EXP.NO : **EXP.DATE :**

CENTRAL TENDENCY MEASURE AND MEASURE OF DISPERSION:

AIM: To Write a Python Program On Compute Central Tendency Measures: Mean, Median, Mode Measure of Dispersion: Variance, Standard Deviation.

DESCRIPTION:

Central Tendency Measures

These describe the center or typical value of a dataset.

1. **Mean (Average)**

- **Definition**: The sum of all values divided by the number of values.
- **Formula**:

\text{Mean} = \frac{\sum x_i}{n}

- **Use**: Best for data without extreme outliers.
- **Example**: For [10, 20, 30],
  Mean = \frac{10 + 20 + 30}{3} = 20

2. **Median**

- **Definition**: The middle value when data is sorted.
- **Use**: Ideal when data has outliers or is skewed.
- **Example**:
- Odd set: [1, 3, 5] → Median = 3
- Even set: [1, 3, 5, 7] → Median = \frac{3 + 5}{2} = 4

3. **Mode**

- **Definition**: The value that appears most frequently.
- **Use**: Useful for categorical or discrete data.
- **Example**: [2, 4, 4, 5] → Mode = 4

Measures of Dispersion

These describe how spread out the data is.

1. **Variance**

- **Definition**: The average of squared differences from the mean.
- **Formula**:

\text{Variance} = \frac{\sum (x_i - \mu)^2}{n}

- **Use**: Indicates how much values deviate from the mean.
- **Example**: For [2, 4, 4, 4, 5, 5, 7, 9], variance ≈ 4.0

2. **Standard Deviation**

- **Definition**: The square root of the variance.
- **Formula**:

\text{Standard Deviation} = \sqrt{\text{Variance}}

- **Use**: Easier to interpret than variance since it's in the same units as the data.
- **Example**: For the same list, standard deviation ≈ 2.0

ALGORITHM:

Step 1: Import Required Libraries

- Import numpy as np for numerical operations.
- Import pandas as pd (not used in this code but included).
- Import stats from scipy for statistical functions.

Step 2: Calculate Mean

- Define a list data = [10, 20, 30, 40, 50].
- Use np.mean(data) to compute the mean.
- Print the result.

Step 3: Calculate Median

- Define a list data_odd = [1, 2, 3, 4, 5].
    - Use np.median(data_odd) to compute the median for an odd-length list.
    - Print the result.
- Define a list data_even = [1, 2, 3, 4, 5, 6].
- Use np.median(data_even) to compute the median for an even-length list.
- Print the result.

Step 4: Calculate Mode

- Use stats.mode(data, keepdims=True) to compute the mode of the original data list.
- Extract the mode value using .mode[0].
- Print the result.

Step 5: Calculate Variance and Standard Deviation

- Define a list list = [2, 4, 4, 4, 5, 5, 7, 9].
- Use np.var(list) to compute the variance.
- Use np.std(list) to compute the standard deviation.
- Print both results.

PROGRAM:

```
import numpy as np
import pandas as pd
from scipy import stats
#mean
data = [10, 20, 30, 40, 50]
mean_value = np.mean(data)
print(f"Mean: {mean_value}")
#median-odd
data_odd = [1, 2, 3, 4, 5]
median_odd = np.median(data_odd)
print(f"Median (odd): {median_odd}")
#median-even
data_even = [1, 2, 3, 4, 5, 6]
median_even = np.median(data_even)
print(f"Median (even): {median_even}")
#mode
mode_result = stats.mode(data, keepdims=True)
print(f"Mode: {mode_result.mode[0]}")
#variance
list=[2,4,4,4,5,5,7,9]
print(f"variance {np.var(list)}")
#standard deviation
print(f"standard deviation {np.std(list)}")
```

OUTPUT:

Mean: 30.0

Median (odd): 3.0

Median (even): 3.5

Mode: 10

variance 4.0

standard deviation 2.0

RESULT: Hence, The python program was executed successfully.

EXP.NO : **EXP.DATE :**

PRE-PROCESSING TECHNIQUES:

AIM: To Write a Python Program On the following Pre-processing techniques for a given dataset.

a. Attribute selection
b. Handling Missing Values
c. Discretization
d. Elimination of Outliers

DESCRIPTION:

Preprocessing Techniques Explained

a. **Attribute Selection (Feature Selection)**

- **Purpose**: Reduce dimensionality by selecting only the most relevant features.
- **Why it matters**: Irrelevant or redundant features can slow down training and reduce model accuracy.
- **Techniques**:
  - **Filter methods**: Use statistical tests (e.g., correlation, chi-square).
  - **Wrapper methods**: Use model performance to evaluate subsets (e.g., recursive feature elimination).
  - **Embedded methods**: Feature selection is part of the model (e.g., Lasso regression).
- **Example**: In a dataset with 100 features, you might select the top 10 that have the highest correlation with the target variable.

b. **Handling Missing Values**

- **Purpose**: Fill in or remove missing data to avoid errors or bias.
- **Why it matters**: Many algorithms can't handle missing values directly.
- **Techniques**:
  - **Deletion**: Remove rows or columns with too many missing values.
  - **Imputation**:

- **Mean/Median/Mode**: Fill missing values with central tendency.
- **Forward/Backward Fill**: Use neighboring values (common in time series).
- **Model-based**: Predict missing values using regression or KNN.

- **Example**: If a column has missing age values, you might fill them with the mean age.

c. **Discretization**

- **Purpose**: Convert continuous variables into categorical bins.
- **Why it matters**: Some models (like decision trees) work better with categorical data.
- **Techniques**:
  - **Equal-width binning**: Divide range into equal intervals.
  - **Equal-frequency binning**: Each bin has the same number of samples.
  - **Custom binning**: Based on domain knowledge.
- **Example**: Convert age into categories like "Youth", "Adult", "Senior".

d. **Elimination of Outliers**

- **Purpose**: Remove extreme values that distort analysis or model performance.
- **Why it matters**: Outliers can skew statistics and lead to poor predictions.
- **Techniques**:
  - **Z-score method**: Remove values with z-scores beyond a threshold (e.g., >3 or <-3).
  - **IQR method**: Remove values outside the interquartile range (Q1 - 1.5×IQR, Q3 + 1.5×IQR).
  - **Visualization**: Use boxplots or scatter plots to detect outliers.
- **Example**: In a salary dataset, a few entries with extremely high values might be removed to normalize the distribution.

ALGORITHM:

Data Preprocessing Pipeline

Step 1: Load Dataset

- Import the pandas library.
- Read the dataset from the specified CSV file path into a DataFrame df.

Step 2: Attribute Selection

- Define a list of relevant features: ['age', 'income', 'education'].
- Create a new DataFrame df_selected containing only these selected columns.
- Display the first few rows of df_selected.

Step 3: Handle Missing Values

- Use dropna() to remove any rows in df that contain missing values.
- Store the cleaned data in df_clean.
- Display the cleaned DataFrame.

Step 4: Discretization (Binning Continuous Data)

- Define bin edges: [0, 18, 35, 60, 100].
- Define corresponding labels: ['Child', 'Youth', 'Adult', 'Senior'].
- Apply pd.cut() to the age column to categorize each value into an age group.
- Store the result in a new column age_group.
- Display the age and age_group columns.

Step 5: Elimination of Outliers

- Import stats from scipy and numpy as np.
- Compute the z-score for the income column.
- Filter out rows where the absolute z-score is greater than or equal to 3.
- Store the filtered data in df_no_outliers.
- Display the shape (rows, columns) of df_no_outliers.

PROGRAM:

```
# Exp. 2.1: Attribute Selection
import pandas as pd
df = pd.read_csv(r"E:\DE\data.csv")
# Select relevant features
selected_features = ['age', 'income', 'education']
df_selected = df[selected_features]
print(df_selected.head())
# Exp. 2.2: Remove missing values
df_clean = df.dropna()
print(df_clean)
# Exp. 2.3: Discretization (binning contiunous data)
bins = [0, 18, 35, 60, 100]
labels = ['Child', 'Youth', 'Adult', 'Senior']
df['age_group'] = pd.cut(df['age'], bins=bins, labels=labels)
print(df[['age', 'age_group']].head())
# Exp. 2.4: Elimination of outliers
from scipy import stats
import numpy as np
z_scores = np.abs(stats.zscore(df['income']))
df_no_outliers = df[(z_scores < 3)]
print(df_no_outliers.shape)
```

OUTPUT:

```
   age    income education
0  25.0   50000.0  Bachelor
1  30.0   60000.0    Master
2  22.0   45000.0  Bachelor
3  35.0   80000.0    Master
4  40.0  120000.0       PhD
   id   age    income  gender education
0   1  25.0   50000.0    Male  Bachelor
1   2  30.0   60000.0  Female    Master
2   3  22.0   45000.0  Female  Bachelor
3   4  35.0   80000.0    Male    Master
4   5  40.0  120000.0  Female       PhD
6   7  28.0   55000.0  Female  Bachelor
8   9  27.0   45000.0    Male  Bachelor
    age age_group
0  25.0     Youth
1  30.0     Youth
2  22.0     Youth
3  35.0     Youth
4  40.0     Adult
(0, 6)
```

RESULT: Hence, The python program was executed successfully.

EXP.NO : **EXP.DATE :**

KNN ALGORITHM FOR CLASSIFICATION AND REGRESSION:

AIM: To Write a Python Program On KNN algorithm for classification and regression.

DESCRIPTION:

What Is KNN?

K-Nearest Neighbors (KNN) is a **non-parametric**, **instance-based** learning algorithm. It makes predictions based on the **similarity** between data points, using **distance metrics** (usually Euclidean distance).

How KNN Works

1. **Choose a value for K** (number of neighbors).
2. **Calculate the distance** between the query point and all points in the training set.
3. **Select the K closest points** (neighbors).
4. **Make a prediction**:

- **Classification**: Use **majority voting** among neighbors.
- **Regression**: Use the **average** of neighbors' values.

KNN for Classification

Use Case:

Predicting a **category** or **label** (e.g., spam vs. not spam, disease vs. no disease).

Logic:

- Count the labels of the K nearest neighbors.
- Assign the label that appears **most frequently**.

Example:

If K=3 and the nearest neighbors have labels [Dog, Dog, Cat], the predicted label is **Dog**.

KNN for Regression

Use Case:

Predicting a **continuous value** (e.g., house price, temperature).

Logic:

- Take the **mean** (or weighted average) of the values of the K nearest neighbors.

Example:

If K=3 and the neighbors have values [100, 120, 130], the predicted value is:

\text{Prediction} = \frac{100 + 120 + 130}{3} = 116.67

ALGORITHM:

Part 1: KNN Classification (Iris Dataset)

Step 1: Import Required Libraries

- Import load_iris from sklearn.datasets.
- Import train_test_split from sklearn.model_selection.
- Import KNeighborsClassifier from sklearn.neighbors.
- Import accuracy_score from sklearn.metrics.

Step 2: Load Dataset

- Load the Iris dataset.
- Extract features X and target labels y.

Step 3: Split Dataset

- Split X and y into training and testing sets (80% train, 20% test).

Step 4: Create and Train Classifier

- Initialize KNeighborsClassifier with n_neighbors=3.
- Fit the classifier using training data.

Step 5: Predict and Evaluate

- Predict labels for the test set.
- Calculate accuracy using accuracy_score.
- Print the classification accuracy.

Part 2: KNN Regression (California Housing Dataset)

Step 1: Import Required Libraries

- Import fetch_california_housing from sklearn.datasets.
- Import train_test_split from sklearn.model_selection.
- Import KNeighborsRegressor from sklearn.neighbors.
- Import mean_squared_error from sklearn.metrics.

Step 2: Load Dataset

- Load the California housing dataset.
- Extract features X and target values y.

Step 3: Split Dataset

- Split X and y into training and testing sets (80% train, 20% test).

Step 4: Create and Train Regressor

- Initialize KNeighborsRegressor with n_neighbors=5.
- Fit the regressor using training data.

Step 5: Predict and Evaluate

- Predict target values for the test set.
- Calculate Mean Squared Error (MSE) using mean_squared_error.
- Print the regression MSE.

PROGRAM:

```
# Exp.3.1 KNN Classifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
# Load dataset
iris = load_iris()
X = iris.data
y = iris.target
# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
# Create and train KNN Classifier
knn_classifier = KNeighborsClassifier(n_neighbors=3)
knn_classifier.fit(X_train, y_train)
# Predict
y_pred = knn_classifier.predict(X_test)
# Evaluate
accuracy = accuracy_score(y_test, y_pred)
print(f"Classification Accuracy: {accuracy:.2f}")
# Exp.3.2 KNN Regression using the Boston housing dataset
# (or California housing, since Boston is deprecated in latest sklearn versions).
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsRegressor
```

```
from sklearn.metrics import mean_squared_error
# Load dataset
data = fetch_california_housing()
X = data.data
y = data.target
# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
# Create and train KNN Regressor
knn_regressor = KNeighborsRegressor(n_neighbors=5)
knn_regressor.fit(X_train, y_train)
# Predict
y_pred = knn_regressor.predict(X_test)
# Evaluate
mse = mean_squared_error(y_test, y_pred)
print(f"Regression MSE: {mse:.2f}")
```

OUTPUT:

3.1- Classification Accuracy: 1.00

3.2-Regression MSE: 1.12

RESULT: Hence, The python program was executed successfully.

EXP.NO : **EXP.DATE :**

DECISION TREE ALGORITHM FOR A CLASSIFICATION PROBLEM:

AIM: To write a python program do demonstrate decision tree algorithm for a classification problem and perform parameter tuning for better results.

DESCRIPTION:

Decision Tree for Classification

The goal of a decision tree for classification is to predict a categorical class label by recursively splitting the dataset into subsets based on the most significant features. The process generally works as follows:

1. **Start at the Root:** The tree begins as a single node, representing the entire training dataset.
2. **Feature Selection/Splitting**: The algorithm selects the best feature to split the data at a node. The "best" feature is determined by metrics that measure the impurity or homogeneity of the resulting subsets. Common metrics include:
    - Gini Impurity: Measures how often a randomly chosen element from the set would be incorrectly labeled if it was labeled according to the distribution of labels in the subset. A lower Gini impurity is better.
    - Information Gain (based on Entropy): Entropy is a measure of randomness or disorder. Information Gain is the reduction in entropy achieved by the split. A higher information gain is better.
3. **Recursive Splitting:** The node is split into two or more sub-nodes based on the value of the selected feature. This process is repeated recursively for each sub-node until one of the stopping conditions is met (e.g., all samples in a subset belong to the same class, the maximum depth is reached, or the number of samples in a node is too small).
4. **Leaf Nodes:** The final sub-nodes that do not split further are called leaf nodes, and they assign the predicted class label to new data based on the majority class of the training samples within that node.

Parameter Tuning (Hyperparameter Optimization)

Decision trees risk overfitting (performing well on training data but poorly on new data). Parameter tuning controls the tree's complexity to improve its performance on unseen data. Key hyperparameters to tune are:

1. max_depth: Limits the number of splits to prevent the tree from becoming too specific to the training data.
2. min_samples_split: Sets the minimum number of samples a node needs before it can be split. Increasing this value makes the tree simpler and more robust.
3. min_samples_leaf: Sets the minimum samples required in any terminal (leaf) node, which helps prune the tree.
4. criterion: Specifies the impurity measure ('gini' or 'entropy').
5. max_features: Restricts the number of features considered at each split, which can reduce complexity and improve generalization.

The tuning is typically performed using:

- **Grid Search:** Tests every combination of a predefined hyperparameter grid using cross-validation.
- **Randomized Search:** Samples and tests a fixed number of random combinations, often more efficient for large search spaces.

ALGORITHM:

Decision Tree with Grid Search Algorithm

Step 1: Setup & Data Prep: Import libraries, load the dataset (e.g., Iris), and split it into training and testing sets.

Step 2: Model & Grid Definition: Initialize the base DecisionTreeClassifier and create a param_grid dictionary listing specific hyperparameter values (e.g., max_depth, criterion) to test.

Step 3: Execute Grid Search: Apply and fit GridSearchCV to the training data. This object uses the base model, the parameter grid, and a chosen cross-validation strategy (e.g., 5-fold CV) to test every combination.

Step 4: Evaluate: Retrieve the best estimator (model with optimal parameters) from the search results, use it to make predictions on the test set, and finally, evaluate performance (e.g., using accuracy and a classification report).

Decision Tree with Random Search Algorithm

Step 1: Setup & Data Prep: Import libraries (e.g., RandomizedSearchCV, randint). Load the dataset (e.g., Iris) and split it into **training** and **testing** sets.

Step 2: Define Search:

- Initialize the **base Decision Tree model** (e.g., Regressor).
- Create a **param_dist** dictionary with statistical distributions (e.g., randint) for hyperparameters like max_depth, min_samples_split, and min_samples_leaf.

Step 3: Execute Search: Apply and **fit** RandomizedSearchCV on the training data, specifying the estimator, param_dist, number of iterations, and cross-validation folds (e.g., 5-fold CV).

Step 4: Evaluate: Retrieve and display the **best parameters** and the resulting **best score** achieved by the model during the search.

PROGRAM:

```
# EXP No.4 Decision tree algorithm for classifier and parameter tuning to check results

# Import necessary libraries

from sklearn.datasets import load_iris

from sklearn.tree import DecisionTreeClassifier

from sklearn.model_selection import train_test_split, GridSearchCV

from sklearn.metrics import classification_report, accuracy_score

import pandas as pd

# Load dataset

data = load_iris()

X = pd.DataFrame(data.data, columns=data.feature_names)
```

```
y = pd.Series(data.target)
# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
# Define the base model
dt = DecisionTreeClassifier(random_state=42)
# Define parameter grid for tuning
param_grid = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [2, 4, 6, 8, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}
# GridSearchCV for hyperparameter tuning
grid_search = GridSearchCV(estimator=dt, param_grid=param_grid,
                           cv=5, scoring='accuracy', n_jobs=-1, verbose=1)
# Fit the model
grid_search.fit(X_train, y_train)
# Best parameters and best estimator
print("Best Parameters:", grid_search.best_params_)
best_dt = grid_search.best_estimator_
# Predictions
y_pred = best_dt.predict(X_test)
# Evaluation
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

```
print("Accuracy Score:", accuracy_score(y_test, y_pred))
```

EXP No.4 Decision tree algorithm for classifier and parameter tuning using Ramdom search

```
# Import necessary libraries
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
import pandas as pd
# Define the parameter distribution to sample from
param_dist = {
    'max_depth': randint(1, 20),
    'min_samples_split': randint(2, 20),
    'min_samples_leaf': randint(1, 20)
}
dtree_reg = DecisionTreeRegressor(random_state=42)
random_search = RandomizedSearchCV(dtree_reg,
param_distributions=param_dist,
                                   n_iter=100, cv=5, random_state=42)
random_search.fit(X_train, y_train)
best_params_random = random_search.best_params_
best_score_random = random_search.best_score_
print(f"Best Parameters (Random Search): {best_params_random}")
print(f"Best Score (Random Search): {best_score_random}")
```

OUTPUT:

Decision Tree Classifier with GridSearchCV:

Fitting 5 folds for each of 90 candidates, totalling 450 fits

Best Parameters: {'criterion': 'entropy', 'max_depth': 4, 'min_samples_leaf': 1, 'min_samples_split': 2}

Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	10
1	1.00	1.00	1.00	9
2	1.00	1.00	1.00	11
accuracy			1.00	30
macro avg	1.00	1.00	1.00	30
weighted avg	1.00	1.00	1.00	30

Accuracy Score: 1.0

Decision Tree Regressor with RandomizedSearchCV:

Best Parameters (Random Search): {'max_depth': 6, 'min_samples_leaf': 2, 'min_samples_split': 3}

Best Score (Random Search): 0.93

RESULT: Hence, The python program was executed successfully.

EXP.NO : **EXP.DATE :**

DECISION TREE ALGORITHM FOR A REGRESSION PROBLEM:

AIM: To Write a Python Program to demonstrate decision tree algorithm for a regression problem.

DESCRIPTION:

A Decision Tree for a regression problem is an adaptation of the standard decision tree, designed to predict a continuous numerical value (like house price, salary, or temperature) instead of a categorical class label.

How Decision Trees Work for Regression

The core mechanism remains the same—recursively splitting the data—but the goal and the splitting criteria change:

1. The Goal: Predict a Value (Not a Class)

In classification, the leaf nodes represent a class label (e.g., 'Yes' or 'No'). In regression, the leaf nodes represent a predicted numerical value, which is typically the average (or mean) of the target variable values for all training data points that fall into that leaf.

2. Splitting Criteria: Minimizing Variance/Error

Since the goal is to predict a continuous value, the algorithm must choose splits that minimize the error in prediction. Instead of using impurity measures like Gini or Entropy (which are for categories), regression trees use metrics that measure the homogeneity of the target values within a potential split:

- Reduction in Variance: This is the most common criterion. The algorithm calculates the variance of the target variable in the current node and in the potential child nodes. It chooses the split that results in the greatest reduction in the overall variance of the target variable across the resulting subsets. Minimizing variance means the values within each new node are closer to their mean, leading to a more precise prediction.
- Mean Squared Error (MSE) / Sum of Squared Errors (SSE): The algorithm may also directly aim to minimize the MSE. The split that minimizes the sum of squared differences between the actual target values and the predicted mean value within each child node is selected.

3. The Prediction

Once the tree is built and a new data point is passed through it:

1. It traverses the tree, following the decision rules until it reaches a leaf node.
2. The final prediction for that new data point is the mean of the target values of all training samples contained in that specific leaf node.

ALGORITHM:

Decision Tree for Regression:

Step 1: Import Required Libraries

- Import necessary modules from sklearn, numpy, pandas, and matplotlib.

Step 2: Generate Synthetic Regression Data

- Use make_regression() to create a dataset with:
  - 200 samples
  - 1 feature
  - Noise level of 15
  - Fixed random_state for reproducibility

Step 3: Split the Dataset

- Use train_test_split() to divide data into:
  - Training set (80%)
  - Testing set (20%)
  - Set random_state for consistency

Step 4: Initialize and Train the Model

- Create a DecisionTreeRegressor with:
  - max_depth=4 to control tree complexity
  - random_state=42 for reproducibility
- Fit the model on training data using .fit()

Step 5: Make Predictions

- Use .predict() to generate predictions on the test set

Step 6: Evaluate the Model

- Calculate:
  - *Mean Squared Error (MSE)* using mean_squared_error()
  - *R^2 Score* using r2_score()
- Print both metrics to assess model performance

Step 7: Visualize Predictions

- Create a grid of input values using np.linspace()
- Predict target values for this grid
- Plot:
  - Training data (light blue)
  - Test data (orange)
  - Model predictions (red line)
- Add labels, title, legend, and grid for clarity

Step 8: Visualize the Decision Tree Structure (Optional)

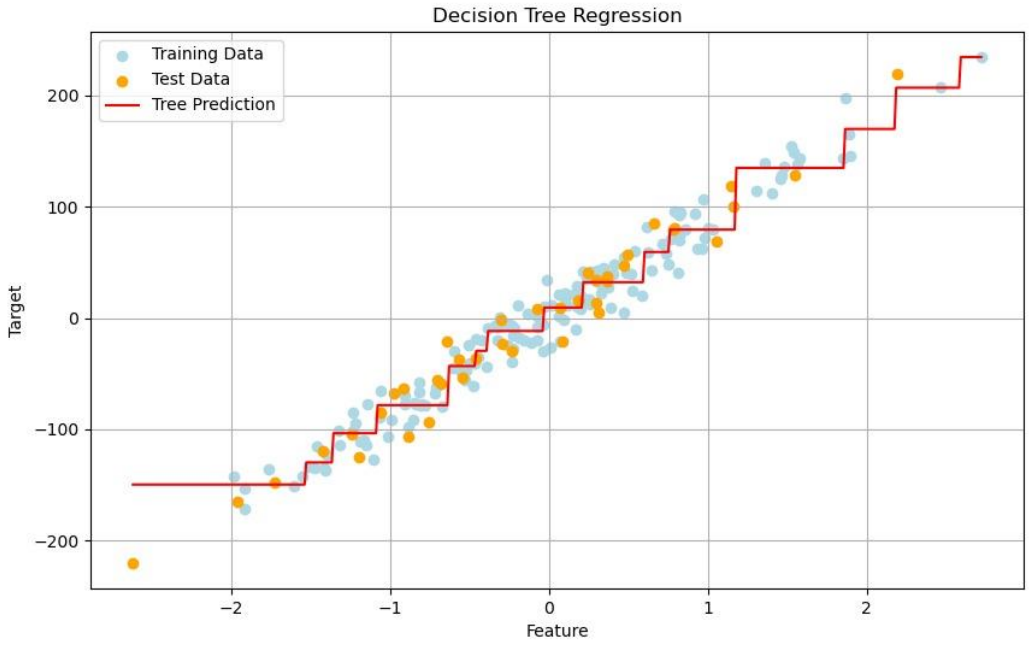
- Use plot_tree() to display the trained tree
- Customize with:
  - filled=True for color-coded nodes
  - feature_names=["Feature"] for labeling
  - rounded=True for aesthetics

PROGRAM:

```
# EXP No.5 Decision tree algorithm for regression problem
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import classification_report, accuracy_score
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor, plot_tree
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.datasets import make_regression
# 1. Generate synthetic regression data
X, y = make_regression(n_samples=200, n_features=1, noise=15,
random_state=42)
# 2. Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
# 3. Create and train the Decision Tree Regressor
regressor = DecisionTreeRegressor(max_depth=4, random_state=42)
regressor.fit(X_train, y_train)
# 4. Predict on test set
y_pred = regressor.predict(X_test)
# 5. Evaluate
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
```

```
print(f"Mean Squared Error: {mse:.2f}")
print(f"R² Score: {r2:.2f}")
# 6. Visualization
# Plot predictions vs ground truth
X_grid = np.linspace(X.min(), X.max(), 500).reshape(-1, 1)
y_grid_pred = regressor.predict(X_grid)
plt.figure(figsize=(10, 6))
plt.scatter(X_train, y_train, color="lightblue", label="Training Data")
plt.scatter(X_test, y_test, color="orange", label="Test Data")
plt.plot(X_grid, y_grid_pred, color="red", label="Tree Prediction")
plt.title("Decision Tree Regression")
plt.xlabel("Feature")
plt.ylabel("Target")
plt.legend()
plt.grid(True)
plt.show()
# 7. Optional: Visualize the Tree
plt.figure(figsize=(12, 6))
plot_tree(regressor, filled=True, feature_names=["Feature"], rounded=True)
plt.title("Decision Tree Structure")
plt.show()
```

OUTPUT:



Mean Squared Error: 458.57

R^2 Score: 0.94

RESULT: Hence, The python program was executed successfully.

EXP.NO : **EXP.DATE :**

RANDOM FOREST ALGORITHM FOR CLASSIFICATION AND REGRESSION:

AIM: To Write a Python Program to apply Random Forest algorithm for classification and regression.

DESCRIPTION:

What Is Random Forest?

Random Forest is an ensemble learning method that builds multiple decision trees and combines their outputs:

- For classification, it uses majority voting.
- For regression, it uses the average of predictions.

It reduces overfitting and improves accuracy by introducing randomness in both data sampling and feature selection.

Summary: Random Forest in Python

- **Classification**: Use RandomForestClassifier for tasks like predicting species, spam detection, or customer churn.
- **Regression**: Use RandomForestRegressor for predicting continuous values like house prices, sales forecasts, or temperature.

Both models follow the same workflow:

1. Load and preprocess data
2. Split into training and test sets
3. Fit the model
4. Predict and evaluate

When to Use Random Forest

- When you want **high accuracy** and **robustness to overfitting**
- When your data has **non-linear relationships**
- When you need **feature importance** insights

ALGORITHM:

Random Forest for Classification (Exp 6a)*

Step 1: Import Required Libraries

- Import modules from sklearn for dataset, model, splitting, and evaluation.

Step 2: Load Dataset

- Load the *Iris dataset* using load_iris().

- Extract features X and target labels y.

Step 3: Split Dataset

- Use train_test_split() to divide data into:

  - Training set (70%)

  - Testing set (30%)

  - Set random_state=42 for reproducibility

Step 4: Initialize Random Forest Classifier

- Create a RandomForestClassifier object with:

  - n_estimators=100 (number of trees)

  - random_state=42

Step 5: Train the Model

- Fit the classifier on training data using .fit(X_train, y_train)

Step 6: Make Predictions

- Use .predict(X_test) to generate predictions on the test set

Step 7: Evaluate the Model

- Calculate *Accuracy Score* using accuracy_score(y_test, y_pred)

- Print the classification accuracy

PROGRAM:

```
# Exp 6(a): Random forest algorithm for classification
# Library files
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
#Load dataset
data = load_iris()
X, y = data.data, data.target
# Split into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)
# Create and train model
clf = RandomForestClassifier(n_estimators=100, random_state=42)
clf.fit(X_train, y_train)
# Predict and evaluate
y_pred = clf.predict(X_test)
print("Classification Accuracy:", accuracy_score(y_test, y_pred))


# Exp 6(b): Random forest algorithm for regression
# Python libraries
from sklearn.datasets import load_boston
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
```

```
# Load dataset
# Note: `load_boston` is deprecated in newer versions of sklearn; consider using another dataset if needed
data = load_boston()
X, y = data.data, data.target
# Split into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
# Create and train model
reg = RandomForestRegressor(n_estimators=100, random_state=42)
reg.fit(X_train, y_train)
# Predict and evaluate
y_pred = reg.predict(X_test)
print("Regression MSE:", mean_squared_error(y_test, y_pred))
```

OUTPUT:

#6(a)

Classification Accuracy: 1.0

#6(b)

RESULT: Hence, The python program was executed successfully.

EXP.NO : **EXP.DATE :**

NAÏVE BAYES CLASSIFICATION ALGORITHM:

AIM: To Write a Python Program to Demonstrate Naïve Bayes Classification algorithm.

DESCRIPTION:

What Is Naïve Bayes?

Naïve Bayes is a **probabilistic classification algorithm** based on **Bayes' Theorem**. It's called "naïve" because it assumes that all features are **independent** of each other — a simplification that often works surprisingly well in practice.

How It Works

1. **Calculate Prior**: Estimate the probability of each class from training data.
2. **Calculate Likelihood**: For each feature, compute the probability of that feature given the class.
3. **Apply Bayes' Theorem**: Combine prior and likelihood to compute posterior probability.
4. **Predict**: Choose the class with the highest posterior probability.

Advantages

- Fast and simple
- Works well with high-dimensional data
- Great for text classification (spam detection, sentiment analysis)

Limitations

- Assumes feature independence (often violated)
- Struggles with correlated features

ALGORITHM:

Naive Bayes Classification on Iris Dataset

1. **Import Required Libraries**
    - Import dataset loader, model selection tools, classifier, and evaluation metrics from sklearn.
2. **Load Dataset**
    - Load the Iris dataset using load_iris().
    - Extract features X and target labels y.
3. **Split Dataset**
    - Use train_test_split() to divide the dataset into:
        - Training set (70%)
        - Testing set (30%)
    - Set random_state=42 for reproducibility.
4. **Initialize Classifier**
    - Create a Gaussian Naive Bayes classifier object using GaussianNB().
5. **Train the Model**
    - Fit the classifier on the training data: model.fit(X_train, y_train).
6. **Make Predictions**
    - Predict the labels for the test data: y_pred = model.predict(X_test).
7. **Evaluate the Model**
    - Calculate the accuracy using accuracy_score(y_test, y_pred).
8. **Display Results**

- Print the classification accuracy.

PROGRAM:

```
# Exp 7. NAive Base classifier

# Library files

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.naive_bayes import GaussianNB

from sklearn.metrics import accuracy_score

# Load the Iris dataset

data = load_iris()

X, y = data.data, data.target

# Split into training and test data

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

# Create the Naive Bayes classifier

model = GaussianNB()

# Train the model

model.fit(X_train, y_train)

# Make predictions

y_pred = model.predict(X_test)

# Evaluate the model

accuracy = accuracy_score(y_test, y_pred)

print("Naive Bayes Classification Accuracy:", accuracy)
```

OUTPUT:

```
Naive Bayes Classification Accuracy: 0.9777777777777777
```

RESULT: Hence, The python program was executed successfully.

EXP.NO : **EXP.DATE :**

SUPPORT VECTOR ALGORITHM FOR CLASSIFICATION:

AIM: To Write a Python Program to Apply Support Vector algorithm for classification.

DESCRIPTION:

What Is SVM?

Support Vector Machine is a supervised machine learning algorithm used for classification and regression. It works by finding the optimal hyperplane that separates data points of different classes with the maximum margin.

Key Concepts

- Hyperplane: The decision boundary that separates classes. In 2D it's a line, in 3D it's a plane, and in higher dimensions it's a hyperplane.

- Support Vectors: Data points closest to the hyperplane. These are critical in defining the margin.

- Margin: Distance between the hyperplane and the nearest support vectors. SVM aims to maximize this.

- Kernel Trick: Transforms non-linearly separable data into higher dimensions to make it linearly separable.

Real-World Use Cases of SVM

SVMs are widely used in domains where **high accuracy and clear decision boundaries** are critical:

- **Text classification**: Spam detection, sentiment analysis
- **Image recognition**: Handwritten digit classification (e.g., MNIST)
- **Bioinformatics**: Cancer detection using gene expression data
- **Finance**: Fraud detection, credit scoring

ALGORITHM:

SVM Classification on Iris Dataset (Binary, 2D)

1. **Import Required Libraries**
    - Import NumPy, Matplotlib, and relevant modules from sklearn.
2. **Load Dataset**
    - Load the Iris dataset using datasets.load_iris().
    - Extract only the first two features (sepal length and sepal width) for 2D visualization.
    - Extract target labels.
3. **Filter for Binary Classification**
    - Remove class 2 (Iris-Virginica) to retain only:
    - Class 0: Iris-Setosa
    - Class 1: Iris-Versicolor
4. **Split Dataset**
    - Use train_test_split() to divide the filtered dataset into:
        - Training set (70%)
        - Testing set (30%)
    - Set random_state=0 for reproducibility.
5. **Initialize and Train SVM Model**
    - Create an SVM classifier using a linear kernel: SVC(kernel='linear').
    - Fit the model on the training data.
6. **Make Predictions**
    - Predict labels for the test data using model.predict().
7. **Evaluate Model**
    - Calculate and print the accuracy using accuracy_score().
8. **Visualize Decision Boundary**

- Define a function plot_decision_boundary():
- Create a mesh grid over the feature space.
- Predict class labels for each point in the grid.
- Plot decision boundary using contourf().
- Overlay training data using scatter().

9. **Display Plot**

- Call plot_decision_boundary() with trained model and full dataset.

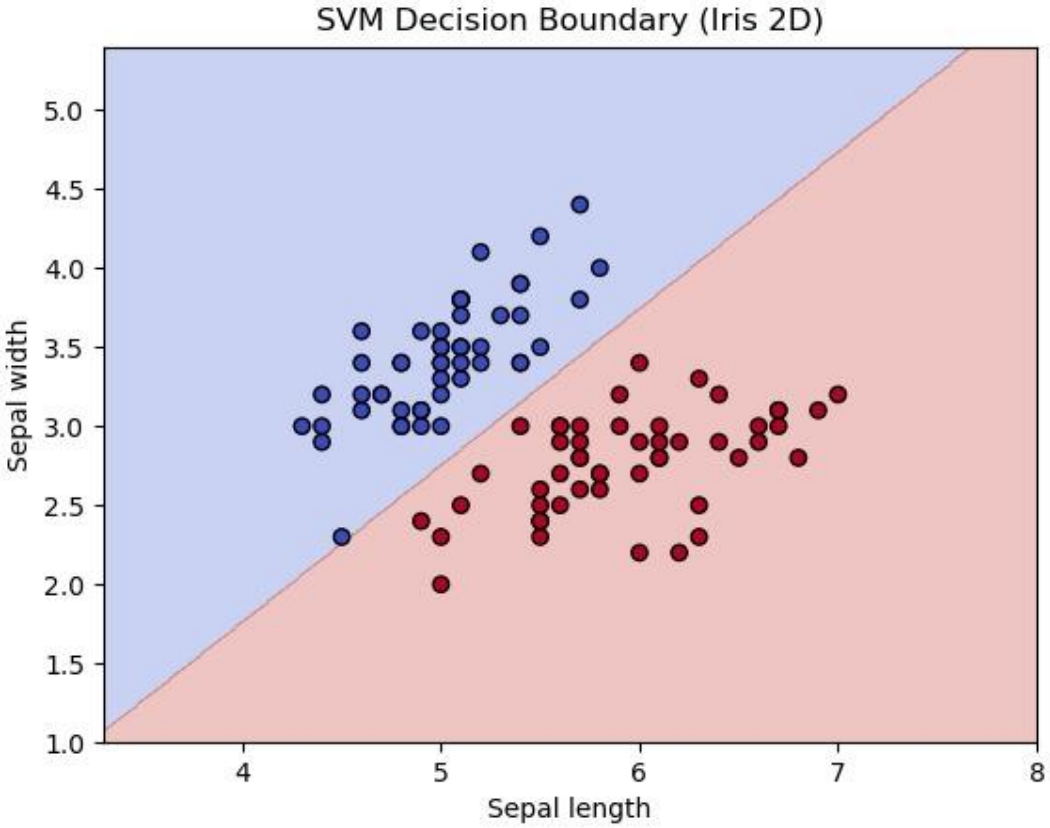
PROGRAM:

```
# Exp 8. Support Vector Algorithm for Classification
# Library files
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
# Load the Iris dataset and select only 2 classes for binary classification
iris = datasets.load_iris()
X = iris.data[:, :2]  # Use only the first 2 features for 2D plot
y = iris.target
# Only keep class 0 and 1 (Setosa and Versicolor)
X = X[y != 2]
y = y[y != 2]
# Split the dataset
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=0)
# Train the SVM classifier
model = SVC(kernel='linear')
model.fit(X_train, y_train)
# Predict and print accuracy
y_pred = model.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))


# Plot decision boundary
def plot_decision_boundary(clf, X, y):
    # Create a mesh grid
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 500),
                         np.linspace(y_min, y_max, 500))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    # Plot the contour and training examples
    plt.contourf(xx, yy, Z, alpha=0.3, cmap=plt.cm.coolwarm)
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.coolwarm, edgecolors='k')
    plt.xlabel('Sepal length')
    plt.ylabel('Sepal width')
    plt.title('SVM Decision Boundary (Iris 2D)')
    plt.show()
    plot_decision_boundary(model, X, y)
```

OUTPUT:



Accuracy: 1.0

RESULT: Hence, The python program was executed successfully.

EXP.NO : **EXP.DATE :**

SIMPLE LINEAR REGRESSION:

AIM: To Write a Python Program to Demonstrate simple linear regression algorithm for a regression problem.

DESCRIPTION:

What Is Simple Linear Regression?

Simple Linear Regression (SLR) predicts a target variable y using one feature x by fitting a line of the form:

y = \beta_0 + \beta_1 x

- **\beta_0** is the intercept (where the line crosses the y-axis).
- **\beta_1** is the slope (how much y changes for a unit change in x).

The goal is to find the best-fitting line that minimizes the error between predicted and actual values, typically using **Least Squares Method**.

Advantages of Simple Linear Regression

- **Easy to Understand and Implement**: The math is straightforward, making it ideal for beginners.
- **Interpretable Coefficients**: You can clearly explain the impact of the independent variable on the dependent variable.
- **Fast to Train**: Computationally efficient even on large datasets.
- **Good Baseline Model**: Often used as a benchmark before trying more complex models.

Limitations

- **Assumes Linearity**: It only works well when the relationship between variables is linear.
- **Sensitive to Outliers**: Outliers can significantly skew the regression line.
- **Single Feature Only**: It cannot handle multiple independent variables (use Multiple Linear Regression instead).
- **Assumes Homoscedasticity**: The variance of residuals should be constant across all levels of the independent variable.

ALGORITHM:

Simple Linear Regression for Synthetic Data

Step 1: Import Required Libraries

- Import NumPy for numerical operations.
- Import Matplotlib for visualization.
- Import LinearRegression from sklearn.linear_model.
- Import train_test_split from sklearn.model_selection.
- Import evaluation metrics: mean_squared_error, r2_score.

Step 2: Generate Synthetic Linear Data

- Set a random seed for reproducibility.
- Generate 100 random values for feature X in the range [0, 2].
- Create target variable y using the linear equation:

where noise is drawn from a standard normal distribution.

Step 3: Split Data into Training and Testing Sets

- Use train_test_split to divide X and y into:
    - 80% training data
    - 20% testing data
- Set random_state=42 for consistent splits.

Step 4: Train the Linear Regression Model

- Initialize a LinearRegression model.
- Fit the model using training data (X_train, y_train).

Step 5: Make Predictions

- Use the trained model to predict y values for X_test.

Step 6: Evaluate the Model

- Print the learned coefficient (slope) and intercept.

- Calculate and print:
- Mean Squared Error (MSE)
- R^2 Score (coefficient of determination)

Step 7: Visualize the Results

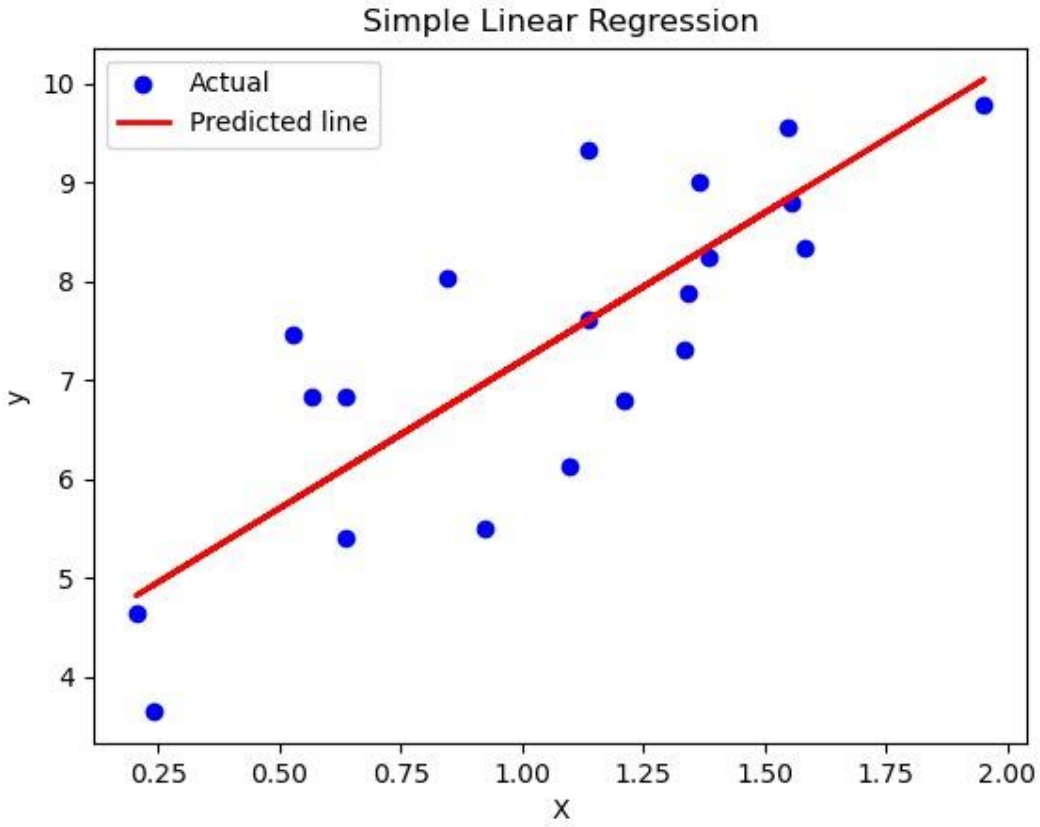
- Create a scatter plot of actual test data (X_test, y_test).
- Overlay the predicted regression line (X_test, y_pred).
- Add labels, title, legend, and display the plot.

PROGRAM:

```
# Exp 9. Simple Linear Regression Algorithm for Regression Problem
# Library files
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
# 1. Generate synthetic linear data
np.random.seed(0)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)  # y = 4 + 3x + noise
# 2. Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
# 3. Create and train the model
model = LinearRegression()
model.fit(X_train, y_train)
```

```
# 4. Predict
y_pred = model.predict(X_test)
# 5. Evaluate
print("Coefficient (slope):", model.coef_[0][0])
print("Intercept:", model.intercept_[0])
print("Mean squared error (MSE):", mean_squared_error(y_test, y_pred))
print("R² score:", r2_score(y_test, y_pred))
# 6. Plot
plt.scatter(X_test, y_test, color='blue', label='Actual')
plt.plot(X_test, y_pred, color='red', linewidth=2, label='Predicted line')
plt.xlabel("X")
plt.ylabel("y")
plt.title("Simple Linear Regression")
plt.legend()
plt.show()
```

OUTPUT:



Coefficient (slope): 2.99025910100489

Intercept: 4.206340188711437

Mean squared error (MSE): 0.9177532469714291

R^2 score: 0.6521157503858556

RESULT: Hence, The python program was executed successfully.

EXP.NO : **EXP.DATE :**

LOGISTIC REGRESSION:

AIM: To Write a Python Program to Apply Logistic regression algorithm for a classification problem.

DESCRIPTION:

What Is Logistic Regression?

Unlike linear regression which predicts continuous values, **logistic regression** estimates the probability of a categorical outcome. It's most commonly used for **binary classification**, such as:

- Spam vs. Not Spam
- Disease vs. No Disease
- Purchase vs. No Purchase

The model uses the **sigmoid function** to map any real-valued number into a range between 0 and 1:

\sigma(z) = \frac{1}{1 + e^{-z}}

Where z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n

Key Features

- **Probability Output**: Instead of raw class labels, it gives probabilities.
- **Decision Boundary**: Typically 0.5 — if probability > 0.5, classify as 1.
- **Interpretability**: Coefficients show the influence of each feature.

Real-World Impact

Logistic Regression is widely used because it's:

- **Fast and interpretable**: Great for baseline models and feature importance.
- **Robust for binary classification**: Especially when the relationship between features and outcome is roughly linear in log-odds.
- **HR**: Predicting employee attrition.

ALGORITHM:

Logistic Regression for Binary Classification

Step 1: Import Required Libraries

- Import NumPy for numerical operations.
- Import scikit-learn modules for:
- Dataset loading
- Model training, Evaluation metrics

Step 2: Load the Dataset

- Use load_breast_cancer() from sklearn.datasets to load features X and target labels y.

Step 3: Split the Dataset

- Use train_test_split() to divide data into:
    - **Training set** (70%)
    - **Testing set** (30%)
- Set random_state=0 for reproducibility.

Step 4: Train the Logistic Regression Model

- Initialize LogisticRegression() with max_iter=10000 to ensure convergence.
- Fit the model using X_train and y_train.

Step 5: Make Predictions

- Use model.predict(X_test) to generate predicted labels y_pred.

Step 6: Evaluate the Model

- Calculate and print:
- **Accuracy Score**: Overall correctness of predictions.
- **Confusion Matrix**: Breakdown of true positives, false positives, true negatives, and false negatives.
- **Classification Report**: Includes precision, recall, F1-score for each class.

PROGRAM:

```
# Exp 10. Logistic Regression Algorithm for Classification Problem

# Library files

import numpy as np

from sklearn.datasets import load_breast_cancer

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report

# 1. Load dataset

data = load_breast_cancer()

X = data.data

y = data.target

# 2. Split into training and test sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=0)

# 3. Train Logistic Regression model

model = LogisticRegression(max_iter=10000)  # Increased iterations for
convergence

model.fit(X_train, y_train)

# 4. Predict

y_pred = model.predict(X_test)

# 5. Evaluate

print("Accuracy:", accuracy_score(y_test, y_pred))

print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))

print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

OUTPUT:

Accuracy: 0.9590643274853801

Confusion Matrix:

[[62 1]

[6 102]]

Classification Report:

	precision	recall	f1-score	support
0	0.91	0.98	0.95	63
1	0.99	0.94	0.97	108
accuracy			0.96	171
macro avg	0.95	0.96	0.96	171
weighted avg	0.96	0.96	0.96	171

RESULT: Hence, The python program was executed successfully.

EXP.NO : **EXP.DATE :**

MULTI-LAYER PERCEPTRON:

AIM: To Write a Python Program to Demonstrate Multi-layer Perceptron algorithm for a classification problem.

DESCRIPTION:

What Is a Multi-layer Perceptron?

An MLP is a supervised learning algorithm that maps input features to output classes using multiple layers of neurons. Each neuron applies a weighted sum followed by a non-linear activation function (like ReLU or sigmoid). The network learns by adjusting weights to minimize classification error.

- Structure:
  - **Input Layer**: Receives feature vectors.
  - **Hidden Layers**: Perform transformations using weights and activation functions.
  - **Output Layer**: Produces class probabilities or labels.

Key Concepts

- **Backpropagation**: Algorithm used to update weights by minimizing loss.
- **Activation Functions**: Introduce non-linearity (e.g., ReLU, sigmoid).
- **Epochs & Iterations**: Control how long the model trains.

Limitations

- **Requires tuning**: Performance depends heavily on hyperparameters like learning rate, number of layers, and activation functions.
- **Computational cost**: Training can be slow for large datasets or deep networks.
- **Black-box nature**: Harder to interpret compared to simpler models like logistic regression.

ALGORITHM:

Multilayer Perceptron for Classification

Step 1: Import Required Libraries

- Import dataset loader, model training tools, and evaluation metrics from scikit-learn.

Step 2: Load the Dataset

- Load the Iris dataset using load_iris().
- Extract features X and target labels y.

Step 3: Split the Dataset

- Use train_test_split() to divide the data into:
    - **Training set** (70%)
    - **Testing set** (30%)
- Set random_state=0 for reproducibility.

Step 4: Initialize and Train the MLP Model

- Create an instance of MLPClassifier with:
    - One hidden layer containing 10 neurons
    - Maximum iterations set to 1000
    - random_state=1 for consistent results
- Fit the model using X_train and y_train.

Step 5: Make Predictions

- Use mlp.predict(X_test) to generate predicted labels y_pred.

Step 6: Evaluate the Model

- Calculate and print:
- **Accuracy Score**: Percentage of correct predictions
- **Classification Report**: Includes precision, recall, F1-score for each clas

PROGRAM:

```
# Exp 11. Multilayer Perceptron (MLP) Algorithm for Classification Problem

# Library files

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.neural_network import MLPClassifier

from sklearn.metrics import accuracy_score, classification_report

# 1. Load dataset

iris = load_iris()

X = iris.data

y = iris.target

# 2. Split dataset

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=0)

# 3. Create and train MLP model

mlp = MLPClassifier(hidden_layer_sizes=(10,), max_iter=1000,
random_state=1)

mlp.fit(X_train, y_train)

# 4. Predict

y_pred = mlp.predict(X_test)

# 5. Evaluate

print("Accuracy:", accuracy_score(y_test, y_pred))

print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

OUTPUT:

Accuracy: 0.9555555555555556

Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	16
1	1.00	0.89	0.94	18
2	0.85	1.00	0.92	11
accuracy			0.96	45
macro avg	0.95	0.96	0.95	45
weighted avg	0.96	0.96	0.96	45

RESULT: Hence, The python program was executed successfully.