```
# none data types: something empty blank insert some data
a=None
print(a,type(a))

# def,oops,dataframe,visulization
```

```
None <class 'NoneType'>
```

classification types of algorithms:

1)logistic regression

2)Decision Tree Classification

3)Rondom forest

4)k-nearest neighbors

5)Navive Bayes

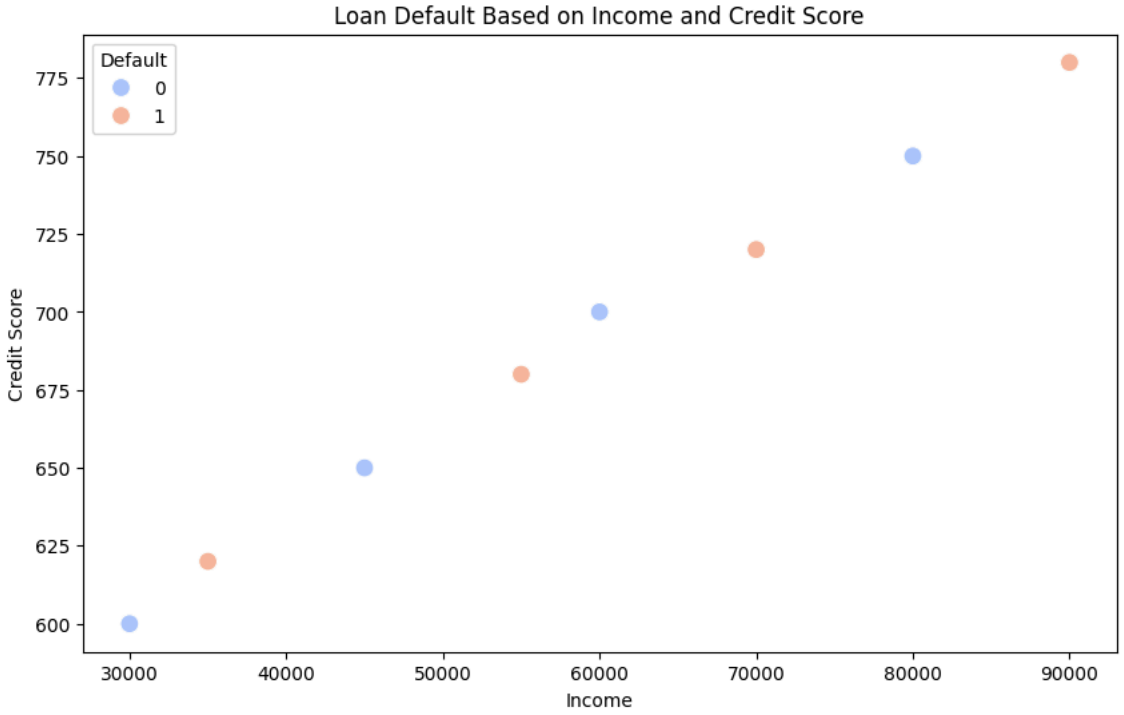
6)support vector machines(svm)

1)classfication algorithm:1)logistic regression Logistic regression is a statistical method used to model the probability of a binary outcome (two possible results, like yes/no, true/false) based on one or more predictor variables. It works by fitting a sigmoid function to the data, which maps input values to a probability between 0 and 1

```
# logistic regression:
# Simple Logistic Regression Example
# Predict if a person will default on a loan (0 = No Default, 1 = Default) based on:
# Income
# Credit Score

# Step 1: Import libraries:
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
import seaborn as sns

# Step 2: Create sample data:
data = {
    'Income': [30000, 45000, 60000, 80000, 35000, 55000, 70000, 90000],
    'Credit_Score': [600, 650, 700, 750, 620, 680, 720, 780],
    'Default': [0, 0, 0, 0, 1, 1, 1, 1] # 0: No Default, 1: Default
}
df_loan = pd.DataFrame(data)
print(df_loan)

# Step 3: Define features and target:
X_loan = df_loan[['Income', 'Credit_Score']] # input
y_loan = df_loan['Default']# output

# Step 4: Train-test split:
X_train_loan, X_test_loan, y_train_loan, y_test_loan = train_test_split(X_loan, y_loan, test_size=0.2

# Step 5: Train the classifier
model_loan = LogisticRegression()
model_loan.fit(X_train_loan, y_train_loan) ## For model training, we should use .fit function
print(f"Training samples: {X_train_loan.shape[0]}, Test samples: {X_test_loan.shape[0]}")
print('Loan Default model is trained successfully')

# Step 6: Predict on test data:
y_pred_loan = model_loan.predict(X_test_loan) # inferencing
print("Predictions on test set:", y_pred_loan)

# Step 7: Evaluate metric:
```

```
# Step 7: Evaluate metric:
accuracy_loan = accuracy_score(y_test_loan, y_pred_loan)
print("Accuracy:", accuracy_loan)

# Step 8: Visualization (optional, for simple 2D data):
plt.figure(figsize=(10, 6))
sns.scatterplot(x='Income', y='Credit_Score', hue='Default', data=df_loan, palette='coolwarm', s=100)
plt.xlabel('Income')
plt.ylabel('Credit Score')
plt.title('Loan Default Based on Income and Credit Score')
```

```
   Income  Credit_Score  Default
0   30000           600        0
1   45000           650        0
2   60000           700        0
3   80000           750        0
4   35000           620        1
5   55000           680        1
6   70000           720        1
7   90000           780        1
Training samples: 6, Test samples: 2
Loan Default model is trained successfully
Predictions on test set: [0 0]
Accuracy: 0.5
Text(0.5, 1.0, 'Loan Default Based on Income and Credit Score')
```



Decision Tree Classification : A Decision Tree Classifier is a supervised machine learning algorithm that uses a tree-like structure to classify instances based on their feature values. It's essentially a flowchart-like structure where each internal node represents a feature (or attribute), the branches represent decision rules based on that feature, and the leaf nodes represent the final classification.

Key Components: Root Node: The starting point of the tree, representing the most important feature for making the initial split. Internal Nodes: Nodes that represent features and have branches leading to other nodes, further splitting the data. Leaf Nodes: The terminal nodes of the tree, representing the final class label predictions. Branches: Represent the decision rules based on the feature values at each internal node.

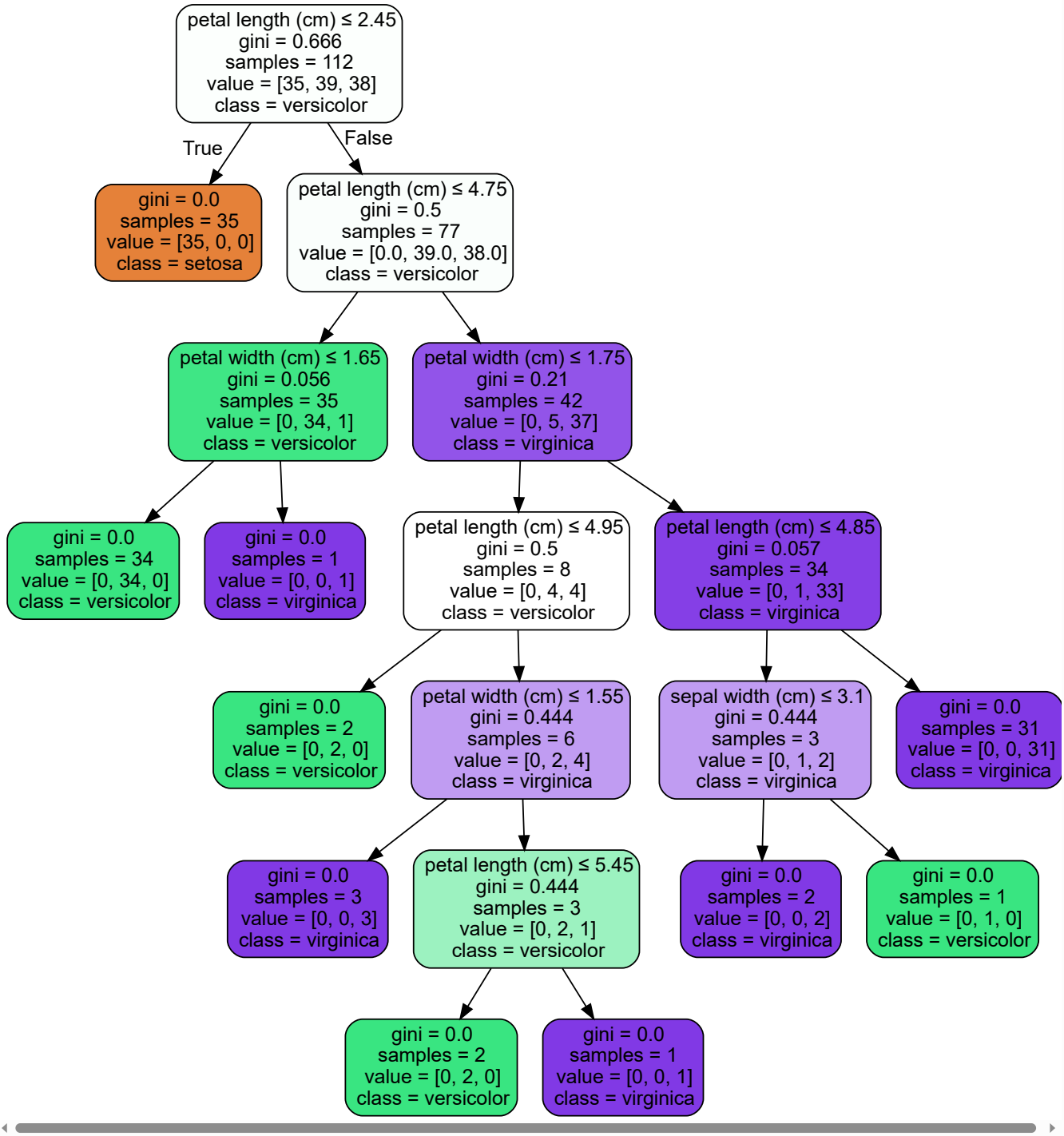
```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

```
# Decision Tree Classification Example
# Load the iris dataset
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
# Load data
```

```
iris = load_iris()
X = iris.data # feature
y = iris.target # target
# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)
# Train a Decision Tree Classifier
dt_classifier = DecisionTreeClassifier(random_state=42)
dt_classifier.fit(X_train, y_train)
# Make predictions
y_pred = dt_classifier.predict(X_test)
# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
print("Classification Report:\n", classification_report(y_test, y_pred, target_names=iris.target_name
# Visualize the decision tree (requires graphviz)
!pip install graphviz
from sklearn.tree import export_graphviz
import graphviz
dot_data = export_graphviz(dt_classifier, out_file=None,
                           feature_names=iris.feature_names,
                           class_names=iris.target_names,
                           filled=True, rounded=True,
                           special_characters=True)
graph = graphviz.Source(dot_data)
display(graph) # In Jupyter/Colab, this will render the graph
```

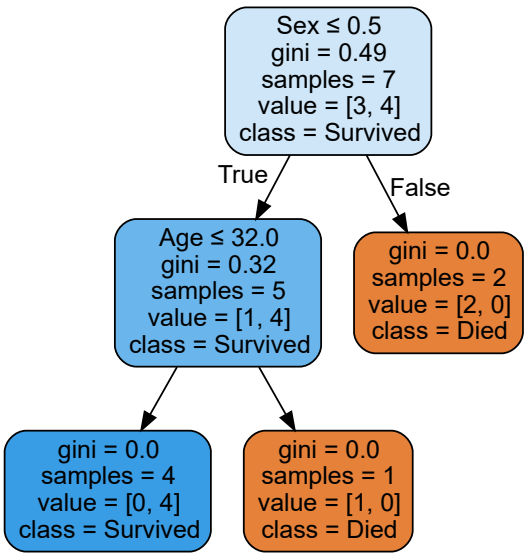
```
Accuracy: 1.0
Classification Report:
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        15
  versicolor       1.00      1.00      1.00        11
   virginica       1.00      1.00      1.00        12

    accuracy                           1.00        38
   macro avg       1.00      1.00      1.00        38
weighted avg       1.00      1.00      1.00        38

Requirement already satisfied: graphviz in /usr/local/lib/python3.11/dist-packages (0.21)
```




```
#decision tree example another
# Predicting Survival on the Titanic
# This example demonstrates building a decision tree to predict survival on the Titanic based on pass

# Step 1: Load necessary libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, export_graphviz
from sklearn.metrics import accuracy_score, classification_report
import graphviz
```

```
# Step 2: Create sample data (a simplified Titanic dataset)
data_titanic = {
    'Pclass': [1, 1, 2, 3, 3, 1, 2, 3, 2, 1],
    'Sex': ['female', 'male', 'female', 'male', 'female', 'male', 'male', 'female', 'male', 'female']
    'Age': [29, 0.9, 2, 22, 35, 54, 2, 27, 14, 4],
    'Survived': [1, 1, 1, 0, 0, 0, 0, 1, 0, 1] # 0: Died, 1: Survived
}
df_titanic = pd.DataFrame(data_titanic)
print(df_titanic)

# Step 3: Preprocessing - Convert categorical features (like 'Sex') to numerical
df_titanic['Sex'] = df_titanic['Sex'].map({'female': 0, 'male': 1})

# Step 4: Define features and target
X_titanic = df_titanic[['Pclass', 'Sex', 'Age']] # input
y_titanic = df_titanic['Survived'] # output

# Step 5: Train-test split
X_train_titanic, X_test_titanic, y_train_titanic, y_test_titanic = train_test_split(X_titanic, y_tita

# Step 6: Train a Decision Tree Classifier
dt_classifier_titanic = DecisionTreeClassifier(random_state=42)
dt_classifier_titanic.fit(X_train_titanic, y_train_titanic)
print('Titanic Survival Decision Tree model is trained successfully')

# Step 7: Make predictions
y_pred_titanic = dt_classifier_titanic.predict(X_test_titanic)

# Step 8: Evaluate the model
accuracy_titanic = accuracy_score(y_test_titanic, y_pred_titanic)
print("Accuracy:", accuracy_titanic)
print("Classification Report:\n", classification_report(y_test_titanic, y_pred_titanic, target_names=

# Step 9: Visualize the decision tree (requires graphviz)
dot_data_titanic = export_graphviz(dt_classifier_titanic, out_file=None,
                          feature_names=X_titanic.columns,
                          class_names=['Died', 'Survived'],
                          filled=True, rounded=True,
                          special_characters=True)
graph_titanic = graphviz.Source(dot_data_titanic)
display(graph_titanic)
```

```
   Pclass     Sex   Age  Survived
0       1  female  29.0         1
1       1    male   0.9         1
2       2  female   2.0         1
3       3    male  22.0         0
4       3  female  35.0         0
5       1    male  54.0         0
6       2    male   2.0         0
7       3  female  27.0         1
8       2    male  14.0         0
9       1  female   4.0         1
Titanic Survival Decision Tree model is trained successfully
Accuracy: 0.6666666666666666
Classification Report:
              precision    recall  f1-score   support

        Died       0.67      1.00      0.80         2
    Survived       0.00      0.00      0.00         1

    accuracy                           0.67         3
   macro avg       0.33      0.50      0.40         3
weighted avg       0.44      0.67      0.53         3

/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Precision is ill-defined ar
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Precision is ill-defined ar
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Precision is ill-defined ar
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```



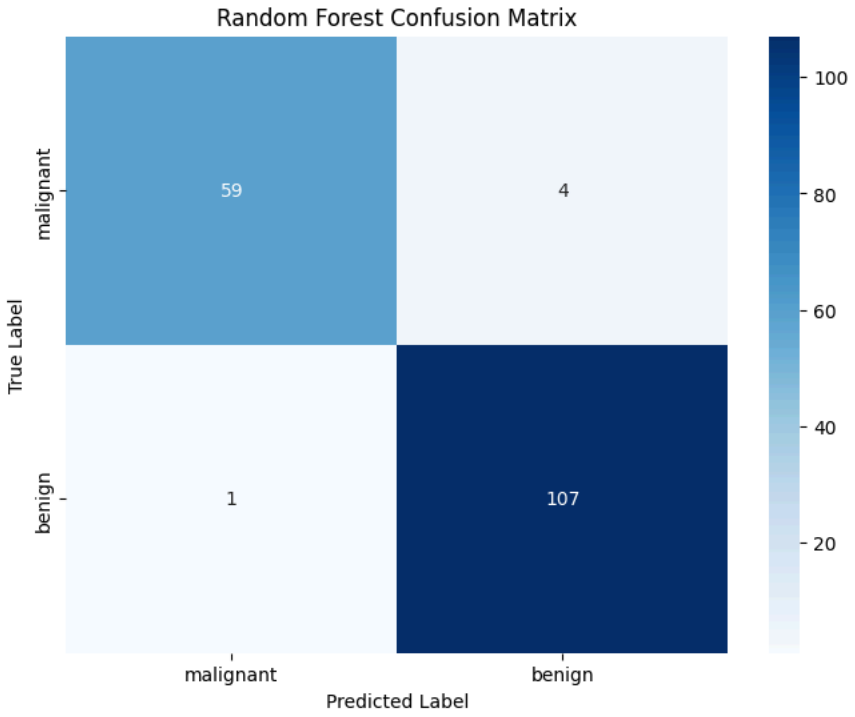
rondom forest: A random forest is a machine learning algorithm that uses an ensemble of decision trees to make predictions. It combines the outputs of multiple decision trees to achieve a more accurate and robust prediction than a single decision tree could provide. This ensemble approach helps to reduce overfitting and improve overall model performance, especially when dealing with complex datasets.

Start coding or generate with AI.

```
# rondom forest:

from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.datasets import load_breast_cancer
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load a dataset (using Breast Cancer dataset for this example)
data = load_breast_cancer() # kaggle dataset
X = pd.DataFrame(data.data, columns=data.feature_names) # feature
y = pd.Series(data.target) # target

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

```
# Create a Random Forest Classifier model
# n_estimators: the number of trees in the forest
# random_state: for reproducibility
model = RandomForestClassifier(n_estimators=100, random_state=42)

# Train the model
model.fit(X_train, y_train)

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

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred, target_names=data.target_names)
cm = confusion_matrix(y_test, y_pred) # confusion matrix use only classification

print("Random Forest Classifier Results:")
print(f"Accuracy: {accuracy:.4f}")
print("\nClassification Report:\n", report)
print("\nConfusion Matrix:\n", cm)

# Visualize the Confusion Matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=data.target_names, yticklabels=data.ta
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Random Forest Confusion Matrix')
plt.show()
```

```
Random Forest Classifier Results:
Accuracy: 0.9708

Classification Report:
              precision    recall  f1-score   support

   malignant       0.98      0.94      0.96        63
      benign       0.96      0.99      0.98       108

    accuracy                           0.97       171
   macro avg       0.97      0.96      0.97       171
weighted avg       0.97      0.97      0.97       171


Confusion Matrix:
 [[ 59   4]
 [  1 107]]
```

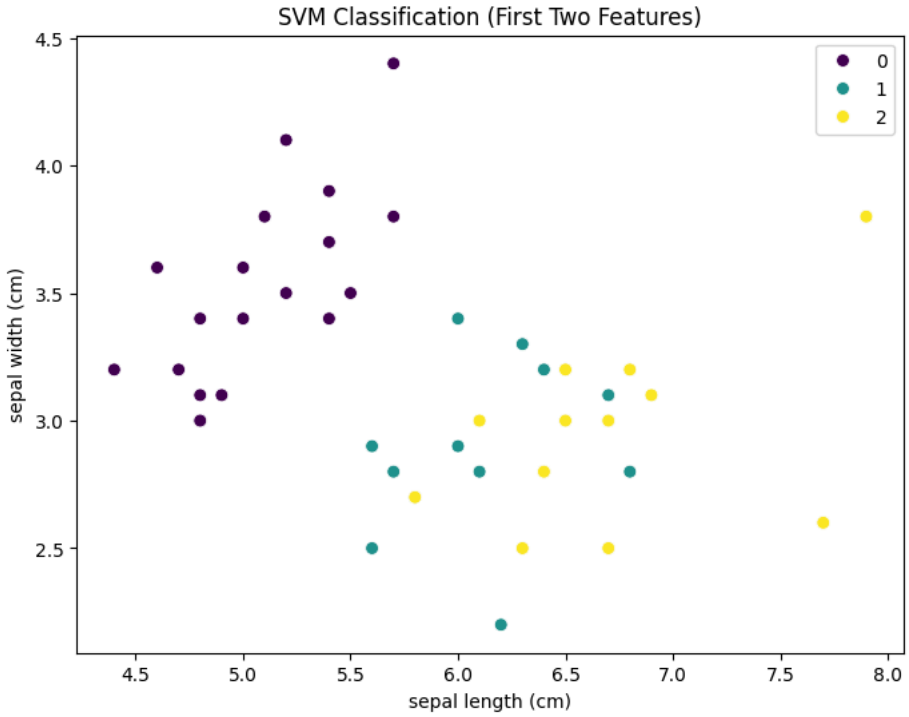


Start coding or generate with AI.

In machine learning, a Support Vector Machine (SVM) is a supervised learning model used for classification and regression tasks. It works by finding the optimal hyperplane that best separates data points into different classes, maximizing the margin between them

```
#svm

# SVM (Support Vector Machine) Classification Example
# Load the iris dataset
from sklearn.svm import SVC # Import SVC
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt # Import matplotlib for plotting
import seaborn as sns # Import seaborn for plotting
iris = load_iris()
X = iris.data
y = iris.target
# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
# Train an SVM Classifier
# kernel='linear' is a common starting point
svm_classifier = SVC(kernel='linear', random_state=42)
svm_classifier.fit(X_train, y_train)
# Make predictions
y_pred = svm_classifier.predict(X_test)
# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print("SVM Classifier Results:")
print("Accuracy:", accuracy)
print("Classification Report:\n", classification_report(y_test, y_pred, target_names=iris.target_name

# Visualize a simple scatter plot of two features colored by actual class
# This is a basic visualization for understanding the data distribution,
# not directly showing the SVM decision boundary without more complex code.
plt.figure(figsize=(8, 6))
sns.scatterplot(x=X_test[:, 0], y=X_test[:, 1], hue=y_test, palette='viridis', s=50)
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1])
plt.title('SVM Classification (First Two Features)')
plt.show()
```

```
SVM Classifier Results:
Accuracy: 1.0
Classification Report:
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        19
  versicolor       1.00      1.00      1.00        13
   virginica       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```



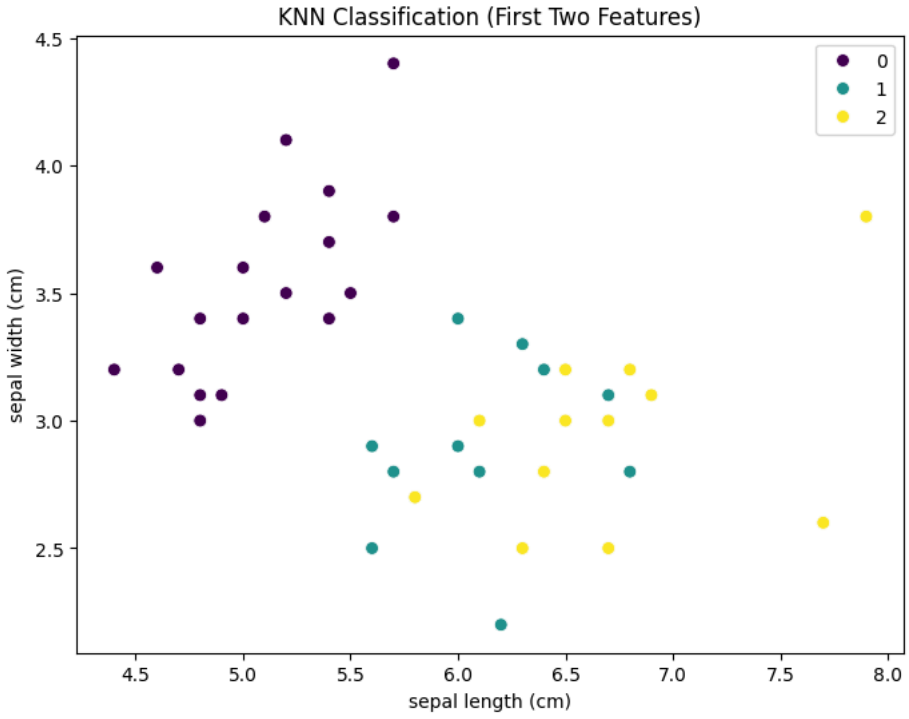
Start coding or generate with AI.

knn:In machine learning, K-Nearest Neighbors (KNN) is a supervised learning algorithm used for both classification and regression tasks. It operates on the principle that similar data points tend to be near each other. KNN classifies a new data point by considering the "k" closest data points (neighbors) from the training data and assigning it to the class that is most frequent among those neighbors (in classification) or by averaging their values (in regression).

```
# knn example

# KNN Classification Example
from sklearn.neighbors import KNeighborsClassifier
# Load the iris dataset
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt # Import matplotlib for plotting
import seaborn as sns # Import seaborn for plotting
iris = load_iris()
X = iris.data
y = iris.target
# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Train a K-Nearest Neighbors (KNN) Classifier
# n_neighbors is the 'k' value, typically chosen based on experiments
knn_classifier = KNeighborsClassifier(n_neighbors=4) # Using k=2 as an example
knn_classifier.fit(X_train, y_train)

# Make predictions
y_pred = knn_classifier.predict(X_test)
```

```
# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print("KNN Classifier Results:")
print("Accuracy:", accuracy)
print("Classification Report:\n", classification_report(y_test, y_pred, target_names=iris.target_name

# Visualize a simple scatter plot of two features colored by actual class
# This helps visualize the data points, though it doesn't directly show KNN's neighbors.
plt.figure(figsize=(8, 6))
sns.scatterplot(x=X_test[:, 0], y=X_test[:, 1], hue=y_test, palette='viridis', s=50)
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1])
plt.title('KNN Classification (First Two Features)')
plt.show()
```

```
KNN Classifier Results:
Accuracy: 1.0
Classification Report:
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        19
  versicolor       1.00      1.00      1.00        13
   virginica       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```



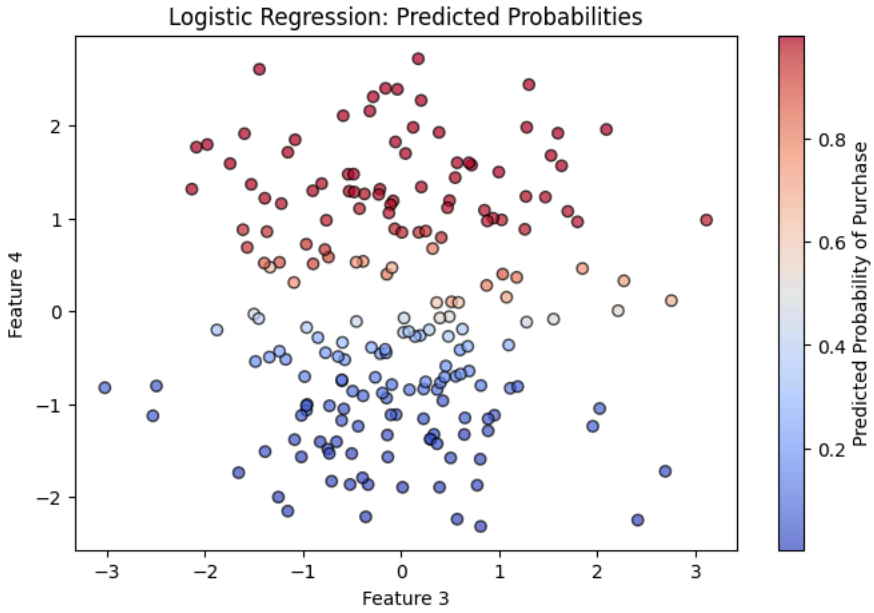
Start coding or generate with AI.

multiple classification knn: KNN serves as a most intuitive approach for tackling multi-class classification tasks. By leveraging the similarity of data points in the feature space, KNN effectively discerns between multiple classes with minimal assumptions

```
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import make_classification

X, y = make_classification(n_samples=1000, n_features=5, n_redundant=0, n_clusters_per_class=1, rando

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train logistic regression
logreg = LogisticRegression()
logreg.fit(X_train, y_train)
```

```
# Predict probabilities
probs = logreg.predict_proba(X_test)[:, 1]

# Show first 10 predicted probabilities
print("First 10 predicted probabilities:", probs[:10])
# Visualize predicted probabilities and decision boundary
plt.figure(figsize=(8, 5))
plt.scatter(X_test[:, 3], X_test[:, 4], c=probs, cmap='coolwarm', edgecolor='k', alpha=0.7)
plt.colorbar(label='Predicted Probability of Purchase')
plt.xlabel('Feature 3')
plt.ylabel('Feature 4')
plt.title('Logistic Regression: Predicted Probabilities')
plt.show()
```

```
First 10 predicted probabilities: [0.09615857 0.77533289 0.99049941 0.00717499 0.99982867 0.97873933
 0.999179   0.99348295 0.99233161 0.04224179]
```



Start coding or generate with AI.

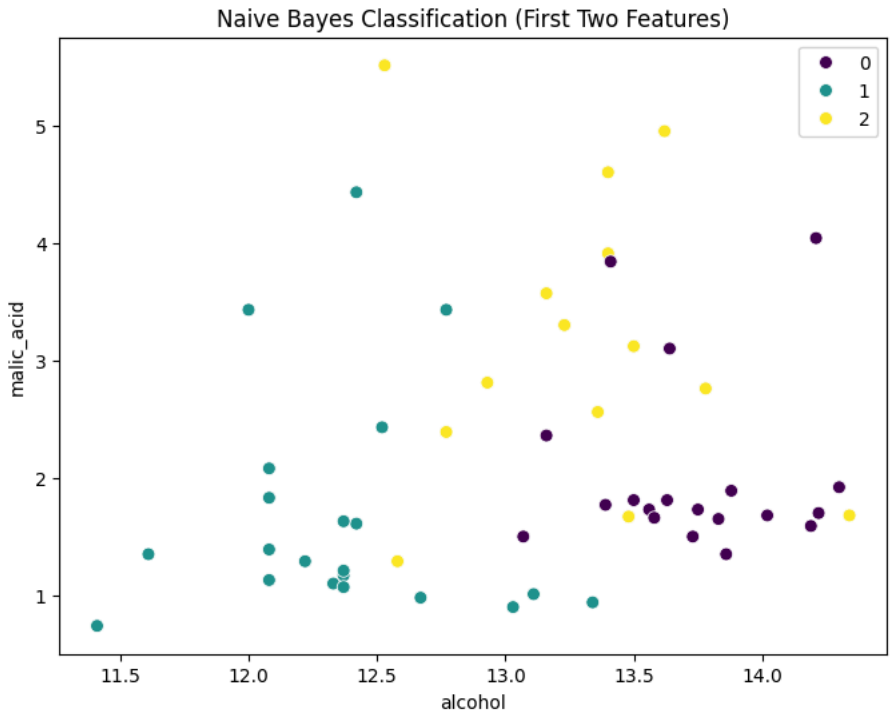
naive Bayes:

A Naive Bayes model is a simple probabilistic classifier that applies Bayes' theorem with strong (naive) independence assumptions between the features. Despite its simplicity, it's often surprisingly effective for various classification tasks, particularly text classification.

```
#naives bayes example
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import accuracy_score, classification_report
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.datasets import load_wine

# Naive Bayes Classification Example
# Load the Wine dataset
wine = load_wine()
X = wine.data
y = wine.target

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Train a Gaussian Naive Bayes classifier
nb_classifier = GaussianNB()
nb_classifier.fit(X_train, y_train)

# Make predictions
y_pred = nb_classifier.predict(X_test)
```

```
# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print("Naive Bayes Classifier Results:")
print("Accuracy:", accuracy)
print("Classification Report:\n", classification_report(y_test, y_pred, target_names=wine.target_name

# Visualize a simple scatter plot of two features colored by actual class
plt.figure(figsize=(8, 6))
sns.scatterplot(x=X_test[:, 0], y=X_test[:, 1], hue=y_test, palette='viridis', s=50)
plt.xlabel(wine.feature_names[0])
plt.ylabel(wine.feature_names[1])
plt.title('Naive Bayes Classification (First Two Features)')
plt.show()
```

```
Naive Bayes Classifier Results:
Accuracy: 1.0
Classification Report:
               precision    recall  f1-score   support

      class_0       1.00      1.00      1.00        19
      class_1       1.00      1.00      1.00        21
      class_2       1.00      1.00      1.00        14

     accuracy                           1.00        54
    macro avg       1.00      1.00      1.00        54
 weighted avg       1.00      1.00      1.00        54
```



Start coding or generate with AI.

Regression alogrithms:

1)linear regression

2)ridge regression

3)lasso regression

4)polynomial regression

5)elastic net regression

6)decision tree regression

7)SVR(Support vector regression)

linear regression: In machine learning, linear regression is a supervised learning algorithm that models the relationship between a dependent variable (the target) and one or more independent variables (features) by finding a linear equation that best fits the data. This equation is

essentially a straight line (in the simplest case) or a hyperplane (in higher dimensions) that minimizes the difference between predicted and actual values.

```
# Step 1: Import libraries
import numpy as np
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt

# Step 2: Sample data
# House sizes (in sqft)
X = np.array([[500], [750], [1000], [1250], [1500]]) # feature
# House prices (in $1000s)
y = np.array([150, 200, 250, 300, 350]) # target

# Step 3: Train the model
model = LinearRegression()
model.fit(X, y)

# Step 4: Predict a new price
size = np.array([[1100]])
predicted_price = model.predict(size)
print(f"Predicted price for 1100 sqft house: ${predicted_price[0]*1000:.2f}")

# Step 5: Visualization
plt.scatter(X, y, color='blue', label='Data points')
plt.plot(X, model.predict(X), color='red', label='Regression line')
plt.xlabel("Size (sqft)")
plt.ylabel("Price ($1000s)")
plt.title("Simple House Price Prediction")
plt.legend()
plt.grid(True)
plt.show()
```

```
Predicted price for 1100 sqft house: $270000.00
```


```
#Linear Regression
# Predict the price of a house based on its size

# Step 1: Import libraries (already done above)
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt

# Step 2: Create sample data
```

```
data_lr = {
    'Size_sqft': [1000, 1500, 1200, 1800, 2000, 1300, 1600, 1900],
    'Price_usd': [250000, 350000, 300000, 420000, 450000, 310000, 380000, 430000]
}
df_house = pd.DataFrame(data_lr)
print(df_house)

# Step 3: Define features and target
# Reshape X to be a 2D array (required by scikit-learn)
X_house = df_house[['Size_sqft']].values # input (independent variable)
y_house = df_house['Price_usd'].values # output (dependent variable)

# Step 4: Train-test split (optional for this very small dataset, but good practice)
X_train_house, X_test_house, y_train_house, y_test_house = train_test_split(X_house, y_house, test_s

# Step 5: Train the Linear Regression model
model_house = LinearRegression()
model_house.fit(X_train_house, y_train_house)
print(f"Training samples: {X_train_house.shape[0]}, Test samples: {X_test_house.shape[0]}")
print('House Price Linear Regression model is trained successfully')

# Step 6: Predict on test data
y_pred_house = model_house.predict(X_test_house)
print("Predictions on test set:", y_pred_house)
print("Actual test set values:", y_test_house)

# Step 7: Evaluate metric
# Mean Squared Error (MSE)
mse_house = mean_squared_error(y_test_house, y_pred_house)
print(f"Mean Squared Error: {mse_house:.2f}")

# R-squared (Coefficient of Determination)
r2_house = r2_score(y_test_house, y_pred_house)
print(f"R-squared: {r2_house:.2f}")

# Step 8: Print the model coefficients
print(f"Intercept: {model_house.intercept_:.2f}")
print(f"Coefficient (Size_sqft): {model_house.coef_[0]:.2f}")

# Step 9: Visualization
plt.figure(figsize=(10, 6))
# Plot training data
plt.scatter(X_train_house, y_train_house, color='blue', label='Training Data')
# Plot test data
plt.scatter(X_test_house, y_test_house, color='green', label='Test Data')
# Plot the regression line
plt.plot(X_test_house, y_pred_house, color='red', linewidth=2, label='Regression Line')
plt.xlabel('Size (sqft)')
plt.ylabel('Price (USD)')
plt.title('House Price Prediction using Linear Regression')
plt.legend()
plt.show()
```

Show hidden output

```
Start coding or generate with AI.
```

ridge regression:

Ridge regression is a linear regression technique that adds a regularization term to the standard linear regression cost function. This regularization term, also known as L2 regularization, penalizes the model for large coefficients, helping to prevent overfitting and mitigate multicollinearity. In essence, it shrinks the coefficients towards zero, making the model more robust and improving its ability to generalize to unseen data.

```
# ridge ression  simple data example

from sklearn.linear_model import Ridge
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# Simple Ridge Regression Example
# Predict a value based on two features

# Step 1: Create sample data
# Using the house price data structure but adding another feature
data_ridge = {
    'Size_sqft': [1000, 1500, 1200, 1800, 2000, 1300, 1600, 1900, 1100, 1700],
    'Num_Bedrooms': [2, 3, 2, 4, 4, 3, 3, 4, 2, 3],
    'Price_usd': [250000, 350000, 300000, 420000, 450000, 310000, 380000, 430000, 270000, 400000]
}
df_ridge = pd.DataFrame(data_ridge)
print(df_ridge)

# Step 2: Define features and target
X_ridge = df_ridge[['Size_sqft', 'Num_Bedrooms']].values # input features
y_ridge = df_ridge['Price_usd'].values # output target

# Step 3: Train-test split
X_train_ridge, X_test_ridge, y_train_ridge, y_test_ridge = train_test_split(X_ridge, y_ridge, test_s

# Step 4: Train the Ridge Regression model
# alpha is the regularization strength (lambda in some contexts)
# A smaller alpha means less regularization (closer to Linear Regression)
# A larger alpha means more regularization
model_ridge = Ridge(alpha=1.0) # alpha=1.0 is the default
model_ridge.fit(X_train_ridge, y_train_ridge)
print(f"Training samples: {X_train_ridge.shape[0]}, Test samples: {X_test_ridge.shape[0]}")
print('House Price Ridge Regression model is trained successfully')

# Step 5: Predict on test data
y_pred_ridge = model_ridge.predict(X_test_ridge)
print("Predictions on test set:", y_pred_ridge)
print("Actual test set values:", y_test_ridge)

# Step 6: Evaluate metric
mse_ridge = mean_squared_error(y_test_ridge, y_pred_ridge)
print(f"Mean Squared Error (Ridge): {mse_ridge:.2f}")

r2_ridge = r2_score(y_test_ridge, y_pred_ridge)
print(f"R-squared (Ridge): {r2_ridge:.2f}")

# Step 7: Print the model coefficients
print(f"Intercept (Ridge): {model_ridge.intercept_:.2f}")
print(f"Coefficients (Ridge): {model_ridge.coef_}")

# Step 8: Visualization (limited for multi-dimensional features)
# We can plot the actual vs predicted values
plt.figure(figsize=(8, 5))
plt.scatter(y_test_ridge, y_pred_ridge)
plt.xlabel("Actual Prices")
plt.ylabel("Predicted Prices (Ridge)")
plt.title("Ridge Regression: Actual vs Predicted Prices")
plt.plot([y_ridge.min(), y_ridge.max()], [y_ridge.min(), y_ridge.max()], 'k--', lw=2) # Diagonal lin
plt.show()
```

Show hidden output

lasso regression:

Lasso regression, or Least Absolute Shrinkage and Selection Operator, is a linear regression technique that adds a penalty term to the standard linear regression model, which helps in feature selection and prevents overfitting. This penalty is based on the absolute values of the coefficients, and it encourages some coefficients to be exactly zero, effectively removing those features from the model.

Regularization: Lasso regression is a type of regularization technique, specifically L1 regularization. Regularization is used to prevent overfitting, which occurs when a model learns the training data too well, including its noise, and performs poorly on unseen data.

Penalty Term: Lasso adds a penalty to the loss function based on the sum of the absolute values of the regression coefficients (L1 norm). This penalty term is controlled by a hyperparameter (usually denoted as lambda or α), which determines the strength of the regularization

Start coding or generate with AI.

```
# lasso regression
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.linear_model import Lasso

# Simple Lasso Regression Example
# Predict a value based on two features, aiming for feature selection

# Step 1: Create sample data
# Using the house price data structure with possibly irrelevant features
data_lasso = {
    'Size_sqft': [1000, 1500, 1200, 1800, 2000, 1300, 1600, 1900, 1100, 1700],
    'Num_Bedrooms': [2, 3, 2, 4, 4, 3, 3, 4, 2, 3],
    'Year_Built': [2005, 1995, 2010, 1990, 2000, 2015, 1985, 2008, 2012, 1998], # Potentially less r
    'Gardern_Size': [100, 150, 120, 180, 200, 130, 160, 190, 110, 170], # Potentially less relevant
    'Price_usd': [250000, 350000, 300000, 420000, 450000, 310000, 380000, 430000, 270000, 400000]
}
df_lasso = pd.DataFrame(data_lasso)
print(df_lasso)

# Step 2: Define features and target
X_lasso = df_lasso[['Size_sqft', 'Num_Bedrooms', 'Year_Built', 'Gardern_Size']].values # input featu
y_lasso = df_lasso['Price_usd'].values # output target

# Step 3: Train-test split
X_train_lasso, X_test_lasso, y_train_lasso, y_test_lasso = train_test_split(X_lasso, y_lasso, test_s

# Step 4: Train the Lasso Regression model
# alpha is the regularization strength (lambda in some contexts)
# Lasso can shrink some coefficients to zero, performing feature selection.
# A larger alpha means more regularization and more coefficients might become zero.
model_lasso = Lasso(alpha=10000) # Using a somewhat larger alpha to demonstrate shrinkage
model_lasso.fit(X_train_lasso, y_train_lasso)
print(f"Training samples: {X_train_lasso.shape[0]}, Test samples: {X_test_lasso.shape[0]}")
print('House Price Lasso Regression model is trained successfully')

# Step 5: Predict on test data
y_pred_lasso = model_lasso.predict(X_test_lasso)
print("Predictions on test set:", y_pred_lasso)
print("Actual test set values:", y_test_lasso)

# Step 6: Evaluate metric
mse_lasso = mean_squared_error(y_test_lasso, y_pred_lasso)
print(f"Mean Squared Error (Lasso): {mse_lasso:.2f}")

r2_lasso = r2_score(y_test_lasso, y_pred_lasso)
print(f"R-squared (Lasso): {r2_lasso:.2f}")
```

```
# Step 7: Print the model coefficients
print(f"Intercept (Lasso): {model_lasso.intercept_:.2f}")
print(f"Coefficients (Lasso): {model_lasso.coef_}") # Observe if any coefficients are close to zero

# Step 8: Visualization (limited for multi-dimensional features)
# We can plot the actual vs predicted values
plt.figure(figsize=(8, 5))
plt.scatter(y_test_lasso, y_pred_lasso)
plt.xlabel("Actual Prices")
plt.ylabel("Predicted Prices (Lasso)")
plt.title("Lasso Regression: Actual vs Predicted Prices")
plt.plot([y_lasso.min(), y_lasso.max()], [y_lasso.min(), y_lasso.max()], 'k--', lw=2) # Diagonal lin
plt.show()
```

Show hidden output

polynomial regression:

Polynomial regression is a form of regression analysis that models the relationship between a dependent variable and one or more independent variables as an nth-degree polynomial. Unlike linear regression, which assumes a linear relationship, polynomial regression can capture non-linear relationships by including polynomial terms (e.g., x^2, x^3) in the equation. This allows it to fit curves and complex patterns in data more accurately.

```
# Predict the price of a house based on its size, allowing for a non-linear relationship
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures # Import PolynomialFeatures
from sklearn.linear_model import LinearRegression # Import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score # Import metrics

# Step 1: Create sample data
# Creating data that might have a non-linear relationship
data_poly = {
    'Size_sqft': [500, 700, 900, 1100, 1300, 1500, 1700, 1900, 2100, 2300],
    'Price_usd': [150000, 180000, 220000, 270000, 330000, 400000, 480000, 570000, 670000, 780000] #
}
df_poly = pd.DataFrame(data_poly)
print(df_poly)

# Step 2: Define features and target
# Reshape X to be a 2D array
X_poly = df_poly[['Size_sqft']].values # input (independent variable)
y_poly = df_poly['Price_usd'].values # output (dependent variable)

# Step 3: Transform features to polynomial features
# degree=2 creates features x and x^2
poly_features = PolynomialFeatures(degree=2)
X_poly_features = poly_features.fit_transform(X_poly)

# Step 4: Train-test split
X_train_poly, X_test_poly, y_train_poly, y_test_poly = train_test_split(X_poly_features, y_poly, tes
# Need the original X_test as well for plotting
X_test_original = X_test_poly[:, 1].reshape(-1, 1) # Get the original feature column (index 1 after

# Step 5: Train the Linear Regression model on the polynomial features
model_poly = LinearRegression()
model_poly.fit(X_train_poly, y_train_poly)
print(f"Training samples: {X_train_poly.shape[0]}, Test samples: {X_test_poly.shape[0]}")
print('House Price Polynomial Regression model is trained successfully')

# Step 6: Predict on test data
```

```
y_pred_poly = model_poly.predict(X_test_poly)
print("Predictions on test set:", y_pred_poly)
print("Actual test set values:", y_test_poly)

# Step 7: Evaluate metric
mse_poly = mean_squared_error(y_test_poly, y_pred_poly)
print(f"Mean Squared Error (Polynomial): {mse_poly:.2f}")

r2_poly = r2_score(y_test_poly, y_pred_poly)
print(f"R-squared (Polynomial): {r2_poly:.2f}")

# Step 8: Print the model coefficients
print(f"Intercept (Polynomial): {model_poly.intercept_:.2f}")
print(f"Coefficients (Polynomial): {model_poly.coef_}")

# Step 9: Visualization
plt.figure(figsize=(10, 6))
# Plot actual data points
plt.scatter(X_poly, y_poly, color='blue', label='Actual Data') # Use original X for scatter plot
plt.plot(X_test_poly, y_pred_poly, color='red', linewidth=2, label='Polynomial Regression Line')
plt.xlabel('Size (sqft)')
plt.ylabel('Price (USD)')
plt.title('House Price Prediction using Polynomial Regression (Degree 2)')
plt.legend()
plt.show()
```

Show hidden output

```
Start coding or generate with AI.
```

Elastic Net Regression:

Elastic Net Regression is a linear regression model that combines the penalties of both Lasso and Ridge regression methods. It's a regularized regression technique that helps to prevent overfitting and perform feature selection, especially when dealing with multicollinearity (when predictors are highly correlated)

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score

from sklearn.linear_model import Ridge, Lasso, ElasticNet
from sklearn.preprocessing import PolynomialFeatures

# Simple Elastic Net Regression Example
# Predict a value based on two features, combining L1 and L2 regularization

# Step 1: Create sample data
# Using the house price data structure with potentially less relevant features
data_enet = {
    'Size_sqft': [1000, 1500, 1200, 1800, 2000, 1300, 1600, 1900, 1100, 1700],
    'Num_Bedrooms': [2, 3, 2, 4, 4, 3, 3, 4, 2, 3],
    'Year_Built': [2005, 1995, 2010, 1990, 2000, 2015, 1985, 2008, 2012, 1998], # Potentially less r
    'Gardern_Size': [100, 150, 120, 180, 200, 130, 160, 190, 110, 170], # Potentially less relevant
    'Price_usd': [250000, 350000, 300000, 420000, 450000, 310000, 380000, 430000, 270000, 400000]
}
df_enet = pd.DataFrame(data_enet)
print(df_enet)

# Step 2: Define features and target
X_enet = df_enet[['Size_sqft', 'Num_Bedrooms', 'Year_Built', 'Gardern_Size']].values # input feature
y_enet = df_enet['Price_usd'].values # output target

# Step 3: Train-test split
```

```
X_train_enet, X_test_enet, y_train_enet, y_test_enet = train_test_split(X_enet, y_enet, test_size=0.

# Step 4: Train the Elastic Net Regression model
model_enet = ElasticNet(alpha=0.1, l1_ratio=0.5, random_state=42) # Example alpha and l1_ratio
model_enet.fit(X_train_enet, y_train_enet)
print(f"Training samples: {X_train_enet.shape[0]}, Test samples: {X_test_enet.shape[0]}")
print('House Price Elastic Net Regression model is trained successfully')

# Step 5: Predict on test data
y_pred_enet = model_enet.predict(X_test_enet)
print("Predictions on test set:", y_pred_enet)
print("Actual test set values:", y_test_enet)

# Step 6: Evaluate metric
mse_enet = mean_squared_error(y_test_enet, y_pred_enet)
print(f"Mean Squared Error (Elastic Net): {mse_enet:.2f}")

r2_enet = r2_score(y_test_enet, y_pred_enet)
print(f"R-squared (Elastic Net): {r2_enet:.2f}")

# Step 7: Print the model coefficients
print(f"Intercept (Elastic Net): {model_enet.intercept_:.2f}")
print(f"Coefficients (Elastic Net): {model_enet.coef_}") # Observe the coefficients

# Step 8: Visualization (limited for multi-dimensional features)
# We can plot the actual vs predicted values
plt.figure(figsize=(8, 5))
plt.scatter(y_test_enet, y_pred_enet)
plt.xlabel("Actual Prices")
plt.ylabel("Predicted Prices (Elastic Net)")
plt.title("Elastic Net Regression: Actual vs Predicted Prices")
plt.plot([y_enet.min(), y_enet.max()], [y_enet.min(), y_enet.max()], 'k--', lw=2) # Diagonal line fo
plt.show()
```

Show hidden output

Gradient Boosting Regression:

Gradient boosting regression is a machine learning technique used for predicting continuous numerical values. It builds an ensemble of decision trees sequentially, where each new tree attempts to correct the errors made by the previous ones. This iterative process of adding trees, adjusting predictions, and minimizing a loss function (like mean squared error) leads to a highly accurate and robust predictive model.

```
# Gradient Boosting Regression
# Using a simple synthetic dataset
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score

# Step 1: Import libraries
from sklearn.ensemble import GradientBoostingRegressor # Import the regressor
from sklearn.datasets import make_regression # For creating synthetic data

# (Other imports like train_test_split, mean_squared_error, r2_score, numpy are already present)

# Step 2: Create a simple synthetic regression dataset
X_gr, y_gr = make_regression(n_samples=100, n_features=5, n_informative=3, noise=10, random_state=42
print("Synthetic Regression Data created.")

# Step 3: Train-test split
X_train_gr, X_test_gr, y_train_gr, y_test_gr = train_test_split(X_gr, y_gr, test_size=0.3, random_st

# Step 4: Create a Gradient Boosting Regressor model
model_gr = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1, max_depth=3, random_state=
```

```
# Step 5: Train the model
model_gr.fit(X_train_gr, y_train_gr)
print(f"Training samples: {X_train_gr.shape[0]}, Test samples: {X_test_gr.shape[0]}")
print('Gradient Boosting Regressor model is trained successfully')

# Step 6: Predict on test data
y_pred_gr = model_gr.predict(X_test_gr)
print("Predictions on test set:", y_pred_gr[:5]) # Print first 5 predictions
print("Actual test set values:", y_test_gr[:5]) # Print first 5 actual values

# Step 7: Evaluate metric
mse_gr = mean_squared_error(y_test_gr, y_pred_gr)
print(f"Mean Squared Error (Gradient Boosting Regression): {mse_gr:.2f}")

r2_gr = r2_score(y_test_gr, y_pred_gr)
print(f"R-squared (Gradient Boosting Regression): {r2_gr:.2f}")

# Step 8: Visualization (limited for multi-dimensional features)
# We can plot the actual vs predicted values
plt.figure(figsize=(8, 5))
plt.scatter(y_test_gr, y_pred_gr)
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values (Gradient Boosting Regression)")
plt.title("Gradient Boosting Regression: Actual vs Predicted Values")
# Add a diagonal line for reference (perfect prediction)
min_val = min(y_test_gr.min(), y_pred_gr.min())
max_val = max(y_test_gr.max(), y_pred_gr.max())
plt.plot([min_val, max_val], [min_val, max_val], 'k--', lw=2)
plt.show()
```

Show hidden output

```
# Gradient Boosting Classification
# Using a dataset like Breast Cancer
#Step 1: Import libraries
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

from sklearn.ensemble import GradientBoostingClassifier # Import the classifier

# (Other imports like pandas, train_test_split, accuracy_score, classification_report, load_breast_c

# Load a dataset (using Breast Cancer dataset again)
data_gb = load_breast_cancer()
X_gb = pd.DataFrame(data_gb.data, columns=data_gb.feature_names)
y_gb = pd.Series(data_gb.target)

# Split data into training and testing sets
X_train_gb, X_test_gb, y_train_gb, y_test_gb = train_test_split(X_gb, y_gb, test_size=0.3, random_st

# Step 2: Create a Gradient Boosting Classifier model
model_gb = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1, max_depth=3, random_state

# Step 3: Train the model
model_gb.fit(X_train_gb, y_train_gb)
print(f"Training samples: {X_train_gb.shape[0]}, Test samples: {X_test_gb.shape[0]}")
print('Gradient Boosting Classifier model is trained successfully')

# Step 4: Make predictions on the test set
y_pred_gb = model_gb.predict(X_test_gb)
```

```
# Step 5: Evaluate the model
accuracy_gb = accuracy_score(y_test_gb, y_pred_gb)
report_gb = classification_report(y_test_gb, y_pred_gb, target_names=data_gb.target_names)
cm_gb = confusion_matrix(y_test_gb, y_pred_gb)

print("Gradient Boosting Classifier Results:")
print(f"Accuracy: {accuracy_gb:.4f}")
print("\nClassification Report:\n", report_gb)
print("\nConfusion Matrix:\n", cm_gb)

# Step 6: Visualize the Confusion Matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm_gb, annot=True, fmt='d', cmap='Blues', xticklabels=data_gb.target_names, yticklabels=
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Gradient Boosting Confusion Matrix')
plt.show()
```

Show hidden output

support vector regression:

Support Vector Regression (SVR) is a powerful machine learning algorithm used for regression tasks, which aims to predict continuous values. It builds upon the principles of Support Vector Machines (SVMs) by finding a hyperplane that best fits the data within a specified margin of error, known as the epsilon-insensitive tube.

```
#support vector regression

# Step 1: Import libraries (already done above)
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.svm import SVR
from sklearn.datasets import make_regression
# Step 2: Create a simple synthetic regression dataset
X_svr, y_svr = make_regression(n_samples=100, n_features=5, n_informative=3, noise=10, random_state=
print("Synthetic Regression Data created for SVR.")

# Step 3: Train-test split
X_train_svr, X_test_svr, y_train_svr, y_test_svr = train_test_split(X_svr, y_svr, test_size=0.3, ran

# Step 4: Create a Support Vector Regressor model
# kernel can be 'linear', 'poly', 'rbf', etc
model_svr = SVR(kernel='rbf', C=100, epsilon=0.1)

# Step 5: Train the model
model_svr.fit(X_train_svr, y_train_svr)
print(f"Training samples: {X_train_svr.shape[0]}, Test samples: {X_test_svr.shape[0]}")
print('Support Vector Regressor model is trained successfully')

# Step 6: Predict on test data
y_pred_svr = model_svr.predict(X_test_svr)
print("Predictions on test set:", y_pred_svr[:5]) # Print first 5 predictions
print("Actual test set values:", y_test_svr[:5]) # Print first 5 actual values

# Step 7: Evaluate metric
mse_svr = mean_squared_error(y_test_svr, y_pred_svr)
print(f"Mean Squared Error (Support Vector Regression): {mse_svr:.2f}")

r2_svr = r2_score(y_test_svr, y_pred_svr)
print(f"R-squared (Support Vector Regression): {r2_svr:.2f}")
```

```
# Step 8: Visualization (limited for multi-dimensional features)
# We can plot the actual vs predicted values
plt.figure(figsize=(8, 5))
plt.scatter(y_test_svr, y_pred_svr)
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values (SVR)")
plt.title("Support Vector Regression: Actual vs Predicted Values")
# Add a diagonal line for reference (perfect prediction)
min_val = min(y_test_svr.min(), y_pred_svr.min())
max_val = max(y_test_svr.max(), y_pred_svr.max())
plt.plot([min_val, max_val], [min_val, max_val], 'k--', lw=2)
plt.show()
```

Show hidden output

unsupervised learning:

Unsupervised learning is a type of machine learning where algorithms learn from unlabeled data, discovering hidden patterns and structures without explicit guidance. Unlike supervised learning, there are no predefined outputs or labels to train against

1. Clustering: This technique groups data points into clusters based on similarity. Common clustering algorithms include: K-Means: Partitions data into k clusters, where each data point belongs to the cluster with the nearest mean (centroid).

Hierarchical Clustering: Builds a hierarchy of clusters, either by merging smaller clusters (agglomerative) or splitting larger clusters (divisive)

2. Association Rule Mining: This type of unsupervised learning aims to discover relationships between items in a dataset. A common example is market basket analysis, where retailers analyze customer purchases to find associations between products. The Apriori algorithm is a popular method for association rule mining.
3. Dimensionality Reduction: This technique reduces the number of variables in a dataset while preserving its essential characteristics. Principal Component Analysis (PCA) is a widely used dimensionality reduction algorithm.

```
Start coding or generate with AI.
```

1)clustring:customer segmention data

```
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Sample customer data
data = {
    'Age': [22, 25, 47, 52, 46, 56, 55, 60, 26, 27],
    'Annual Income (k$)': [15, 18, 25, 52, 40, 60, 52, 61, 19, 20],
    'Spending Score': [39, 81, 6, 77, 40, 70, 60, 80, 76, 94]
}
df = pd.DataFrame(data)

# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(df)

# Apply KMeans
kmeans = KMeans(n_clusters=3, random_state=42)
df['Segment'] = kmeans.fit_predict(X_scaled)

# Show segmented customers
print(df)

# Visualize clusters (in 2D using first two features)
plt.scatter(df['Annual Income (k$)'], df['Spending Score'], c=df['Segment'], cmap='viridis')
plt.xlabel('Annual Income (k$)')
plt.ylabel('Spending Score')
plt.title('Customer Segments')
```

```
plt.grid(True)
plt.show()
```

Show hidden output

real time data for customers

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

# 1. Load data from CSV
df = pd.read_csv('/content/customers.csv')

# 2. Optional: Preview your data
print(df.head())

# 3. Select features for clustering
X = df[['Age', 'Annual Income (k$)', 'Spending Score (1-100)']]  # You can add 'Age' if needed

# 4. Scale the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# 5. Apply KMeans clustering
kmeans = KMeans(n_clusters=3, random_state=42, n_init=10) # Added n_init for KMeans
df['Segment'] = kmeans.fit_predict(X_scaled)

# 6. Visualize the clusters (using the original columns for better readability on the plot)
plt.figure(figsize=(8, 5))
sns.scatterplot(data=df, x='Annual Income (k$)', y='Spending Score (1-100)', hue='Segment', palette='
plt.title('Customer Segmentation')
plt.xlabel('Annual Income ($k)')
plt.ylabel('Spending Score (1-100)')
plt.grid(True)
plt.show()

# 7. Save segmented data (optional)
df.to_csv('segmented_customers.csv', index=False)
```

Show hidden output

```
Start coding or generate with AI.
```

```
#load_iris data k-means
import numpy
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MultiLabelBinarizer
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans
import warnings
warnings.filterwarnings('ignore')

# Set plot style
sns.set(style='whitegrid')
iris = load_iris()
X = iris.data

kmeans = KMeans(n_clusters=2, random_state=42)
```