
Name: count, dtype: int64
```

2. Standardization

Standardization is a data preprocessing technique that transforms features to have a mean of zero and a standard deviation of one. It ensures that all features are on the same scale, which is particularly important for algorithms that rely on distance calculations, such as k-nearest neighbors (KNN), or gradient-based methods like logistic regression and neural networks.

$$X\_new = (X - mean)/Std$$

```
scaler = StandardScaler()
standardized_features = scaler.fit_transform(df.drop('Target', axis=1))

df_standardized = pd.DataFrame(standardized_features, columns=[f'Feature_{i}' for i
df_standardized['Target'] = y

print("\nStandardized Dataset:")
print(df_standardized.head())
```

```
Standardized Dataset:
   Feature_1  Feature_2  Feature_3  Feature_4  Feature_5  Target
0   1.152671  -0.561341  -0.355004   1.171796  -0.504032       0
1  -0.424585   0.370879   0.004243  -0.328198  -0.123348       0
2  -0.226418   0.258107   0.475856   0.493169  -0.677500       0
3   0.359511  -0.500827  -0.179598   0.401988  -0.292253       0
4  -0.954013  -0.852521   0.102875  -1.063277   0.368942       0
```

3. Normalization

Normalization (often called **Min-Max scaling**) is a technique that transforms features by scaling them to a fixed range, usually [0, 1] or [-1, 1]. This is done by subtracting the minimum value and dividing by the range of the data (max - min).

$$X\_new = (X - X\_min)/(X\_max - X\_min)$$

```
normalizer = MinMaxScaler()
normalized_features = normalizer.fit_transform(df.drop('Target', axis=1))

df_normalized = pd.DataFrame(normalized_features, columns=[f'Feature_{i}' for i in
df_normalized['Target'] = y

print("\nNormalized Dataset:")
print(df_normalized.head())
```

```
Normalized Dataset:
   Feature_1  Feature_2  Feature_3  Feature_4  Feature_5  Target
0   0.769621   0.415069   0.342741   0.704099   0.450796       0
1   0.532611   0.578676   0.406113   0.470654   0.485004       0
2   0.562389   0.558884   0.489306   0.598483   0.435208       0
3   0.650435   0.425689   0.373683   0.584293   0.469826       0
4   0.453056   0.363966   0.423512   0.356253   0.529241       0
```

- **Standardization** rescales the data to have a mean of 0 and standard deviation of 1, without being bound to a specific range.
- **Normalization** rescales the data to a specific range (usually [0, 1]), which may be important for certain algorithms.

4. Synthetic Minority Oversampling Technique (SMOTE) (balance and clean the data)

SMOTE is a data augmentation technique used to address the class imbalance problem in datasets. It works by generating synthetic samples for the minority class by:

1. Selecting a sample from the minority class.
2. Finding its nearest neighbors within the minority class.
3. Creating synthetic data points along the line segments connecting the selected sample and its neighbors.

```
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(df.drop('Target', axis=1), df['Target'

df_resampled = pd.DataFrame(X_resampled, columns=[f'Feature_{i}' for i in range(1, 6
df_resampled['Target'] = y_resampled

print("\nAfter Applying SMOTE (Balanced Dataset Class Distribution):")
print(df_resampled['Target'].value_counts())
```

```
After Applying SMOTE (Balanced Dataset Class Distribution):
Target
0    900
1    900
Name: count, dtype: int64
```

EXPERIMENT - 4

Aim: Write a program to perform data prediction through

1. Supervised (Random Forest)

Supervised learning involves training a model on labeled data to make predictions. **Random Forest** is an ensemble method that uses multiple decision trees to improve accuracy and reduce overfitting, making it suitable for both classification and regression tasks.

```
X_train, X_test, y_train, y_test = train_test_split(df.drop('Target', axis=1), df['
                                                    test_size=0.3, random_state=42)

rf = RandomForestClassifier(random_state=42, n_estimators=100)
rf.fit(X_train, y_train)

y_pred = rf.predict(X_test)
print("\nRandom Forest - Classification Report:")
print(classification_report(y_test, y_pred))
print(f"Random Forest - Accuracy: {accuracy_score(y_test, y_pred):.2f}")
```

```
Random Forest - Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.96      0.97       151
           1       0.96      0.98      0.97       149

    accuracy                           0.97       300
   macro avg       0.97      0.97      0.97       300
weighted avg       0.97      0.97      0.97       300

Random Forest - Accuracy: 0.97
```

2. Unsupervised (k means clustering)

Unsupervised learning involves finding patterns or groupings in data without labeled targets. **K-Means** is a clustering algorithm that partitions data into kkk clusters based on similarity, helping to discover inherent groupings within the data.

```
kmeans = KMeans(n_clusters=2, random_state=42)
clusters = kmeans.fit_predict(df.drop('Target', axis=1))

df['Cluster'] = clusters

pca = PCA(n_components=2)
df_pca = pca.fit_transform(df.drop(['Target', 'Cluster'], axis=1))
df_pca = pd.DataFrame(df_pca, columns=['PCA1', 'PCA2'])
df_pca['Cluster'] = clusters
print("\nK-Means Clustering - Cluster Distribution:")
print(df['Cluster'].value_counts())
```

```
K-Means Clustering - Cluster Distribution:
Cluster
0    674
1    326
Name: count, dtype: int64
```

EXPERIMENT - 5

Aim: Write a program to implement a reinforcement learning algorithm on a dataset.

Reinforcement Learning (RL) is a branch of machine learning focused on making decisions to maximize cumulative rewards in a given situation.

- **Agent:** The learner or decision-maker.
- **Environment:** Everything the agent interacts with.
- **State:** A specific situation in which the agent finds itself.
- **Action:** All possible moves the agent can make.
- **Reward:** Feedback from the environment based on the action taken.

CartPole Environment in OpenAI Gym

The CartPole environment is a classic reinforcement learning problem where the goal is to balance a pole on a cart by applying forces to the left or right.

1. **Environment Setup:** We load the CartPole environment with gym.make and specify render_mode=”human” to visualize the environment.

2. **State and Action Spaces:** We print the state space and action space to understand the dimensions and types of actions available.

3. **Random Actions:** The agent takes random actions for a few steps, and the state transitions, rewards, and other information are printed for each step.

4. **Termination Handling:** If an episode ends (i.e., the pole falls), the environment is reset to start a new episode.

```python
import gym
import numpy as np

env = gym.make('CartPole-v1', render_mode="human")

print("State Space: ", env.observation_space)
print("Action Space: ", env.action_space)

for episode in range(5):
    state, info = env.reset(return_info=True)
    done = False

    print(f"Episode {episode + 1} - Starting state: {state}")

    while not done:
        action = env.action_space.sample()
        next_state, reward, done, info = env.step(action)

        print(f"Action: {action}, Reward: {reward}, Next State: {next_state}")

        env.render()

        if done:
            print(f"Episode {episode + 1} ended.")
            break

env.close()
```

```
Episode 4 ended.
Episode 5 - Starting state: [-0.04955993  0.00701418  0.01617307  0.04153911]
Action: 0, Reward: 1.0, Next State: [-0.04941965 -0.18833591  0.01700385  0.3392806 ]
Action: 0, Reward: 1.0, Next State: [-0.05318637 -0.38369563  0.02378946  0.6372767 ]
Action: 1, Reward: 1.0, Next State: [-0.06086028 -0.18891338  0.036535    0.35217944]
Action: 1, Reward: 1.0, Next State: [-0.06463855  0.0056705   0.04357859  0.07123729]
Action: 1, Reward: 1.0, Next State: [-0.06452513  0.20014147  0.04500333 -0.20738417]
Action: 1, Reward: 1.0, Next State: [-0.06052231  0.394592    0.04085565 -0.4855381 ]
Action: 0, Reward: 1.0, Next State: [-0.05263047  0.19891803  0.03114489 -0.18026388]
Action: 0, Reward: 1.0, Next State: [-0.04865211  0.00336456  0.02753961  0.12207919]
Action: 1, Reward: 1.0, Next State: [-0.04858482  0.19808134  0.02998119 -0.16178963]
Action: 1, Reward: 1.0, Next State: [-0.04462319  0.39276156  0.0267454  -0.44486555]
Action: 1, Reward: 1.0, Next State: [-0.03676796  0.5874951   0.01784809 -0.7289992 ]
Action: 0, Reward: 1.0, Next State: [-0.02501806  0.39213103  0.00326811 -0.4307527 ]
Action: 1, Reward: 1.0, Next State: [-0.01717544  0.58720654 -0.00534695 -0.7224036 ]
Action: 1, Reward: 1.0, Next State: [-0.00543131  0.78240204 -0.01979502 -1.0167646 ]
Action: 1, Reward: 1.0, Next State: [ 0.01021674  0.97778225 -0.04013031 -1.3155969 ]
Action: 1, Reward: 1.0, Next State: [ 0.02977238  1.1733884  -0.06644225 -1.6205649 ]
Action: 1, Reward: 1.0, Next State: [ 0.05324015  1.369227   -0.09885355 -1.9331945 ]
Action: 1, Reward: 1.0, Next State: [ 0.08062469  1.565258   -0.13751744 -2.254821  ]
Action: 0, Reward: 1.0, Next State: [ 0.11192985  1.3716697  -0.18261386 -2.0074828 ]
Action: 1, Reward: 1.0, Next State: [ 0.13936324  1.5681646  -0.22276351 -2.3507147 ]
Episode 5 ended.
```

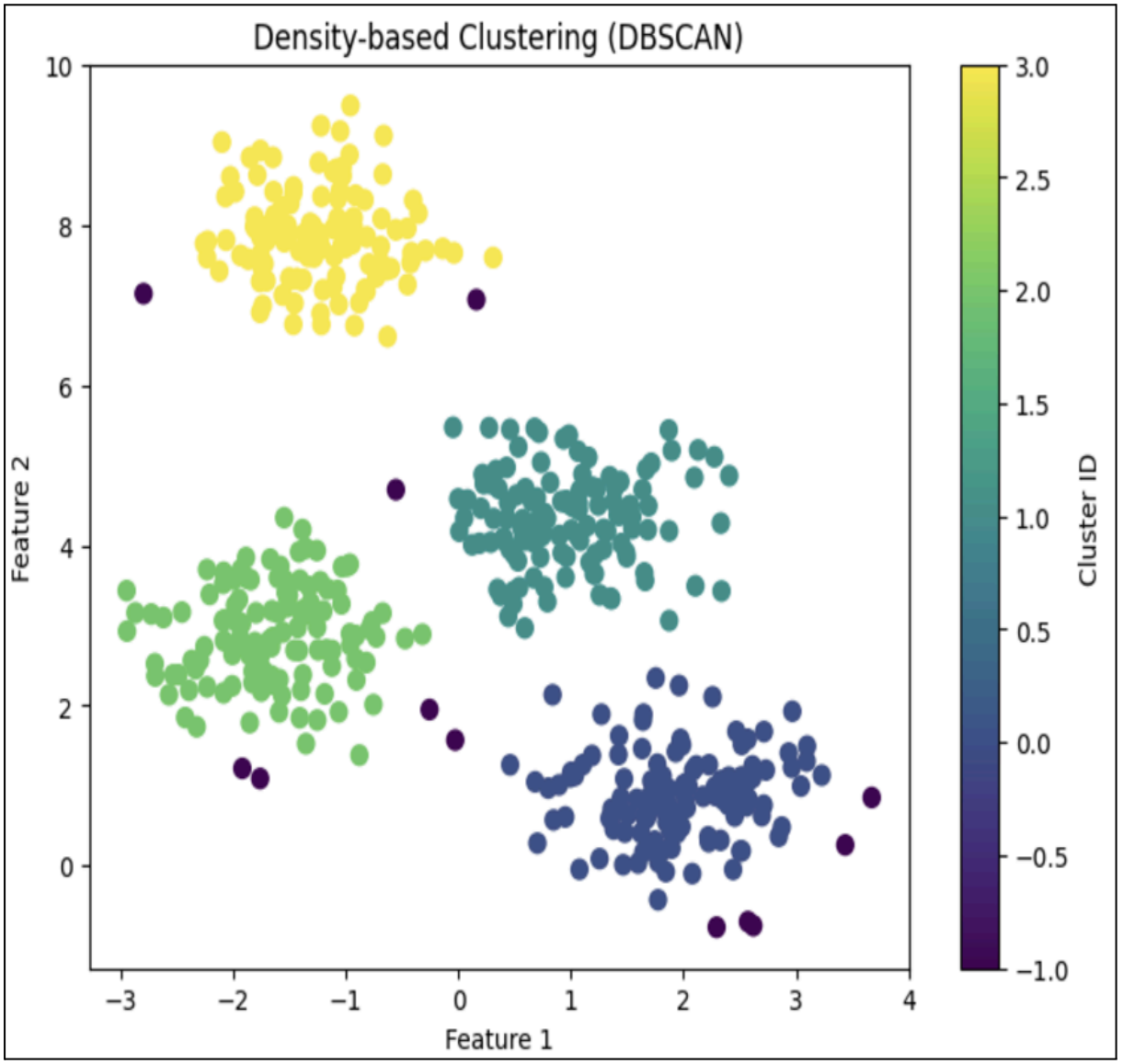
EXPERIMENT - 6

Aim: Write a program to implement the DBSCAN clustering algorithm.

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

It is an unsupervised machine learning algorithm used for clustering. It groups together data points that are closely packed based on a distance metric (usually Euclidean distance), and marks points that are in low-density regions as outliers (noise). It does not require the number of clusters to be specified beforehand, unlike algorithms such as K-Means.

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

X, _ = make_blobs(n_samples=500, centers=4, cluster_std=0.60, random_state=0)

dbscan = DBSCAN(eps=0.5, min_samples=5)
y_db = dbscan.fit_predict(X)

plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c=y_db, cmap='viridis', s=50)
plt.title("Density-based Clustering (DBSCAN)")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.colorbar(label="Cluster ID")
plt.show()
```



1. **Core Points**: Data points that have at least a specified minimum number of neighbors within a given radius (eps).

2. **Border Points**: Points that are within the eps radius of a core point but do not have enough neighbors to be considered core points themselves.

3. **Noise Points**: Points that do not belong to any cluster because they are not within the eps radius of any core point and don't meet the minimum neighbors condition.

EXPERIMENT - 7

Aim: Write a program to perform

1. Model Regularization

Model regularization refers to techniques used in machine learning and statistical modeling to prevent overfitting by penalizing excessively complex models. The goal is to strike a balance between fitting the data well and keeping the model simple and generalizable.

• L2 regularization (Ridge)

Adds a penalty equal to the square of the coefficients. This helps to shrink the coefficients, but unlike L1 regularization, it doesn't make them exactly zero.

```
import numpy as np
import pandas as pd
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Lasso, Ridge
from sklearn.metrics import accuracy_score

X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, random_state=42)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

lasso = Lasso(alpha=0.1)
lasso.fit(X_train, y_train)
lasso_predictions = lasso.predict(X_test)
lasso_accuracy = accuracy_score(y_test, (lasso_predictions > 0.5).astype(int))
print("L1 Regularization (Lasso) Accuracy: ", lasso_accuracy)
```

```
L1 Regularization (Lasso) Accuracy:  0.8066666666666666
```

• L1 regularization (Lasso)

Adds a penalty equal to the absolute value of the coefficients. This can drive some coefficients to zero, leading to sparse models where only the most important features are retained.

```
ridge = Ridge(alpha=1.0)
ridge.fit(X_train, y_train)
ridge_predictions = ridge.predict(X_test)
ridge_accuracy = accuracy_score(y_test, (ridge_predictions > 0.5).astype(int))

print("L2 Regularization (Ridge) Accuracy: ", ridge_accuracy)
```

```
L2 Regularization (Ridge) Accuracy:  0.8133333333333334
```

2. Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a statistical technique used for dimensionality reduction. It transforms the original features into a new set of orthogonal components (principal components) ordered by the variance they explain in the data. PCA is widely used for:

- Reducing the number of features while retaining most of the variance in the dataset.
- Identifying patterns in data.
- Visualizing high-dimensional data in lower dimensions (typically 2D or 3D).

• PCA+ L2 regularisation

PCA + L2 combines PCA with **L2 regularization** (Ridge). Similar to PCA + L1, the data is first reduced using PCA, but instead of applying L1 to enforce sparsity, L2 regularization is used to shrink the coefficients. The goal of PCA + L2 is to:

• Use PCA for dimensionality reduction.
• Apply L2 regularization to shrink the feature coefficients, helping to improve the model's generalizability while avoiding large weights in the transformed space.

```
[52] ridge = Ridge(alpha=1.0)
     ridge.fit(X_train_pca, y_train)
     ridge_predictions = ridge.predict(X_test_pca)
     ridge_accuracy = accuracy_score(y_test, (ridge_predictions > 0.5).astype(int))

     print("L2 Regularization (Ridge) Accuracy after PCA: ", ridge_accuracy)
```

```
L2 Regularization (Ridge) Accuracy after PCA:  0.8266666666666667
```

• PCA+ L1 regularisation

PCA + L1 combines dimensionality reduction with sparse feature selection. In this approach, the data is first transformed using PCA to reduce dimensionality, and then L1 regularization (Lasso) is applied to further promote sparsity. The idea is to:

• Reduce the data's dimensions to capture most of the variance.
• Apply L1 regularization to eliminate less important components, driving some coefficients to zero and ensuring that the resulting model is sparse.

```python
import numpy as np
import pandas as pd
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Lasso, Ridge
from sklearn.decomposition import PCA
from sklearn.metrics import accuracy_score

X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, random_state=42)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

pca = PCA(n_components=5)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)

lasso = Lasso(alpha=0.1)
lasso.fit(X_train_pca, y_train)
lasso_predictions = lasso.predict(X_test_pca)
lasso_accuracy = accuracy_score(y_test, (lasso_predictions > 0.5).astype(int))
print("L1 Regularization (Lasso) Accuracy: ", lasso_accuracy)
```

```
L1 Regularization (Lasso) Accuracy:  0.8433333333333334
```

3. Optimisation using feature selection methods

Feature selection is the process of identifying and selecting the most relevant and important features (variables) in a dataset to improve the performance of a machine learning model. It helps reduce overfitting, improve accuracy, and decrease training time by removing irrelevant, redundant, or noisy features.

Types of Feature Selection:

1. **Filter Methods**: Select features based on statistical measures (e.g., correlation, chi-square test).

2. **Wrapper Methods**: Use iterative modeling (e.g., RFE) to evaluate feature subsets.

3. **Embedded Methods**: Select features during model training using

algorithms like decision trees or Lasso regression.

```python
import numpy as np
import pandas as pd
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Lasso, Ridge
from sklearn.feature_selection import RFE
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, random_state=42)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

svc = SVC(kernel='linear')
selector = RFE(svc, n_features_to_select=5)
selector = selector.fit(X_train, y_train)
X_train_rfe = selector.transform(X_train)
X_test_rfe = selector.transform(X_test)

lasso = Lasso(alpha=0.1)
lasso.fit(X_train_rfe, y_train)
lasso_predictions = lasso.predict(X_test_rfe)
lasso_accuracy = accuracy_score(y_test, (lasso_predictions > 0.5).astype(int))

ridge = Ridge(alpha=1.0)
ridge.fit(X_train_rfe, y_train)
ridge_predictions = ridge.predict(X_test_rfe)
ridge_accuracy = accuracy_score(y_test, (ridge_predictions > 0.5).astype(int))
```

```python
print("L1 Regularization (Lasso) Accuracy after Feature Selection: ", lasso_accuracy)
print("L2 Regularization (Ridge) Accuracy after Feature Selection: ", ridge_accuracy)
```

```
L1 Regularization (Lasso) Accuracy after Feature Selection:  0.8066666666666666
L2 Regularization (Ridge) Accuracy after Feature Selection:  0.8166666666666667
```

EXPERIMENT - 8

Aim: Write a program to perform kernel based SVM model.

Kernel-based SVM is an extension of the linear SVM that enables it to perform non-linear classification by mapping data into a higher-dimensional space where it can be separated linearly. Kernels are mathematical functions used to compute the similarity (or relationship) between data points in the input space without explicitly transforming the data into higher dimensions. This approach makes SVM computationally efficient even in complex feature spaces.

1. **Hyperplane**: A decision boundary that separates data points of different classes.
2. **Support Vectors**: The data points closest to the hyperplane, which influence its position and orientation.
3. **Kernel Trick**: Avoids the explicit computation of high-dimensional mappings by using kernel functions directly to compute the similarity between points.

Common Kernel Functions:

1. **Linear Kernel**: Used when data is linearly separable.

2. **Polynomial Kernel**: Maps data into higher polynomial dimensions.

3. **Radial Basis Function (RBF) Kernel**: Used for non-linear classification.

1. Gaussian (Radial Basis Function)

Gaussian Similarity: The kernel computes the similarity between two data points based on their distance. Points closer to each other have higher similarity.

1. **Non-linear Mapping:** Effectively maps the input features into an infinite dimensional space.

2. **Gamma (γ\gamma):**
   - A hyperparameter that controls the influence of a single training example.
   - Smaller γ\gammaγ values mean the model considers points far apart as similar, leading to smoother decision boundaries.
   - Larger γ\gammaγ values result in tighter decision boundaries, which can lead to overfitting.

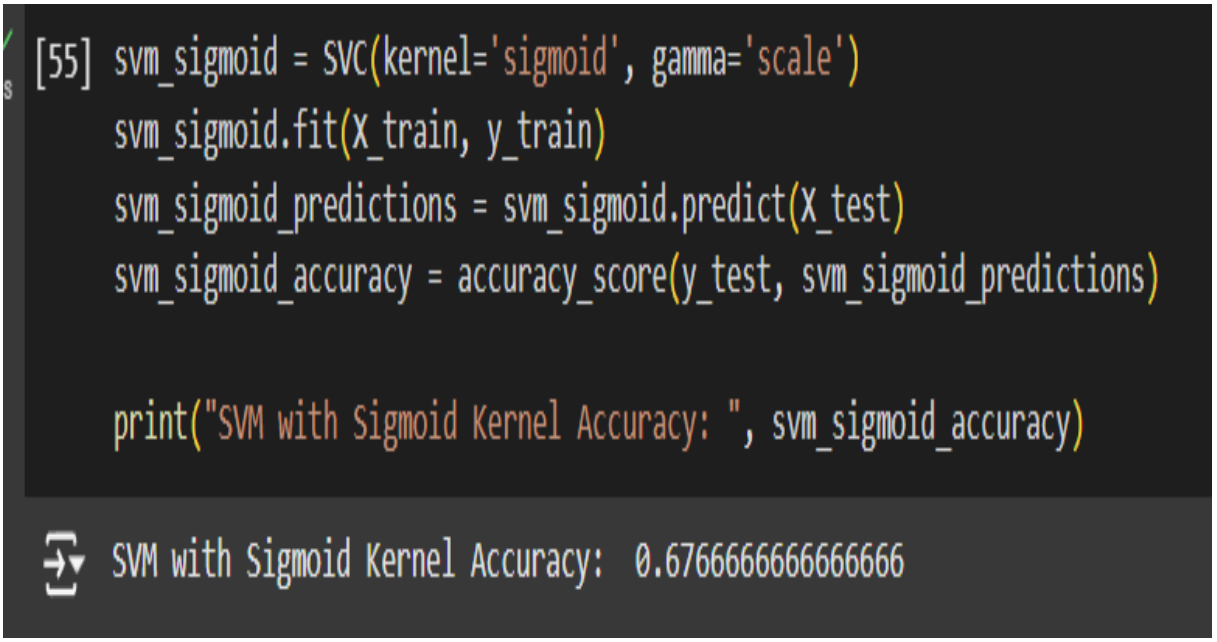
Used in: Image classification, text categorization, and bioinformatics, where data relationships are often non-linear.

```python
import numpy as np
import pandas as pd
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, random_state=42)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

svm_rbf = SVC(kernel='rbf', gamma='scale')
svm_rbf.fit(X_train, y_train)
svm_rbf_predictions = svm_rbf.predict(X_test)
svm_rbf_accuracy = accuracy_score(y_test, svm_rbf_predictions)
print("SVM with Gaussian (RBF) Kernel Accuracy: ", svm_rbf_accuracy)
```

```
SVM with Gaussian (RBF) Kernel Accuracy:  0.9066666666666666
```

2. Sigmoid kernel

- **Hyperbolic Tangent Function**: Resembles the behavior of neural networks by using a non-linear activation function.
- **Parameters**:
  - o α\alpha: Scale of the input features.
  - o c: Offset or bias term.
- **Dual Nature**: The sigmoid kernel can behave similarly to a perceptron and is sometimes equivalent to a shallow neural network.



EXPERIMENT - 9

Aim: Write a program to perform model regularisation and optimisation using ensemble methods.

1. Random Forest

Random Forest is an **ensemble learning method** primarily used for classification and regression tasks. It builds multiple decision trees during training and merges their outputs for better performance and stability.

```
random_forest = RandomForestClassifier(n_estimators=100, random_state=42)
random_forest.fit(X_train, y_train)
rf_predictions = random_forest.predict(X_test)
rf_accuracy = accuracy_score(y_test, rf_predictions)

print("Ensemble Method (Random Forest) Accuracy: ", rf_accuracy)
```

```
Ensemble Method (Random Forest) Accuracy:  0.94
```

2. Gradient Boosting

Gradient Boosting is an **ensemble method** that builds models sequentially, where each model corrects the errors of the previous ones. Unlike Random Forest, it uses the concept of boosting.

```
gradient_boosting = GradientBoostingClassifier(n_estimators=100, random_state=42)
gradient_boosting.fit(X_train, y_train)
gb_predictions = gradient_boosting.predict(X_test)
gb_accuracy = accuracy_score(y_test, gb_predictions)

print("Ensemble Method (Gradient Boosting) Accuracy: ", gb_accuracy)
```

```
Ensemble Method (Gradient Boosting) Accuracy:  0.9366666666666666
```

EXPERIMENT - 10

Aim: Write a program to perform the over-sampling and under-sampling of a considered data in machine learning.

1. SMOTE (Synthetic Minority Over-sampling Technique)

SMOTE is a method used to balance class distributions in imbalanced datasets by generating synthetic samples for the minority class. Instead of simply duplicating existing minority class examples, SMOTE creates new synthetic examples by interpolating between existing samples.

```
import numpy as np
import pandas as pd
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from imblearn.over_sampling import SMOTE, RandomOverSampler
from imblearn.under_sampling import RandomUnderSampler
from imblearn.combine import SMOTEENN
from collections import Counter

X, y = make_classification(n_samples=1000, n_features=10, n_informative=5,
                           n_redundant=5, n_classes=2, weights=[0.9, 0.1],
                           random_state=42)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)
print("Class distribution after SMOTE (Over-sampling): ", Counter(y_train_smote))
```

```
Class distribution after SMOTE (Over-sampling):  Counter({0: 631, 1: 631})
```

2. Random Sampler (RandomUnderSampler and RandomOverSampler)

- Reduces the number of samples in the majority class by randomly selecting a subset.
- Achieves balance by decreasing the size of the majority class.

```
[63]
ros = RandomOverSampler(random_state=42)
X_train_ros, y_train_ros = ros.fit_resample(X_train, y_train)
print("Class distribution after Random Over-sampling: ", Counter(y_train_ros))

Class distribution after Random Over-sampling:  Counter({0: 631, 1: 631})
```

```
[64]
rus = RandomUnderSampler(random_state=42)
X_train_rus, y_train_rus = rus.fit_resample(X_train, y_train)
print("Class distribution after Random Under-sampling: ", Counter(y_train_rus))

Class distribution after Random Under-sampling:  Counter({0: 69, 1: 69})
```

3. SMOTEENN (Combination of SMOTE and Edited Nearest Neighbors)

SMOTEENN combines **SMOTE** for over-sampling and **Edited Nearest Neighbors (ENN)** for cleaning the dataset by removing ambiguous or noisy samples from the majority class.

```
smoteenn = SMOTEENN(random_state=42)
X_train_smoteenn, y_train_smoteenn = smoteenn.fit_resample(X_train, y_train)
print("Class distribution after SMOTE + ENN (SMOTEENN): ", Counter(y_train_smoteenn))

Class distribution after SMOTE + ENN (SMOTEENN):  Counter({1: 609, 0: 583})
```