In [3]:
```
# import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

Everything About Machine Learning using scikit learn library

1. What is Machine learning

Scikit-learn, often abbreviated as sklearn, is an open-source machine learning library for the Python programming language. It provides a wide range of tools for data mining and data analysis, built on top of the NumPy, SciPy, and Matplotlib libraries. Scikit-learn is designed to be simple and efficient for tasks such as:

- **Preprocessing**: Feature extraction and normalization (e.g., scaling data).
- **Model Selection**: Comparing, validating, and choosing parameters and models (e.g., grid search).

--

- **Classification**: Identifying which category an object belongs to (e.g., spam detection, image recognition).
- **Regression**: Predicting a continuous-valued attribute associated with an object (e.g., predicting house prices).
- **Clustering**: Automatic grouping of similar objects into sets (e.g., customer segmentation).
- **Dimensionality Reduction**: Reducing the number of random variables under consideration (e.g., PCA for visualization).

1.1 Preprocessing

Preprocessing in scikit-learn (sklearn) refers to a set of techniques used to transform and prepare raw data before feeding it into a machine learning model. These transformations help improve the performance and accuracy of the model by ensuring the data is in a suitable format and scale. Scikit-learn offers a variety of preprocessing tools, including:

1. **Scaling and Normalization**

Standardization (StandardScaler): Scales features to have zero mean and unit variance. This is important when features have different units or scales. Min-Max Scaling (MinMaxScaler): Scales features to a specific range, typically [0, 1]. Normalization (Normalizer): Scales individual samples to have unit norm (useful for sparse data or data with varying lengths).

1. **Encoding Categorical Variables**

Label Encoding (LabelEncoder): Converts categorical labels into numerical values (0, 1, 2, ...). One-Hot Encoding (OneHotEncoder): Converts categorical features into a binary matrix, creating a new column for each category.

1. **Handling Missing Values**

Imputation (SimpleImputer): Replaces missing values with a specific value (e.g., mean, median) or strategy (e.g., most frequent).

1. **Binarization**

Binarizer: Converts numerical features into binary values based on a threshold.

1. **Polynomial Features**

PolynomialFeatures: Generates polynomial and interaction features, allowing the model to capture non-linear relationships.

1. **Discretization**

KBinsDiscretizer: Converts continuous data into discrete bins.

1. **Feature Selection**

SelectKBest, RFE, etc.: Selects the most important features based on statistical tests or model performance.

1. **Dimensionality Reduction**

Principal Component Analysis (PCA): Reduces the number of features while preserving as much information as possible.

In [4]:
```
from sklearn.datasets import make_classification
from sklearn.preprocessing import StandardScaler, MinMaxScaler, LabelEncoder, OneHo
from sklearn.impute import SimpleImputer

# Generate synthetic data
X, y = make_classification(n_samples=100, n_features=4, n_informative=2, n_redundar
categories = np.random.choice(['cat', 'dog', 'mouse'], size=100)

# Convert to DataFrame for easier manipulation
df = pd.DataFrame(X, columns=['Feature1', 'Feature2', 'Feature3', 'Feature4'])
df['Category'] = categories
df['Target'] = y

# Introduce some missing values
df.iloc[5:10, 0] = np.nan


df
```

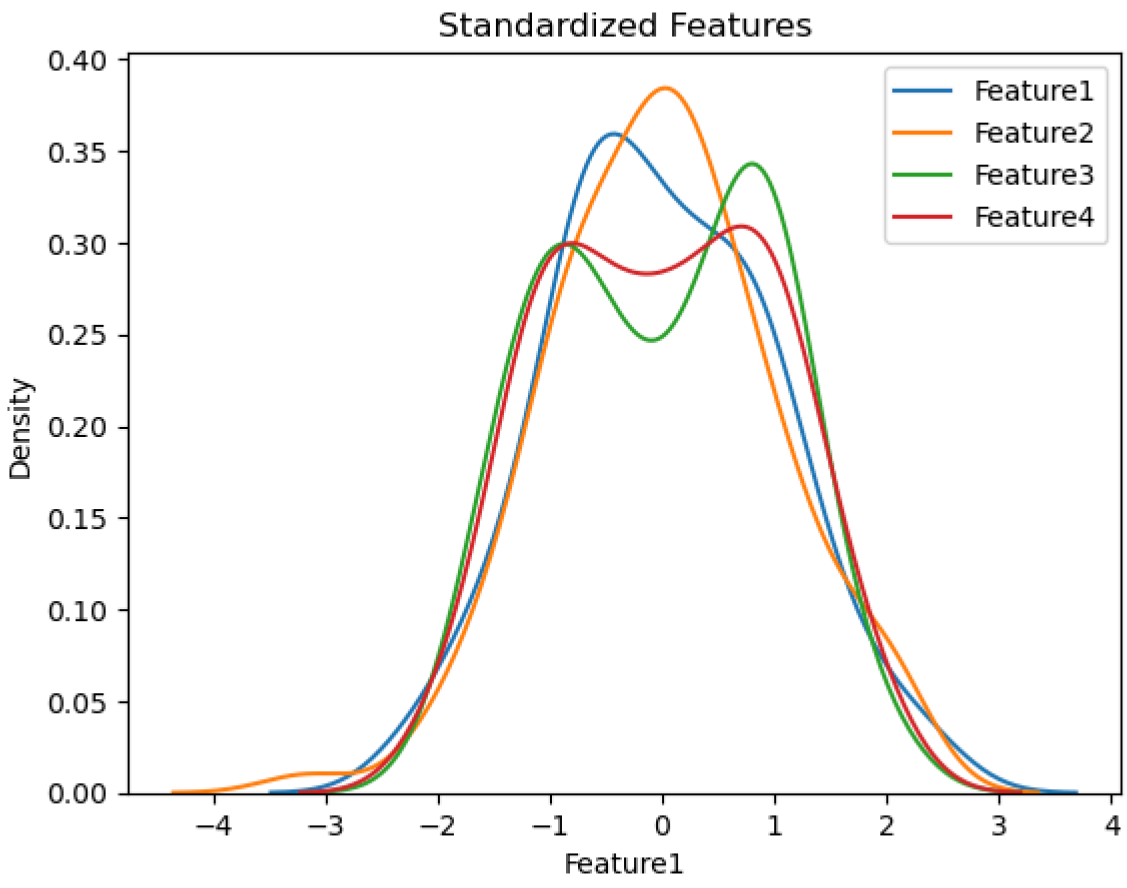
Out[4]:

	Feature1	Feature2	Feature3	Feature4	Category	Target
0	-0.334501	-1.200296	-0.539630	-0.724280	cat	0
1	-0.202193	0.547097	0.254157	-1.795320	cat	1
2	1.277677	-1.534114	0.341294	0.573042	dog	1
3	-0.600217	0.199060	0.714794	-1.429220	mouse	1
4	-1.077745	0.064280	-1.658304	-1.571273	mouse	0
...	...	...	...	...	...	...
95	-1.407464	-1.448084	-1.876538	0.230859	cat	0
96	-0.651600	0.045572	-2.009185	-2.241820	cat	0
97	-0.213447	-0.718444	-1.755186	0.360170	mouse	0
98	-1.024388	-3.241267	-1.548510	-1.435349	mouse	0
99	-0.599393	-0.839722	2.368674	2.256612	mouse	1

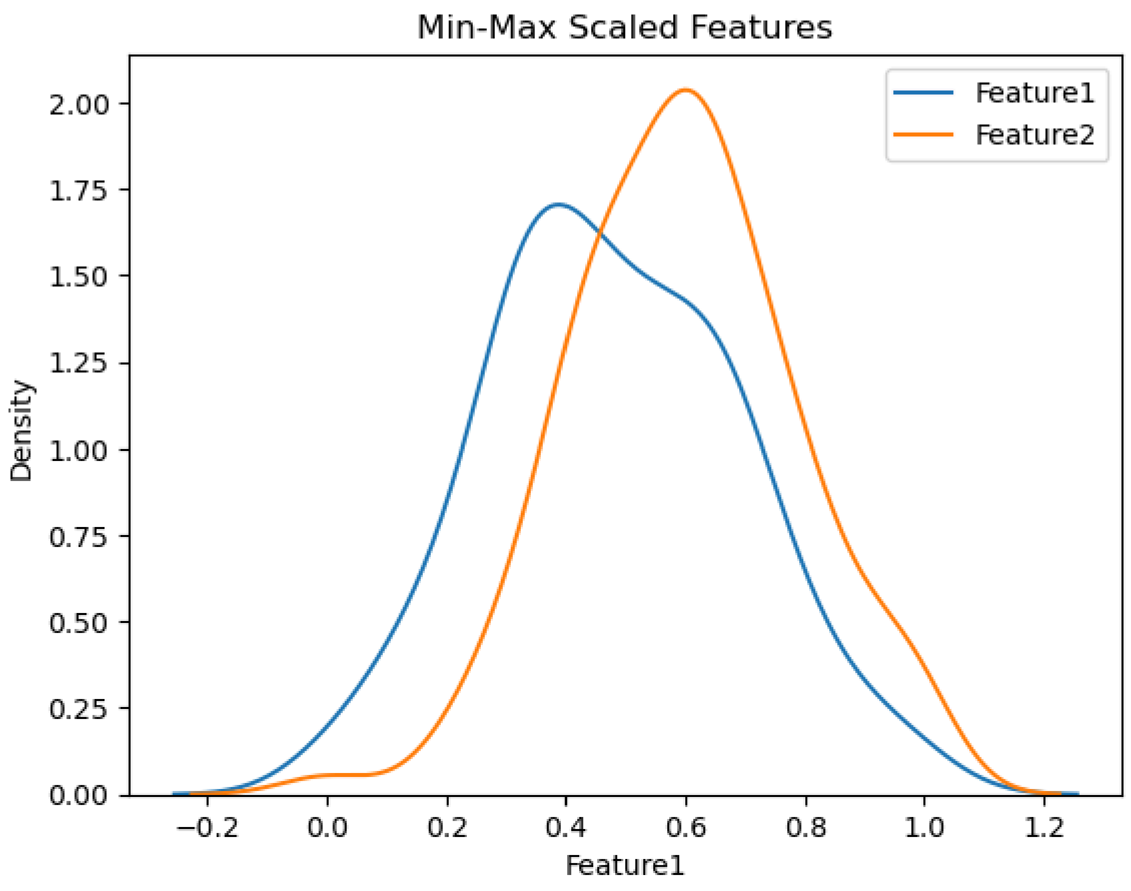
100 rows × 6 columns

In [7]:

```
scaler = StandardScaler()
df[['Feature1', 'Feature2', 'Feature3', 'Feature4']] = scaler.fit_transform(df[['Fe

# Visualization
sns.kdeplot(df['Feature1'], label='Feature1')
sns.kdeplot(df['Feature2'], label='Feature2')
sns.kdeplot(df['Feature3'], label='Feature3')
sns.kdeplot(df['Feature4'], label='Feature4')


plt.title('Standardized Features')
plt.legend()
plt.show()
```



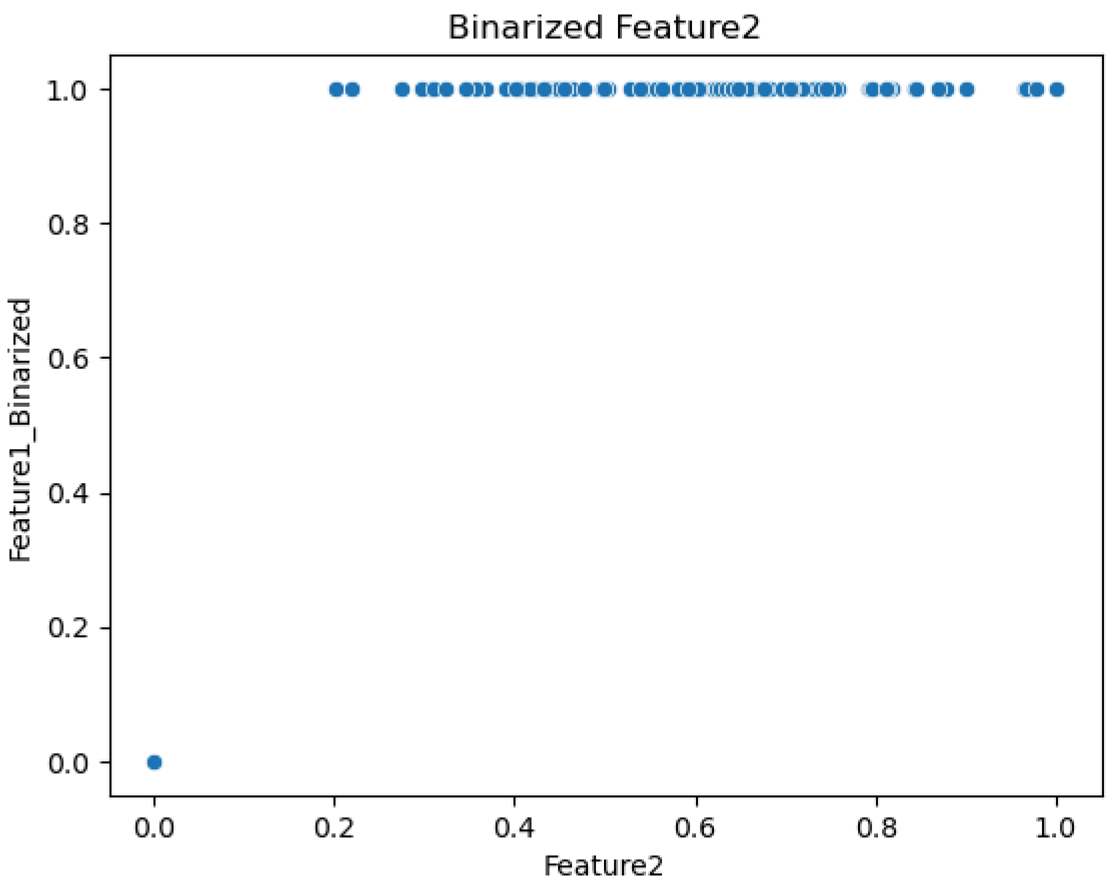
In [8]:
```
min_max_scaler = MinMaxScaler()
df[['Feature1', 'Feature2', 'Feature3', 'Feature4']] = min_max_scaler.fit_transforn

# Visualization
sns.kdeplot(df['Feature1'], label='Feature1')
sns.kdeplot(df['Feature2'], label='Feature2')
plt.title('Min-Max Scaled Features')
plt.legend()
plt.show()
```



In [11]:
```
label_encoder = LabelEncoder()
df['Category_LabelEncoded'] = label_encoder.fit_transform(df['Category'])

# Example: 'cat' -> 0, 'dog' -> 1, 'mouse' -> 2
print(df[['Category', 'Category_LabelEncoded']].head(10))
```

```
  Category  Category_LabelEncoded
0      cat                      0
1      cat                      0
2      dog                      1
3    mouse                      2
4    mouse                      2
5      dog                      1
6      cat                      0
7    mouse                      2
8    mouse                      2
9      dog                      1
```

In [15]:
```
binarizer = Binarizer(threshold=0.0)
df['Feature1_Binarized'] = binarizer.fit_transform(df[['Feature2']])

# Visualization
sns.scatterplot(data=df, x='Feature2', y='Feature1_Binarized')
plt.title('Binarized Feature2')
plt.show()
```



In [17]:
```
poly = PolynomialFeatures(degree=2, include_bias=False)
poly_features = poly.fit_transform(df[['Feature2', 'Feature3']])

# Convert to DataFrame for easier visualization
poly_df = pd.DataFrame(poly_features, columns=poly.get_feature_names_out(['Feature2

# Display the Polynomial Features
print(poly_df.head())
```

```
   Feature2  Feature3  Feature2^2  Feature2 Feature3  Feature3^2
0  0.367350  0.395252    0.134946           0.145196    0.156224
1  0.681860  0.548577    0.464933           0.374053    0.300937
2  0.307267  0.565408    0.094413           0.173731    0.319687
3  0.619218  0.637552    0.383430           0.394784    0.406473
4  0.594959  0.179173    0.353976           0.106601    0.032103
```

In [13]:
```
imputer = SimpleImputer(strategy='mean')
df['Feature1_Imputed'] = imputer.fit_transform(df[['Feature1']])

# Display the imputed data
print(df[['Feature1', 'Feature1_Imputed']].head(10))
```

```
   Feature1  Feature1_Imputed
0  0.399950          0.399950
1  0.432663          0.432663
2  0.798562          0.798562
3  0.334251          0.334251
4  0.216182          0.216182
5       NaN          0.478569
6       NaN          0.478569
7       NaN          0.478569
8       NaN          0.478569
9       NaN          0.478569
```

1.2 Model Selection

Model selection is the process of choosing the most appropriate machine learning model for a given dataset and problem. This involves evaluating different models to determine which one performs the best according to specific criteria or metrics. The goal is to find the model that generalizes well to new, unseen data, rather than one that merely fits the training data perfectly.

1. **Overfitting vs. Underfitting**:

Overfitting: When a model learns the training data too well, including noise and outliers, leading to poor performance on new data. Underfitting: When a model is too simple to capture the underlying patterns in the data, resulting in poor performance on both training and test data. Model selection aims to find a balance between overfitting and underfitting, leading to better generalization.

1. **Cross-Validation:**

Cross-validation is a technique used to assess the generalization ability of a model. The most common method is k-fold cross-validation, where the dataset is split into k subsets (folds). The model is trained on k-1 folds and validated on the remaining fold. This process is repeated k times, and the results are averaged to estimate the model's performance. Leave-One-Out Cross-Validation (LOOCV) is a special case where k equals the number of data points, so each data point is used once as the validation set.

1. **Hyperparameter Tuning:**

Machine learning models often have hyperparameters (parameters not learned from the data) that need to be set before training. Examples include the learning rate in gradient boosting, the number of neighbors in k-nearest neighbors, or the regularization strength in linear models. Grid Search: A systematic approach to tuning hyperparameters by evaluating a predefined set of hyperparameter values. Random Search: A more efficient method that randomly samples hyperparameter values, potentially covering the search space more broadly. Bayesian Optimization: A probabilistic model that selects the next set of hyperparameters based on past evaluations, aiming to find the best hyperparameters with fewer iterations.

1. **Performance Metrics:**

The choice of performance metric is crucial in model selection. Common metrics include accuracy, precision, recall, F1 score, ROC-AUC, mean squared error (MSE), and R-squared. Different metrics might be more appropriate depending on the problem type (e.g., classification vs. regression) and the specific goals of the model (e.g., high precision vs. high recall).

1. **Model Comparison:**

After cross-validation and hyperparameter tuning, models are compared based on their performance metrics. The model with the best performance metric on the validation set is usually selected. Ensemble Methods: Sometimes, instead of selecting a single model, multiple models are combined (e.g., via bagging, boosting, or stacking) to improve performance.

1. **Train-Validation-Test Split:**

The dataset is often split into three parts: training set (to train the model), validation set (to tune hyperparameters and compare models), and test set (to evaluate final performance). This helps in assessing how well the model generalizes to completely unseen data.

In [19]:
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score

# Load a sample dataset (Iris dataset for example)
data = load_iris()
X = data.data  # Features
y = data.target  # Target variable (labels)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_sta


models = {
    'Random Forest': RandomForestClassifier(),
    'Support Vector Machine': SVC()
}

for name, model in models.items():
    scores = cross_val_score(model, X_train, y_train, cv=5, scoring='accuracy')
    print(f'{name} Accuracy: {scores.mean():.3f} ± {scores.std():.3f}')
```

```
Random Forest Accuracy: 0.943 ± 0.036
Support Vector Machine Accuracy: 0.952 ± 0.043
```

In [20]:
```
from sklearn.model_selection import GridSearchCV

param_grid = {'n_estimators': [50, 100, 200],
              'max_depth': [None, 10, 20, 30]}
grid_search = GridSearchCV(RandomForestClassifier(), param_grid, cv=5, scoring='acc
grid_search.fit(X_train, y_train)
print(f'Best Parameters: {grid_search.best_params_}')
print(f'Best Cross-Validation Accuracy: {grid_search.best_score_:.3f}')
```

```
Best Parameters: {'max_depth': None, 'n_estimators': 50}
Best Cross-Validation Accuracy: 0.943
```

Performance Metrics

classification

In [22]:
```
from sklearn.metrics import accuracy_score

y_true = [0, 1, 1, 0, 1]
y_pred = [0, 1, 0, 0, 1]
accuracy = accuracy_score(y_true, y_pred)
print(f'Accuracy: {accuracy:.2f}')
```

```
Accuracy: 0.80
```

In [23]:
```
from sklearn.metrics import precision_score

precision = precision_score(y_true, y_pred)
print(f'Precision: {precision:.2f}')
```

```
Precision: 1.00
```

In [24]:
```
from sklearn.metrics import recall_score

recall = recall_score(y_true, y_pred)
print(f'Recall: {recall:.2f}')
```

```
Recall: 0.67
```

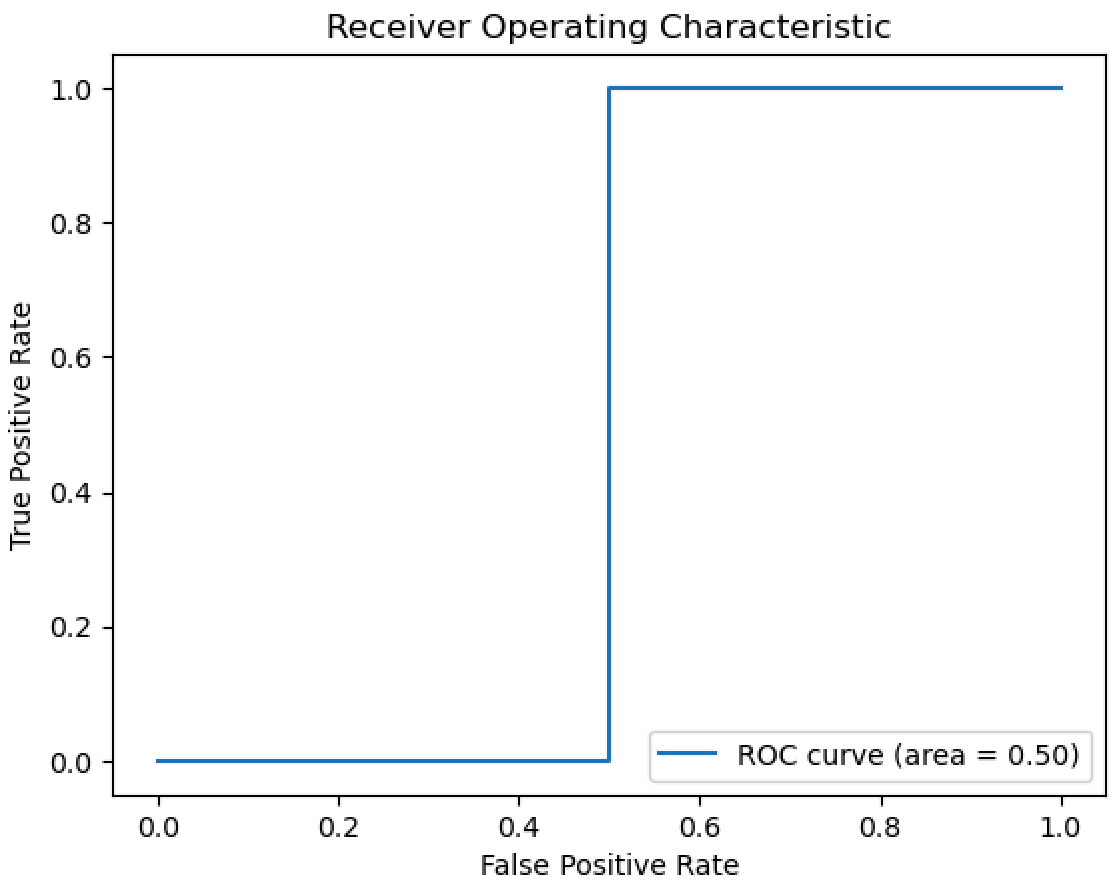
In [25]:
```
from sklearn.metrics import f1_score

f1 = f1_score(y_true, y_pred)
print(f'F1-Score: {f1:.2f}')
```

```
F1-Score: 0.80
```

In [26]:
```
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_true, y_pred)
print('Confusion Matrix:')
print(cm)
```

```
Confusion Matrix:
[[2 0]
 [1 2]]
```

In [30]:
```
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

y_scores = [0.1, 0.4, 0.35, 0.8,0.3]
fpr, tpr, thresholds = roc_curve(y_true, y_scores)
roc_auc = auc(fpr, tpr)

plt.plot(fpr, tpr, label=f'ROC curve (area = {roc_auc:.2f})')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc='lower right')
plt.show()
```



Regression Metrics

In [31]:
```
from sklearn.metrics import mean_absolute_error

y_true = [3.0, -0.5, 2.0, 7.0]
y_pred = [2.5, 0.0, 2.1, 7.8]
mae = mean_absolute_error(y_true, y_pred)
print(f'MAE: {mae:.2f}')
```

```
MAE: 0.47
```

In [33]:
```
from sklearn.metrics import mean_squared_error

mse = mean_squared_error(y_true, y_pred)
print(f'MSE: {mse:.2f}')
```

```
MSE: 0.29
```

In [34]:
```
rmse = np.sqrt(mse)
print(f'RMSE: {rmse:.2f}')
```

```
RMSE: 0.54
```

In [35]:
```
from sklearn.metrics import r2_score

r2 = r2_score(y_true, y_pred)
print(f'R-squared: {r2:.2f}')
```

```
R-squared: 0.96
```

culstering Metrics

In [36]:
```
from sklearn.metrics import silhouette_score

labels = [0, 1, 1, 0, 1]
```

```
X = [[1, 2], [1, 4], [1, 0], [10, 2], [10, 4]]
score = silhouette_score(X, labels)
print(f'Silhouette Score: {score:.2f}')
```

```
Silhouette Score: -0.30
```

In [37]:
```
from sklearn.metrics import davies_bouldin_score

dbi = davies_bouldin_score(X, labels)
print(f'Davies-Bouldin Index: {dbi:.2f}')
```

```
Davies-Bouldin Index: 5.47
```

1.3 Classification

Classification in machine learning refers to the task of predicting the categorical label of new observations based on training data. In classification problems, the output variable is a category or class, such as "spam" or "not spam," "malignant" or "benign," etc. Here's a breakdown of the key concepts and steps involved in classification:

1. Key Concepts

Classes/Labels:

The distinct categories or outcomes that the model predicts. For example, in a binary classification problem, there are two classes (e.g., "positive" and "negative"). In multi-class classification, there are more than two classes.

Features/Attributes:

The input variables or characteristics used to make predictions. Features are typically represented as a vector of numerical or categorical values.

Training Data:

A set of labeled examples used to train the model. Each example consists of features and the corresponding class label.

Test Data:

A set of examples used to evaluate the performance of the trained model. It also consists of features and class labels (in supervised learning).

Prediction:

The process of using a trained model to assign a class label to new, unseen data based on the input features.

2. Common Classification Models

Here are some commonly used classification models, their brief descriptions, and use cases:

- **Logistic Regression**

Description: Models the probability of a binary outcome using a logistic function. Use Case: Binary classification problems. Library: sklearn.linear_model.LogisticRegression

- **k-Nearest Neighbors (k-NN)**

Description: Classifies instances based on the majority class among its k-nearest neighbors. Use Case: Classification based on similarity. Library: sklearn.neighbors.KNeighborsClassifier

- **Support Vector Machine (SVM)**

Description: Finds the hyperplane that best separates different classes in the feature space. Use Case: Binary and multi-class classification. Library: sklearn.svm.SVC

- **Decision Tree**

Description: Uses a tree-like model of decisions to classify data based on feature tests. Use Case: Classification with interpretability. Library: sklearn.tree.DecisionTreeClassifier

- **Random Forest**

Description: An ensemble of decision trees that improves accuracy and reduces overfitting. Use Case: Robust classification. Library: sklearn.ensemble.RandomForestClassifier

- **Gradient Boosting Machines (GBM)**

Description: Builds models sequentially to correct errors made by previous models. Use Case: High-performance classification. Library: sklearn.ensemble.GradientBoostingClassifier

In [ ]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier, Ac
from sklearn.naive_bayes import GaussianNB, MultinomialNB, BernoulliNB
from sklearn.neural_network import MLPClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis, QuadraticDisc


models = {
    "Logistic Regression": LogisticRegression(),
    "k-Nearest Neighbors (k-NN)": KNeighborsClassifier(),
    "Support Vector Machine (SVM)": SVC(),
    "Decision Tree": DecisionTreeClassifier(),
    "Random Forest": RandomForestClassifier(),
    "Gradient Boosting Machines (GBM)": GradientBoostingClassifier(),
    "AdaBoost (Adaptive Boosting)": AdaBoostClassifier(),
    "Gaussian Naive Bayes": GaussianNB(),
    "Multinomial Naive Bayes": MultinomialNB(),
    "Bernoulli Naive Bayes": BernoulliNB(),
    "Neural Networks": MLPClassifier(),
    "Linear Discriminant Analysis (LDA)": LinearDiscriminantAnalysis(),
    "Quadratic Discriminant Analysis (QDA)": QuadraticDiscriminantAnalysis(),
}



# Evaluate each model
results = {}
for name, model in models.items():
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    # Calculate performance metrics
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='weighted')
    recall = recall_score(y_test, y_pred, average='weighted')
    f1 = f1_score(y_test, y_pred, average='weighted')

    results[name] = {
        "Accuracy": accuracy,
        "Precision": precision,
        "Recall": recall,
        "F1 Score": f1
    }

# Visualization
model_names = list(results.keys())
accuracy_scores = [results[model]['Accuracy'] for model in model_names]
f1_scores = [results[model]['F1 Score'] for model in model_names]

# Plot Accuracy
plt.figure(figsize=(14, 6))

plt.subplot(1, 2, 1)
bars_accuracy = plt.bar(model_names, accuracy_scores, color='skyblue', edgecolor='k
plt.xticks(rotation=45, ha='right')
plt.ylabel('Accuracy')
plt.title('Accuracy for Classification Models')

# Highlight the best model for Accuracy
best_accuracy_idx = np.argmax(accuracy_scores)
bars_accuracy[best_accuracy_idx].set_color('orange')
plt.text(best_accuracy_idx, accuracy_scores[best_accuracy_idx] + 0.02, 'Best Model'

# Plot F1 Score
plt.subplot(1, 2, 2)
bars_f1 = plt.bar(model_names, f1_scores, color='lightgreen', edgecolor='k')
plt.xticks(rotation=45, ha='right')
plt.ylabel('F1 Score')
plt.title('F1 Score for Classification Models')

# Highlight the best model for F1 Score
best_f1_idx = np.argmax(f1_scores)
bars_f1[best_f1_idx].set_color('orange')
plt.text(best_f1_idx, f1_scores[best_f1_idx] + 0.02, 'Best Model', ha='center', col

# Show plots
plt.tight_layout()
plt.show()
```

1.4 Regression

Regression in machine learning involves predicting a continuous output variable based on one or more input features. Unlike classification, where the goal is to assign inputs to predefined categories, regression aims to estimate a numerical value.

Types of Regression Models

- **Linear Regression**

Description: Models the relationship between a dependent variable and one or more independent variables by fitting a linear equation to the observed data. Library:

sklearn.linear_model.LinearRegression

- **Ridge Regression**

Description: A type of linear regression that includes a regularization term to prevent overfitting by penalizing large coefficients.

Library: sklearn.linear_model.Ridge

- **Lasso Regression**

Description: Similar to Ridge Regression, but uses L1 regularization, which can shrink some coefficients to zero, effectively performing feature selection.

Library: sklearn.linear_model.Lasso

- **Polynomial Regression**

Description: Extends linear regression by fitting a polynomial equation to the data, which allows for modeling more complex relationships. Library: sklearn.preprocessing.PolynomialFeatures (used to create polynomial features) Note: The model itself is still linear in terms of coefficients.

- **Support Vector Regression (SVR)**

Description: Uses Support Vector Machines to perform regression tasks. It tries to fit the best line within a margin of tolerance. Library: sklearn.svm.SVR

- **Decision Tree Regressor**

Description: Models the target variable as a decision tree where splits are made to reduce variance in the target variable. Library: sklearn.tree.DecisionTreeRegressor

- **Random Forest Regressor**

Description: An ensemble method that uses multiple decision trees to improve performance and control overfitting. Library: sklearn.ensemble.RandomForestRegressor

- **Gradient Boosting Regressor**

Description: An ensemble technique that builds models sequentially, with each new model correcting errors made by previous models. Library: sklearn.ensemble.GradientBoostingRegressor

In [38]:
```
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.preprocessing import PolynomialFeatures
from sklearn.svm import SVR
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor, AdaE
from sklearn.neighbors import KNeighborsRegressor

from sklearn.metrics import mean_squared_error, r2_score


models = {
    "Linear Regression": LinearRegression(),
    "Ridge Regression": Ridge(),
    "Lasso Regression": Lasso(),
    "Polynomial Regression": PolynomialFeatures(),  # Note: PolynomialFeatures is r
    "Support Vector Regression (SVR)": SVR(),
    "Decision Tree Regressor": DecisionTreeRegressor(),
    "Random Forest Regressor": RandomForestRegressor(),
    "Gradient Boosting Regressor": GradientBoostingRegressor(),
    "AdaBoost Regressor": AdaBoostRegressor(),
    "k-Nearest Neighbors Regressor (k-NN)": KNeighborsRegressor(),
}

results = {}
for name, model in models.items():
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)
    results[name] = {
        "Mean Squared Error": mse,
        "R^2 Score": r2
    }


# Extract model names
models = list(results.keys())

# Extract MSE and R^2 Scores
mse_values = [results[model]['Mean Squared Error'] for model in models]
r2_values = [results[model]['R^2 Score'] for model in models]

# Find the best models
best_mse_idx = np.argmin(mse_values)
best_r2_idx = np.argmax(r2_values)

# Plot MSE
plt.figure(figsize=(14, 6))

# Bar plot for Mean Squared Error
plt.subplot(1, 2, 1)
bars = plt.bar(models, mse_values, color='skyblue', edgecolor='k')
plt.xticks(rotation=45, ha='right')
plt.ylabel('Mean Squared Error')
plt.title('Mean Squared Error for Regression Models')

# Highlight the best model for MSE
bars[best_mse_idx].set_color('orange')
plt.text(best_mse_idx, mse_values[best_mse_idx] + 500, 'Best Model', ha='center', c

# Plot R^2 Score
plt.subplot(1, 2, 2)
bars_r2 = plt.bar(models, r2_values, color='lightgreen', edgecolor='k')
plt.xticks(rotation=45, ha='right')
plt.ylabel('R^2 Score')
plt.title('R^2 Score for Regression Models')

# Highlight the best model for R^2 Score
bars_r2[best_r2_idx].set_color('orange')
plt.text(best_r2_idx, r2_values[best_r2_idx] + 0.05, 'Best Model', ha='center', col

# Show plots
plt.tight_layout()
plt.show()
```

1.5 Culstering

Clustering in machine learning is an unsupervised learning technique used to group similar data points together based on their features. Unlike supervised learning, clustering does not use labeled data; instead, it discovers patterns or groupings in the data based on similarity.

In [39]:
```
from sklearn.cluster import KMeans, DBSCAN, AgglomerativeClustering, MeanShift, Spe
from sklearn.mixture import GaussianMixture
from sklearn.preprocessing import StandardScaler

clustering_models = {
    "KMeans": KMeans(),
    "DBSCAN": DBSCAN(),
    "Agglomerative Clustering": AgglomerativeClustering(),
    "Mean Shift": MeanShift(),
    "Spectral Clustering": SpectralClustering(),
    "Gaussian Mixture Model (GMM)": GaussianMixture(),
    "Standard Scaler": StandardScaler()  # Note: StandardScaler is not a clustering
}
```

1.6 Dimensionality reduction

Dimensionality reduction is a technique used in machine learning and data analysis to reduce the number of features or variables in a dataset while retaining as much of the original information as possible. This is useful for simplifying models, improving performance, and visualizing high-dimensional data.

In [40]:
```
from sklearn.decomposition import PCA, NMF, FastICA, TruncatedSVD
from sklearn.manifold import TSNE, Isomap
from sklearn.preprocessing import StandardScaler

dimensionality_reduction_models = {
    "Principal Component Analysis (PCA)": PCA(),
    "Non-Negative Matrix Factorization (NMF)": NMF(),
    "Fast Independent Component Analysis (FastICA)": FastICA(),
    "Truncated Singular Value Decomposition (TruncatedSVD)": TruncatedSVD(),
    "t-Distributed Stochastic Neighbor Embedding (t-SNE)": TSNE(),
    "Isomap": Isomap()
}
```

In [ ]: