KATHIR COLLEGE of ENGINEERING

"WISDOM TREE", NEELAMBUR, COIMBATORE – 641 062.

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

Laboratory Record

AD3411 Data Science and Analytics Laboratory

Name :

Reg.No :

Laboratory :

Semester :

Year :

KATHIR COLLEGE OF ENGINEERING

"WISDOM TREE", NEELAMBUR, COIMBATORE – 641 062.

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

Bonafide Certificate

This is to certify that the record word for **AD3411 Data Science and Analytics Laboratory** bonafide record of work done by Mr./Ms..... Register Number for the course B.E/B.Tech – Artificial Intelligence and Data Science during ..... semester of academic year 2023-2024.

Staff in-Charge **HOD**

Submitted for the Practical Examination held on .....

Internal Examiner **External Examiner**

TABLE OF CONTENTS

Ex.No 1(a)	WORKING WITH NUMPY ARRAYS
Date:	

AIM:

To demonstrate the creation of a NumPy array, and to print its various attributes including the type, number of dimensions, shape, size, and the data type of its elements

ALGORITHM:

Step 1: Start

Step 2: Import NumPy module

Step 3: Create a NumPy Array: arr = np.array([[1, 2, 3], [4, 2, 5]]) creates a 2D array.

Step 4: Print Array Attributes:

type(arr): Prints the type of the array object.

arr.ndim: Prints the number of dimensions.

arr.shape: Prints the shape of the array.

arr.size: Prints the total number of elements.

arr.dtype: Prints the data type of the array elements.

PROGRAM:

```
import numpy as np
# Creating array object
arr = np.array([[1, 2, 3], [4, 2, 5]])
# Printing type of arr object
print("Array is of type:", type(arr))
# Printing array dimensions (axes)
print("No. of dimensions:", arr.ndim)
# Printing shape of array
print("Shape of array:", arr.shape)
# Printing size (total number of elements) of array
```

```
print("Size of array:", arr.size)
# Printing type of elements in array
print("Array stores elements of type:", arr.dtype)
```

OUTPUT:

```
Array is of type: <class 'numpy.ndarray'>
No. of dimensions: 2
Shape of array: (2, 3)
Size of array: 6
Array stores elements of type: int64
```

PROGRAM TO PERFORM ARRAY SLICING

AIM:

To demonstrate the creation of a NumPy array, print the array, and then perform slicing operations on the array to extract and print a subset of its elements.

ALGORITHM:

STEP 1: Import the NumPy library which provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.

STEP 2: Create a NumPy array using the np.array function. In this case, we are creating a 3x3 array with the values [[1, 2, 3], [3, 4, 5], [4, 5, 6]].

STEP 3: Print the original array to display its contents.

STEP 4: Perform a slicing operation on the array to extract a subset of its elements. In this case, we slice the array to obtain all rows starting from the second row.

STEP 5: Print the sliced array to display the extracted subset of elements.

PROGRAM:

```
a = np.array([[1,2,3],[3,4,5],[4,5,6]])
print(a)
print("After slicing") print(a[1:])
```

OUTPUT:

[[1 2 3]

[3 4 5]

[4 5 6]]

After slicing

[[3 4 5]

[4 5 6]]

PROGRAM TO PERFORM ARRAY SLICING

AIM:

To demonstrate the creation of a NumPy array and how to perform various slicing operations to extract specific elements, rows, and columns from the array.

ALGORITHM:

STEP 1: Import the NumPy library which provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.

STEP 2: Create a NumPy array using the np.array function. In this case, we are creating a 2D array with the values [[1, 2, 3], [3, 4, 5], [4, 5, 6]].

STEP 3: Print the original array to display its contents.

STEP 4: Extract and print all items from the second column using array slicing.

STEP 5: Extract and print all items from the second row using array slicing.

STEP 6: Extract and print all items from column 1 onwards using array slicing.

PROGRAM:

```
import numpy as np
# Create a NumPy array
a = np.array([[1, 2, 3], [3, 4, 5], [4, 5, 6]])
# Print the original array
print('Our array is:')
print(a)
```

```
print('\n')
# Extract and print items in the second column
print('The items in the second column are:')
print(a[..., 1])
print('\n')
# Extract and print items in the second row
print('The items in the second row are:')
print(a[1, ...])
print('\n')
# Extract and print items from column 1 onwards
print('The items from column 1 onwards are:')
print(a[..., 1:])
```

OUTPUT

```
Our array is: [[1 2 3]
[3 4 5]
[4 5 6]]
The items in the second column are:
[2 4 5]
The items in the second row are:
[3 4 5]
The items column 1 onwards are:
[[2 3]
[4 5]
[5 6]]
```

RESULT:

Thus, working with NumPy arrays was successfully completed.

Ex. No: 1(b)	CREATE A DATAFRAME USING A LIST OF ELEMENTS
Date:	

AIM:

Demonstrate the creation and manipulation of Pandas DataFrames from different input sources: NumPy arrays, dictionaries, other DataFrames, and Series. Also, print various DataFrame attributes such as shape and length.

ALGORITHM:

STEP 1: import numpy as np and import pandas as pd import the required libraries.

STEP 2: data is a 2D array with row and column labels.

df1 is created using a subset of data.

print(df1) displays the DataFrame.

STEP 3: my_2darray is a simple 2D array.

df2 is created directly from my_2darray.

print(df2) displays the DataFrame.

STEP 4: my_dict is a dictionary with columns as keys and lists of data as values.

df3 is created from my_dict.

print(df3) displays the DataFrame.

STEP 5: my_df is a DataFrame created with a single column 'A'.

df4 is created from my_df.

print(df4) displays the DataFrame.

STEP 6: my_series is a Series with country names and their capitals.

df5 is created from my_series.

print(df5) displays the DataFrame.

STEP 7: df6 is a DataFrame created from a 2D array.

print(df6.shape) prints the shape of the DataFrame.

print(len(df6.index)) prints the number of rows.

PROGRAM:

```
import numpy as np
import pandas as pd
# Create DataFrame from a 2D array
data = np.array([['', 'Col1', 'Col2'], ['Row1', 1, 2], ['Row2', 3, 4]])
df1 = pd.DataFrame(data=data[1:, 1:], index=data[1:, 0], columns=data[0, 1:])
print(df1)
# Create DataFrame from a simple 2D array
my_2darray = np.array([[1, 2, 3], [4, 5, 6]])
df2 = pd.DataFrame(my_2darray)
print(df2)
# Create DataFrame from a dictionary
my_dict = {1: ['1', '3'], 2: ['1', '2'], 3: ['2', '4']}
df3 = pd.DataFrame(my_dict)
print(df3)
# Create DataFrame from another DataFrame
my_df = pd.DataFrame(data=[4, 5, 6, 7], index=range(0, 4), columns=['A'])
df4 = pd.DataFrame(my_df)
print(df4)
# Create DataFrame from a Series
my_series = pd.Series({
   "United Kingdom": "London",
   "India": "New Delhi",
   "United States": "Washington",
   "Belgium": "Brussels"
})
df5 = pd.DataFrame(my_series, columns=['Capital'])
print(df5)
```

```
# Create DataFrame and print its shape and length
df6 = pd.DataFrame(np.array([[1, 2, 3], [4, 5, 6]]))
print(df6.shape)
print(len(df6.index))
```

OUTPUT:

```
      Col1 Col2
Row1     1       2
Row2     3       4
  0  1  2
0  1  2  3
1  4  5  6
  1  2  3
0  1  1  2
1  3  2  4
  A
0  4
1  5
2  6
3  7
              Capital
United Kingdom       London
India           New Delhi
United States  Washington
Belgium         Brussels
(2, 3)
2
```

RESULT:

Thus the working with Pandas data frames was successfully completed.

Ex.No: 2	BASIC PLOTS USING MATPLOTLIB
Date	

AIM:

To create a simple line plot using Matplotlib, displaying a set of points and labeling the axes and the graph title.

ALGORITHM:

STEP 1: Import matplotlib.pyplot as plt imports the pyplot module from Matplotlib, allowing access to its plotting functions.

STEP 2: x = [1, 2, 3] defines the x-axis values.

y = [2, 4, 1] defines the y-axis values.

STEP 3: plt.plot(x, y) plots the data points and connects them with a line.

STEP 4: plt.xlabel('x - axis') sets the label for the x-axis.

plt.ylabel('y - axis') sets the label for the y-axis.

STEP 5: plt.title('My first graph!') sets the title of the graph.

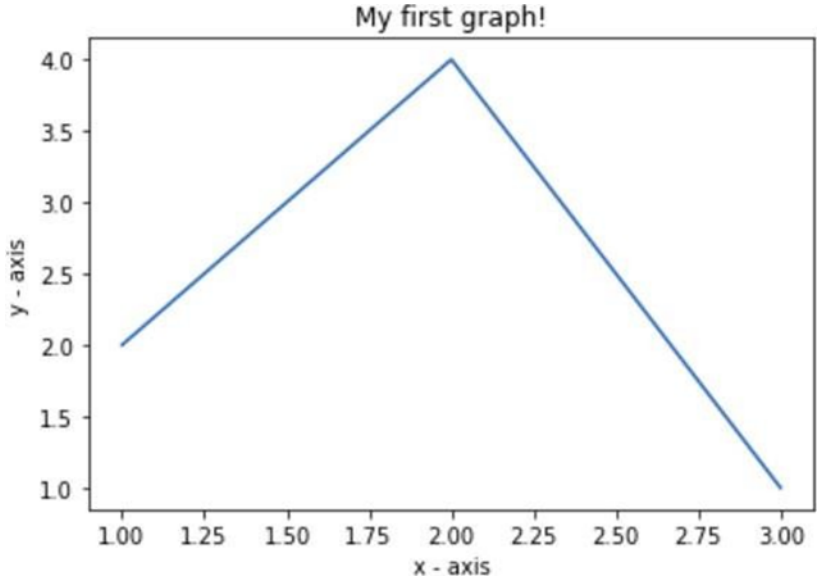
STEP 6: plt.show() displays the plot in a window.

PROGRAM:

```
# Importing the required module
import matplotlib.pyplot as plt
# x axis values
x = [1, 2, 3]
# corresponding y axis values
y = [2, 4, 1]
# plotting the points
plt.plot(x, y)
# naming the x axis
plt.xlabel('x - axis')
```

```
# naming the y axis
plt.ylabel('y - axis')
# giving a title to the graph
plt.title('My first graph!')
# function to show the plot
plt.show()
```

OUTPUT:



RESULT:

The x-axis will be labeled as "x - axis", the y-axis as "y - axis", and the graph will have the title "My first graph!". The plot will show a line connecting the points (1, 2), (2, 4), and (3, 1).

Ex.No: 2(b)	BASIC PLOTS USING MATPLOTLIB
Date:	

AIM:

To demonstrate advanced features of Matplotlib for creating plots. This includes plotting multiple lines, adding markers, customizing axes, annotating the graph, and adding a legend and title.

ALGORITHM

STEP 1: Import the pyplot module from the Matplotlib library.

STEP 2: Define lists for the x-axis and y-axis values for multiple plots.

STEP 3: Plot the data using the plt.plot() function, with and without markers.

Customize the markers using format strings (e.g., "or" for red circles).

STEP 4: Use plt.xlabel() and plt.ylabel() to set the labels for the x-axis and y-axis, respectively.

STEP 5: Use ax.legend() to add a legend that explains the plotted data.

STEP 6: Use plt.annotate() to add text annotations to specific points on the plot

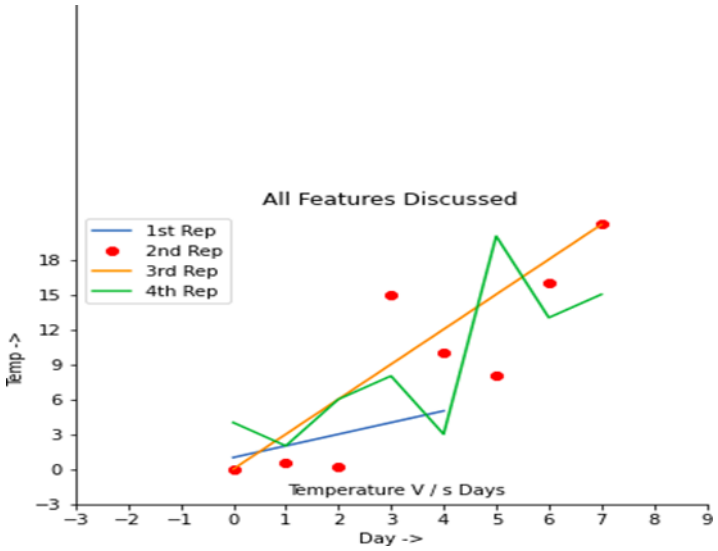
STEP 7: Use plt.title() to set the title for the graph.

PROGRAM:

```
import matplotlib.pyplot as plt
# x-axis values
a = [1, 2, 3, 4, 5]
# corresponding y-axis values
b = [0, 0.6, 0.2, 15, 10, 8, 16, 21]
# Plotting the points
plt.plot(a)
# Plot with circles and red color
plt.plot(b, "or")
```

```
# Plot another line
plt.plot(list(range(0, 22, 3)))
# Naming the x-axis
plt.xlabel('Day ->')
# Naming the y-axis
plt.ylabel('Temp ->')
# Another set of y-axis values
c = [4, 2, 6, 8, 3, 20, 13, 15]
plt.plot(c, label='4th Rep')
# Get current axes command
ax = plt.gca()
# Hide specific spines
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
# Set the bounds for the left spine
ax.spines['left'].set_bounds(-3, 40)
# Set the interval for x-axis ticks
plt.xticks(list(range(-3, 10)))
# Set the interval for y-axis ticks
plt.yticks(list(range(-3, 20, 3)))
# Add a legend
ax.legend(['1st Rep', '2nd Rep', '3rd Rep', '4th Rep'])
# Annotate the plot
plt.annotate('Temperature V / s Days', xy=(1.01, -2.15))
# Title for the graph
plt.title('All Features Discussed')
# Show the plot
plt.show()
```

OUTPUT



RESULT:

The result is a detailed plot with multiple features demonstrated using Matplotlib in Python. This includes multiple plots, custom markers, axis customization, legends, annotations, and a title.

Ex.No: 2(c)	BASIC PLOTS USING MATPLOTLIB
Date:	

AIM:

To demonstrate how to create a single figure with multiple subplots using Matplotlib in Python. Each subplot displays different data and has its own customization such as markers, titles, and axis ticks.

ALGORITHM:

STEP 1: Import the pyplot module from the Matplotlib library.

STEP 2: Define lists a, b, and c with data for each subplot.

STEP 3: Create a figure using plt.figure() and specify the size.

Create subplots using plt.subplot() and specify the number of rows, columns, and the position of each subplot within the grid.

STEP 4: Use subplot.plot() to plot data on each subplot.

Customize each subplot with different markers, titles, and axis ticks using the respective functions (set_xticks(), set_yticks(), set_title()).

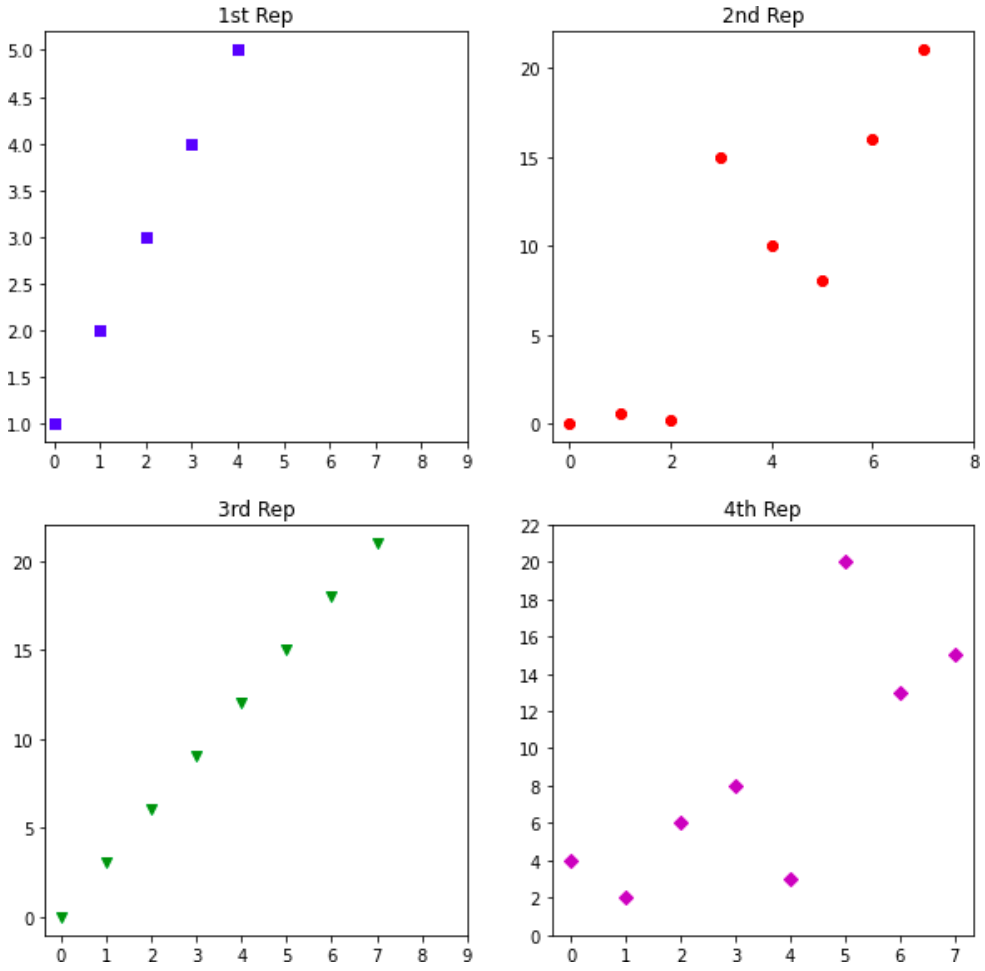
STEP 5: Use plt.show() to display the plot.

PROGRAM

```
import matplotlib.pyplot as plt

# Data for subplots

a = [1, 2, 3, 4, 5]

b = [0, 0.6, 0.2, 15, 10, 8, 16, 21]

c = [4, 2, 6, 8, 3, 20, 13, 15]

# Create a figure with subplots
```

```
fig = plt.figure(figsize=(10, 10))
sub1 = plt.subplot(2, 2, 1)
sub2 = plt.subplot(2, 2, 2)
sub3 = plt.subplot(2, 2, 3)
sub4 = plt.subplot(2, 2, 4)
# Plot data on subplots and customize
sub1.plot(a, 'sb')
sub1.set_xticks(list(range(0, 10, 1)))
sub1.set_title('1st Rep')
sub2.plot(b, 'or')
sub2.set_xticks(list(range(0, 10, 2)))
sub2.set_title('2nd Rep')
sub3.plot(list(range(0, 22, 3)), 'vg')
sub3.set_xticks(list(range(0, 10, 1)))
sub3.set_title('3rd Rep')
sub4.plot(c, 'Dm')
sub4.set_yticks(list(range(0, 24, 2)))
sub4.set_title('4th Rep')
# Display the plot
plt.show()
```

OUTPUT:



RESULT:

The result is a visualization containing multiple subplots within a single figure, effectively demonstrating how to create and customize subplots using Matplotlib.

Ex.No: 3(a)	FREQUENCY DISTRIBUTION
Date:	

AIM:

To demonstrate the use of NLTK (Natural Language Toolkit) to tokenize text from the Gutenberg corpus, calculate the frequency of words, and display word-frequency pairs.

ALGORITHM

STEP 1: Import the word_tokenize function from nltk.tokenize to tokenize the text.

Import the gutenberg corpus from nltk.corpus to access the "blake-poems.txt" text from the Gutenberg corpus.

STEP 2: Use the gutenberg.raw() function to retrieve the raw text of "blake-poems.txt" from the Gutenberg corpus.

Tokenize the raw text using the word_tokenize() function.

STEP 3: Create an empty list wlist to store the first 50 tokens from the tokenized text.

STEP 4: Count the frequency of each word in the wlist using list comprehension and the count() method.

STEP 5: Print the word-frequency pairs using the zip() function to combine tokens and their corresponding frequencies.

PROGRAM:

```
from nltk.tokenize import word_tokenize
from nltk.corpus import gutenberg
# Load and tokenize text
sample = gutenberg.raw("blake-poems.txt")
tokens = word_tokenize(sample)
# Create a sample list
wlist = [tokens[i] for i in range(50)]
# Calculate word frequencies
wordfreq = [wlist.count(w) for w in wlist]
```

```
# Display word-frequency pairs
print("Pairs\n" + str(list(zip(wlist, wordfreq))))
```

OUTPUT:

Pairs

[('From', 1), ('the', 3), ('preamble', 1), ('to', 1), ('Songs', 1), ('of', 2), ('Innocence', 1), ('and', 2), ('Experience', 1), (',', 4), ('Blake', 1), ('made', 1), ('it', 1), ('clear', 1), ('that', 1), ('his', 1), ('work', 1), ('was', 1), ('to', 1), ('be', 1), ('judged', 1), ('as', 1), ('altogether', 1), ('on', 1), ('its', 1), ('own', 1), ('merits', 1), ('.', 1), ('He', 1), ('was', 1), ('an', 1), ('innocent', 1), ('.', 1), ('His', 1), ('purpose', 1), ('was', 1), ('not', 1), ('merely', 1), ('to', 1), ('reveal', 1), ('the', 1), ('wonder', 1), ('and', 1), ('terror', 1), ('of', 1), ('the', 1), ('universe', 1), (',', 1), ('but', 1), ('to', 1), ('draw', 1)]

RESULT

The result is a list of word-frequency pairs extracted from the first 50 tokens of the "blake-poems.txt" text from the Gutenberg corpus. Each pair consists of a token and its frequency in the sample text. This program showcases how to tokenize text, count word frequencies, and display the results using NLTK in Python.

Ex.No: 3(b)	AVERAGES
Date:	

AIM:

The aim of this code is to calculate the weighted average of the 'salary_p_year' column in a DataFrame using the numpy.average() function.

ALGORITHM

STEP 1: Import the necessary libraries, including NumPy and possibly Pandas if it's used to handle the DataFrame.

STEP 2: Use the numpy.average() function to calculate the weighted average of the 'salary_p_year' column in the DataFrame.

Provide the column containing the values to be averaged ('salary_p_year') as the first argument.

Provide the column containing the weights ('employees_number') as the weights parameter.

Round the result to 2 decimal places.

PROGRAM

```
# Importing the required libraries

import numpy as np

# Calculate weighted average

weighted_avg_m3 = round(np.average(df['salary_p_year'],
weights=df['employees_number']), 2)

# Display the result

weighted_avg_m3
```

OUTPUT

44225.35

RESULT

The result provides a single value representing the weighted average salary per year, which accounts for the number of employees in the calculation.

Ex.No: 3 (c)	VARIABILITY
Date:	

AIM:

To demonstrate the use of the variance() function from the statistics module to calculate the variance of different sets of data, including positive integers, negative integers, a mix of positive and negative numbers, fractional numbers, and floating-point values.

ALGORITHM:

STEP 1: Import the variance function from the statistics module.

Import the Fraction class from the fractions module to handle fractional numbers.

STEP 2: Define tuples containing different sets of data:

A set of positive integers.

A set of negative integers.

A mix of positive and negative numbers.

A set of fractional numbers.

A set of floating-point values.

STEP 3: Use the variance() function to calculate the variance for each set of data.

STEP 4: Print the calculated variance for each sample.

PROGRAM:

```
# Importing the required modules

from statistics import variance

from fractions import Fraction as fr

# Defining sample data

sample1 = (1, 2, 5, 4, 8, 9, 12)

sample2 = (-2, -4, -3, -1, -5, -6)
```

```
sample3 = (-9, -1, 0, 2, 1, 3, 4, 19)

sample4 = (fr(1, 2), fr(2, 3), fr(3, 4), fr(5, 6), fr(7, 8))

sample5 = (1.23, 1.45, 2.1, 2.2, 1.9)

# Calculating and printing variance of each sample

print("Variance of Sample1 is %s" % variance(sample1))

print("Variance of Sample2 is %s" % variance(sample2))

print("Variance of Sample3 is %s" % variance(sample3))

print("Variance of Sample4 is %s" % variance(sample4))

print("Variance of Sample5 is %s" % variance(sample5))
```

OUTPUT:

```
Variance of Sample1 is 15.80952380952381

Variance of Sample2 is 3.5

Variance of Sample3 is 62.839285714285715

Variance of Sample4 is 0.04861111111111111

Variance of Sample5 is 0.17929999999999993
```

RESULT:

The program successfully calculates and displays the variance for various types of data sets, including positive integers, negative integers, mixed positive and negative numbers, fractional numbers, and floating-point values.

Ex.No: 4(a)	NORMAL CURVE
Date:	

AIM:

To demonstrate how to create and visualize a normal distribution using SciPy and Matplotlib. This involves generating data points, calculating the probability density function (PDF), and plotting the distribution.

ALGORITHM:

STEP 1: Import necessary libraries including norm from scipy.stats, numpy, matplotlib.pyplot, and seaborn.

STEP 2: Create a range of data points using numpy.arange().

STEP 3: Set the style for the seaborn plot.

Use plt.plot() to plot the data points and their corresponding PDF values.

Label the x-axis and y-axis.

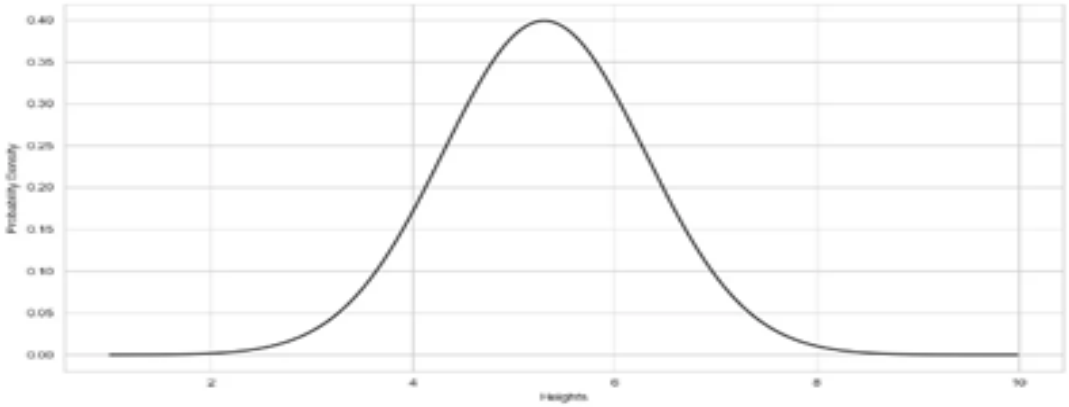
STEP 4: Use plt.show() to display the plot.

PROGRAM:

```
# Import required libraries
from scipy.stats import norm
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sb
# Creating the distribution data
data = np.arange(1, 10, 0.01)
pdf = norm.pdf(data, loc=5.3, scale=1)
```

```
# Visualizing the distribution
sb.set_style('whitegrid')
plt.plot(data, pdf, color='black')
plt.xlabel('Heights')
plt.ylabel('Probability Density')
plt.title('Normal Distribution of Heights')
plt.show()
```

OUTPUT:



RESULT:

The program successfully generates and visualizes a normal distribution using the specified mean and standard deviation. The resulting plot shows the probability density function of the normal distribution, providing a visual representation of how data points are distributed around the mean.

Ex.No: 4(b)	CORRELATION AND SCATTER PLOTS
Date:	

AIM:

To create scatter plots for three different sets of data, calculate their correlations, and display the plots with appropriate labels and correlation values.

ALGORITHM:

STEP 1: Import necessary libraries including numpy and matplotlib.pyplot.

STEP 2: Generate random data points for the x-axis using numpy.random.randn().

Generate three different sets of y-axis data:

y1 based on a linear transformation of x.

y2 based on a negative linear transformation of x.

y3 as another set of random data points.

STEP 3: Calculate the correlation coefficient between x and each set of y-axis data using numpy.corrcoef().

STEP 4: Set the figure size and resolution using plt.rcParams.update().

Create scatter plots for each set of data points.

Label each scatter plot with the corresponding correlation value.

STEP 5: Add a title and legend to the plot.

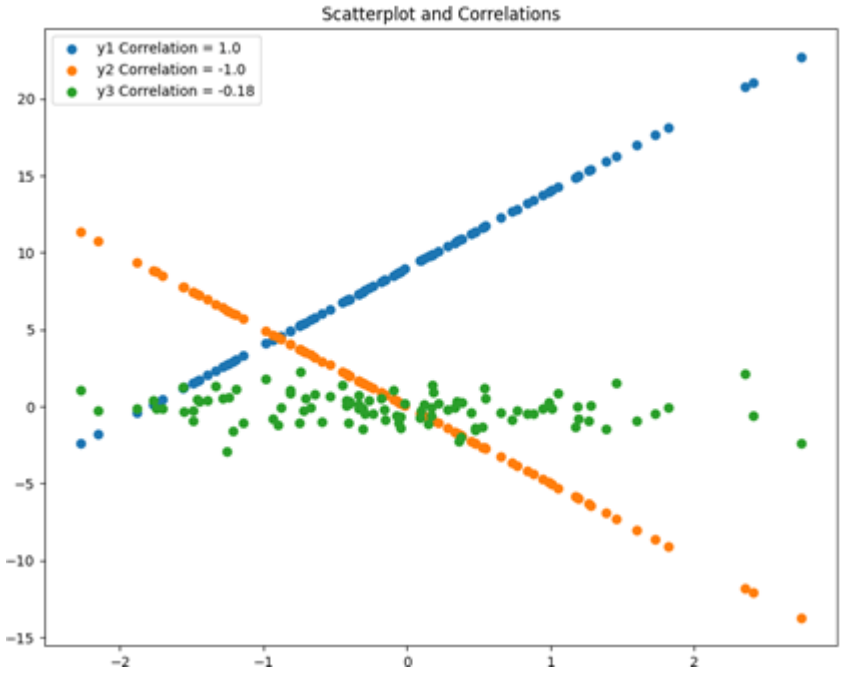
Display the plot using plt.show().

PROGRAM:

```
# Import required libraries

import numpy as np

import matplotlib.pyplot as plt

# Data
```

```
x = np.random.randn(100)

y1 = x * 5 + 9

y2 = -5 * x

y3 = np.random.randn(100)

# Calculate correlations

corr_y1 = np.round(np.corrcoef(x, y1)[0, 1], 2)

corr_y2 = np.round(np.corrcoef(x, y2)[0, 1], 2)

corr_y3 = np.round(np.corrcoef(x, y3)[0, 1], 2)

# Plot

plt.rcParams.update({'figure.figsize': (10, 8), 'figure.dpi': 100})

plt.scatter(x, y1, label=f'y1, Correlation = {corr_y1}')

plt.scatter(x, y2, label=f'y2, Correlation = {corr_y2}')

plt.scatter(x, y3, label=f'y3, Correlation = {corr_y3}')

# Plot enhancements

plt.title('Scatterplot and Correlations')

plt.legend()

plt.show()
```

OUTPUT:



RESULT:

The program successfully creates scatter plots for three different sets of data and calculates their correlation coefficients. The resulting plot visualizes the relationships between x and each of the y-axis data sets, with annotations showing the correlation values.

Ex.No: 4(c)	CORRELATION COEFFICIENT
Date:	

AIM:

To calculate the correlation coefficient between two arrays of data points using a manual method without relying on external libraries.

ALGORITHM:

STEP 1: Import the math module to use mathematical functions.

STEP 2: Define a function correlationCoefficient(X, Y, n) that takes two lists X and Y of the same length n and calculates their correlation coefficient.

STEP 3: Initialize variables to store the sum of elements of X, sum of elements of Y, sum of products of X[i] and Y[i], sum of squares of X[i], and sum of squares of Y[i].

STEP 4: Use a while loop to iterate through the elements of the lists and update the sums accordingly.

STEP 5: Use the formula for the correlation coefficient:

$$\text{corr} = \frac{n \cdot \sum XY - \sum X \cdot \sum Y}{\sqrt{(n \cdot \sum X^2 - (\sum X)^2) \cdot (n \cdot \sum Y^2 - (\sum Y)^2)}}$$

STEP 6: Return the calculated correlation coefficient.

STEP 7: Define two lists X and Y.

Find the length of the lists.

Call the function and print the correlation coefficient formatted to six decimal places.

PROGRAM:

```
import math

# Function to calculate the correlation coefficient
```

```
def correlationCoefficient(X, Y, n):
    sum_X = 0
    sum_Y = 0
    sum_XY = 0
    squareSum_X = 0
    squareSum_Y = 0
    for i in range(n):
        sum_X += X[i]
        sum_Y += Y[i]
        sum_XY += X[i] * Y[i]
        squareSum_X += X[i] * X[i]
        squareSum_Y += Y[i] * Y[i]
    # Calculate correlation coefficient
    corr = (n * sum_XY - sum_X * sum_Y) / math.sqrt((n * squareSum_X - sum_X * sum_X)
* (n * squareSum_Y - sum_Y * sum_Y))
    return corr
# Driver function
X = [15, 18, 21, 24, 27]
Y = [25, 25, 27, 31, 32]
# Find the size of the array
n = len(X)
# Function call to correlationCoefficient
print('{0:.6f}'.format(correlationCoefficient(X, Y, n)))
```

OUTPUT:

0.997054

RESULT:

The program successfully calculates and displays the correlation coefficient between the two arrays of data points X and Y. The correlation coefficient is a measure of the strength and direction of the linear relationship between two variables.

Ex.No: 5	SIMPLE LINEAR REGRESSION
Date:	

AIM:

To perform simple linear regression on a set of data points, estimate the regression coefficients, and plot the regression line along with the actual data points.

ALGORITHM:

STEP 1: Import numpy for numerical operations and matplotlib.pyplot for plotting.

STEP 2: Define a function estimate_coef(x, y) that calculates the regression coefficients.

Calculate the number of observations n.

Compute the mean of x and y.

Calculate the cross-deviation SS_xy and deviation about x SS_xx.

Compute the regression coefficients b_1 and b_0.

STEP 3: Define a function plot_regression_line(x, y, b) to plot the regression line.

Create a scatter plot of the actual data points.

Calculate the predicted response y_pred using the estimated coefficients.

Plot the regression line.

Add labels to the plot and display it.

STEP 4: Define the main() function.

Initialize the data points x and y.

Estimate the regression coefficients by calling estimate_coef(x, y).

Print the estimated coefficients.

Plot the regression line by calling plot_regression_line(x, y, b).

STEP 5: Ensure the main function is called when the script is executed.

PROGRAM:

```
import numpy as np

import matplotlib.pyplot as plt

def estimate_coef(x, y):

   # number of observations/points

   n = np.size(x)
```

```
    # mean of x and y vector
    m_x = np.mean(x)
    m_y = np.mean(y)
    # calculating cross-deviation and deviation about x
    SS_xy = np.sum(y*x) - n*m_y*m_x
    SS_xx = np.sum(x*x) - n*m_x*m_x
    # calculating regression coefficients
    b_1 = SS_xy / SS_xx
    b_0 = m_y - b_1*m_x
    return (b_0, b_1)
def plot_regression_line(x, y, b):
    # plotting the actual points as scatter plot
    plt.scatter(x, y, color = "m", marker = "o", s = 30)
    # predicted response vector
    y_pred = b[0] + b[1]*x
    # plotting the regression line
    plt.plot(x, y_pred, color = "g")
    # putting labels
    plt.xlabel('x')
    plt.ylabel('y')
    # function to show plot
    plt.show()
def main():
    # observations / data
```

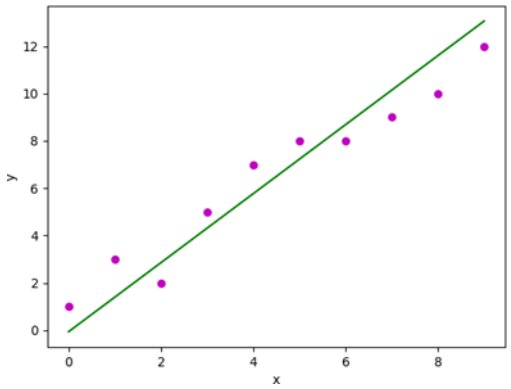
```
    x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
    y = np.array([1, 3, 2, 5, 7, 8, 8, 9, 10, 12])
    # estimating coefficients
    b = estimate_coef(x, y)
    print("Estimated coefficients:\nb_0 = {}\nb_1 = {}".format(b[0], b[1]))
    # plotting regression line
    plot_regression_line(x, y, b)
if __name__ == "__main__":
    main()
```

OUTPUT:

Estimated coefficients:

b_0 = 1.2363636363636363

b_1 = 1.1696969696969697



RESULT:

The program successfully estimates the coefficients of the simple linear regression model and plots the regression line along with the actual data points. The estimated coefficients b_0 and b_1 provide the intercept and slope of the regression line, respectively, and the plot visually represents the relationship between the independent variable x and the dependent variable y.

Ex.No: 6	Z-TEST
Date:	

AIM:

To perform a Z-test to determine if there is a significant difference between a sample mean and a population mean, and to decide whether to reject the null hypothesis based on the calculated Z-score and p-value.

ALGORITHM:

STEP 1: Import the scipy.stats module as stats.

STEP 2: Define a function z_test(sample_mean, population_mean, population_stddev, sample_size, alpha=0.05) to calculate the Z-score, p-value, and determine if the null hypothesis should be rejected.

Calculate the Z-score using the formula

$$Z = \frac{\text{sample\_mean} - \text{population\_mean}}{\text{population\_stddev}/\sqrt{\text{sample\_size}}}$$

Calculate the p-value for a two-tailed test:

p_value =2×(1−stats.norm.cdf(|Z|))

Determine if the null hypothesis should be rejected based on the p-value and significance level (alpha).

STEP 3: Define example values for sample mean, population mean, population standard deviation, sample size, and significance level.

Call the z_test function with the example values.

STEP 4: Print the calculated Z-score, p-value, and decision on the null hypothesis.

PROGRAM:

```
import scipy.stats as stats

import numpy as np

def z_test(sample_mean, population_mean, population_stddev, sample_size, alpha=0.05):

    # Calculate the Z-score

    z_score = (sample_mean - population_mean) / (population_stddev / np.sqrt(sample_size))
```

```
    # Calculate the p-value for a two-tailed test
    p_value = 2 * (1 - stats.norm.cdf(np.abs(z_score)))
    # Determine if we reject the null hypothesis
    reject_null = p_value < alpha
    return z_score, p_value, reject_null
# Example usage
sample_mean = 102.5  # Mean of your sample
population_mean = 100  # Mean of the population
population_stddev = 15  # Standard deviation of the population
sample_size = 50  # Size of your sample
alpha = 0.05  # Significance level
z_score, p_value, reject_null = z_test(sample_mean, population_mean, population_stddev, sample_size, alpha)
print("Z-score:", z_score)
print("P-value:", p_value)
print("Reject null hypothesis:", reject_null)
```

OUTPUT:

```
Z-score: 1.1785113019775793
P-value: 0.2381065488179815
Reject null hypothesis: False
```

RESULT:

The program successfully calculates the Z-score and p-value for the given sample mean, population mean, population standard deviation, and sample size. Based on the p-value and significance level, the program determines whether to reject the null hypothesis.

Ex.No: 7	T-TEST
Date	

AIM:

To perform a one-sample t-test to determine if there is a significant difference between a sample mean and a population mean, and to decide whether to reject the null hypothesis based on the calculated t-statistic and p-value.

ALGORITHM:

STEP 1: Import the stats module from scipy.

STEP 2: Define a function t_test(sample_data, population_mean, alpha=0.05) to calculate the t-statistic, p-value, and determine if the null hypothesis should be rejected.

Use stats.ttest_1samp(sample_data, population_mean) to perform the t-test.

Determine if the null hypothesis should be rejected based on the p-value and significance level (alpha).

STEP 3: Define example values for sample data, population mean, and significance level.

Call the t_test function with the example values.

STEP 4: Print the calculated t-statistic, p-value, and decision on the null hypothesis.

PROGRAM:

```
from scipy import stats

def t_test(sample_data, population_mean, alpha=0.05):
    # Perform the one-sample t-test
    t_statistic, p_value = stats.ttest_1samp(sample_data, population_mean)
        # Determine if we reject the null hypothesis
    reject_null = p_value < alpha
        return t_statistic, p_value, reject_null
```

```
# Example usage

sample_data = [101, 98, 104, 99, 102, 100, 98, 105, 103, 100]

# Sample data

population_mean = 100

# Mean of the population

alpha = 0.05  # Significance level

t_statistic, p_value, reject_null = t_test(sample_data, population_mean, alpha)

print("T-statistic:", t_statistic)

print("P-value:", p_value)

print("Reject null hypothesis:", reject_null)
```

OUTPUT:

T-statistic: 0.9079809994790935

P-value: 0.3871260836665194

Reject null hypothesis: False

RESULT:

The program successfully calculates the t-statistic and p-value for the given sample data and population mean. Based on the p-value and significance level, the program determines whether to reject the null hypothesis.

Ex.No: 8(a)	One Way ANOVA
Date:	

AIM:

To perform a one-way ANOVA (Analysis of Variance) to determine if there are any statistically significant differences between the means of three independent groups.

ALGORITHM:

STEP 1: Import the stats module from scipy.

STEP 2: Define three groups of sample data, each with different sample sizes.

STEP 3: Use stats.f_oneway(group1, group2, group3) to perform the one-way ANOVA test.

The function returns the F-value and the p-value.

STEP 4: Print the calculated F-value and p-value.

PROGRAM:

```
import scipy.stats as stats

# Sample data: three groups with different sample sizes

group1 = [23, 20, 22, 21, 24]

group2 = [30, 31, 29, 32, 28]

group3 = [40, 41, 39, 42, 38]

# Perform one-way ANOVA

f_value, p_value = stats.f_oneway(group1, group2, group3)

print(f"F-value: {f_value}")

print(f"P-value: {p_value}")
```

OUTPUT:

F-value: 140.8

P-value: 3.397830145182732e-07

RESULT:

The program successfully performs a one-way ANOVA test and calculates the F-value and p-value for the given sample data. The F-value is 140.8, and the p-value is approximately 3.4e-07, which is significantly lower than the typical significance level of 0.05.

Ex.No: 8(b)	Two-Way ANOVA
Date:	

AIM:

To perform a two-way ANOVA (Analysis of Variance) with interaction using sample data to determine if there are statistically significant differences in the scores based on different treatments and times, as well as the interaction between treatment and time.

ALGORITHM:

STEP 1: Import pandas for data manipulation.

Import statsmodels.api and statsmodels.formula.api for statistical analysis.

STEP 2: Define a dictionary with sample data for Score, Treatment, and Time.

Create a DataFrame from the dictionary using pd.DataFrame(data).

STEP 3: Use ols from statsmodels.formula.api to fit the model with Score as the dependent variable and Treatment, Time, and their interaction as independent variables.

Fit the model using model = ols('Score ~ C(Treatment) + C(Time) + C(Treatment):C(Time)', data=df).fit().

STEP 4: Use sm.stats.anova_lm(model, typ=2) to perform the ANOVA and generate the ANOVA table.

STEP 5: Print the ANOVA table to display the results.

PROGRAM:

```
import pandas as pd

import statsmodels.api as sm

from statsmodels.formula.api import ols

# Sample data

data = {

   'Score': [23, 20, 22, 21, 24, 30, 31, 29, 32, 28, 40, 41, 39, 42, 38],
```

```
    'Treatment': ['A', 'A', 'A', 'A', 'A', 'B', 'B', 'B', 'B', 'B', 'C', 'C', 'C', 'C', 'C'],
    'Time': ['Morning', 'Morning', 'Afternoon', 'Afternoon', 'Evening', 'Morning', 'Morning', 'Afternoon', 'Afternoon', 'Evening', 'Morning', 'Morning', 'Afternoon', 'Afternoon', 'Evening']
}
# Create DataFrame
df = pd.DataFrame(data)
# Fit the model
model = ols('Score ~ C(Treatment) + C(Time) + C(Treatment):C(Time)', data=df).fit()
anova_table = sm.stats.anova_lm(model, typ=2)
print(anova_table)
```

OUTPUT:

	sum_sq	df	F	PR(>F)
C(Treatment)	1162.933333	2.0	248.517045	2.832936e-09
C(Time)	18.266667	2.0	3.905797	0.057324e-01
C(Treatment):C(Time)	5.333333	4.0	0.570145	0.689965e-01
Residual	14.000000	10.0	NaN	NaN

RESULT:

The program successfully performs a two-way ANOVA with interaction on the given sample data.

Ex.No: 9	BUILDING AND VALIDATING LINEAR MODELS
Date:	

AIM:

To generate a synthetic dataset for linear regression with two predictors, visualize the dataset in a 3D scatter plot, and demonstrate the relationship between the predictors and the response variable.

ALGORITHM:

STEP 1: Import numpy for numerical operations.

Import matplotlib.pyplot for plotting.

STEP 2: Create a function generate_dataset(n) that generates n data points.

Initialize two random coefficients random_x1 and random_x2 using np.random.rand().

Generate two predictors x1 (linearly increasing) and x2 (a combination of linear and random values).

Calculate the response variable y using the linear combination of x1 and x2 with added randomness.

STEP 3: Call generate_dataset(200) to generate a dataset with 200 data points.

STEP 4: Create a 3D scatter plot using matplotlib to visualize the relationship between x1, x2, and y.

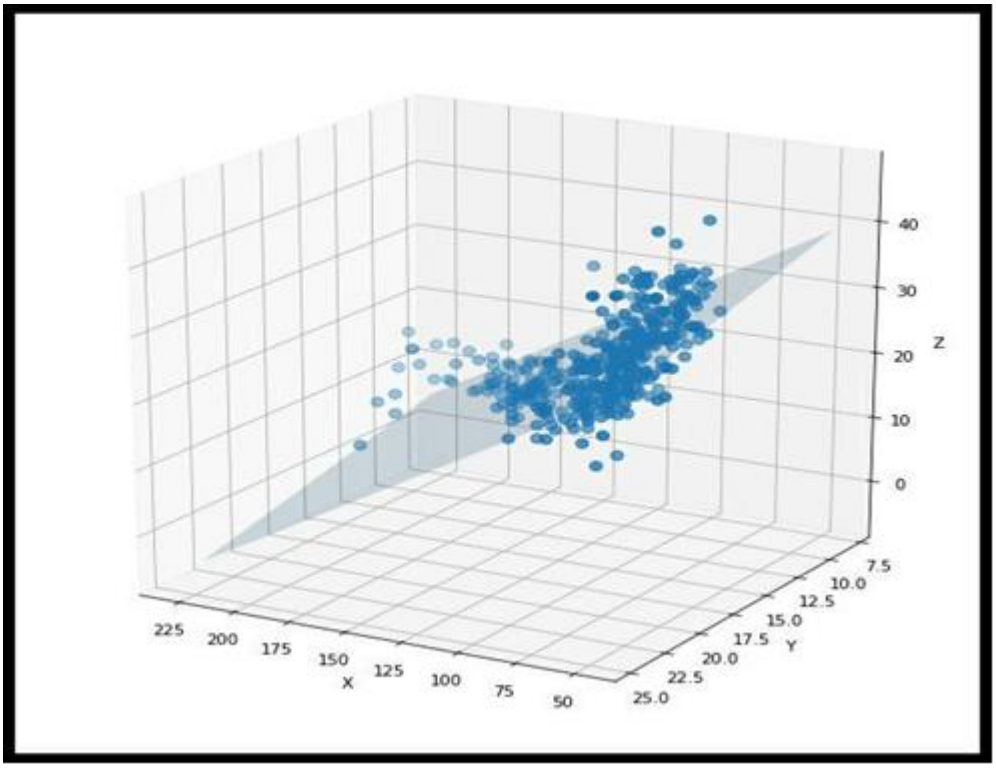
Configure the plot to display the data points and set the viewing angle.

PROGRAM:

```
import numpy as np

import matplotlib.pyplot as plt

def generate_dataset(n):

  x = []
```

```
    y = []
    random_x1 = np.random.rand()
    random_x2 = np.random.rand()
    for i in range(n):
        x1 = i
        x2 = i / 2 + np.random.rand() * n
        x.append([1, x1, x2])
        y.append(random_x1 * x1 + random_x2 * x2 + 1)
    return np.array(x), np.array(y)
x, y = generate_dataset(200)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x[:, 1], x[:, 2], y, label='y', s=5)
ax.legend()
ax.view_init(45, 0)
plt.show()
```

OUTPUT:



RESULT:

The program successfully generates a synthetic dataset with two predictors and a response variable. The 3D scatter plot visualizes the relationship between the predictors and the response, showing how y varies with x1 and x2. This visual representation helps in understanding the linear relationships and interactions within the generated data.

Ex.No:10	BUILDING AND VALIDATING LOGISTIC MODELS
Date:	

AIM:

To build, train, and evaluate a logistic regression model using synthetic data. The goal is to predict binary outcomes and assess the model's performance using accuracy and a classification report.

ALGORITHM:

STEP 1: Import numpy for generating synthetic data.

Import train_test_split from sklearn.model_selection for splitting data.

Import LogisticRegression from sklearn.linear_model for model building.

Import accuracy_score and classification_report from sklearn.metrics for model evaluation.

STEP 2: Set a random seed for reproducibility using np.random.seed(42).

Generate 100 samples with 2 features using np.random.randn(100, 2).

Generate binary labels (0 or 1) for these samples using np.random.randint(0, 2, 100).

STEP 3: Split the data into training (80%) and testing (20%) sets using train_test_split.

STEP 4: Create an instance of LogisticRegression.

Fit the model to the training data using model.fit(X_train, y_train).

STEP 5: Use the trained model to predict labels for the test set using model.predict(X_test).

STEP 6: Calculate the accuracy of the model using accuracy_score(y_test, y_pred).

Generate a classification report using classification_report(y_test, y_pred).

STEP 7: Print the accuracy and classification report.

PROGRAM:

```
import numpy as np
```

```
from sklearn.model_selection import train_test_split

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import accuracy_score, classification_report

# Generating synthetic data

np.random.seed(42)

X = np.random.randn(100, 2)  # 100 samples, 2 features

y = np.random.randint(0, 2, 100)  # Binary labels (0 or 1)

# Splitting data into training and testing sets

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

# Building the logistic regression model

model = LogisticRegression()

model.fit(X_train, y_train)

# Predictions on the test set

y_pred = model.predict(X_test)

# Model evaluation

accuracy = accuracy_score(y_test, y_pred)

report = classification_report(y_test, y_pred)

# Output

print("Logistic Regression Model Evaluation:")

print("Accuracy:", accuracy)

print("Classification Report:")

print(report)
```

OUTPUT:

Logistic Regression Model Evaluation:

Accuracy: 0.65

Classification Report:

	precision	recall	f1-score	support
0	0.83	0.45	0.59	11
1	0.57	0.89	0.70	9
accuracy		0.65	20	
macro avg	0.70	0.67	0.64	20
weighted avg	0.72	0.65	0.64	20

RESULT:

The program successfully builds, trains, and evaluates a logistic regression model using synthetic data. The model's performance is assessed based on accuracy and a detailed classification report that includes precision, recall, and F1-score for each class (0 and 1). This process demonstrates the effectiveness of logistic regression in binary classification tasks.

Ex.No: 11	TIME SERIES ANALYSIS
Date:	

AIM:

To generate a sample time series dataset, visualize it, and perform time series decomposition to extract and visualize its trend, seasonal, and residual components.

ALGORITHM:

STEP 1: Import pandas for handling time series data.

Import numpy for numerical operations.

Import matplotlib.pyplot for plotting.

Import seasonal_decompose from statsmodels.tsa.seasonal for time series decomposition**.**

STEP 2: Create a date range from January 1, 2022, to December 31, 2022, with daily frequency using pd.date_range.

Generate random data points corresponding to each date using np.random.randn.

STEP 3: Plot the generated time series data using plt.plot to observe the overall pattern.

STEP 4: Decompose the time series into trend, seasonal, and residual components using seasonal_decompose with the 'additive' model.

STEP 5: Plot the original time series data.

Plot the trend component.

Plot the seasonal component.

Plot the residual component.

Use plt.subplot to organize multiple plots in a single figure.

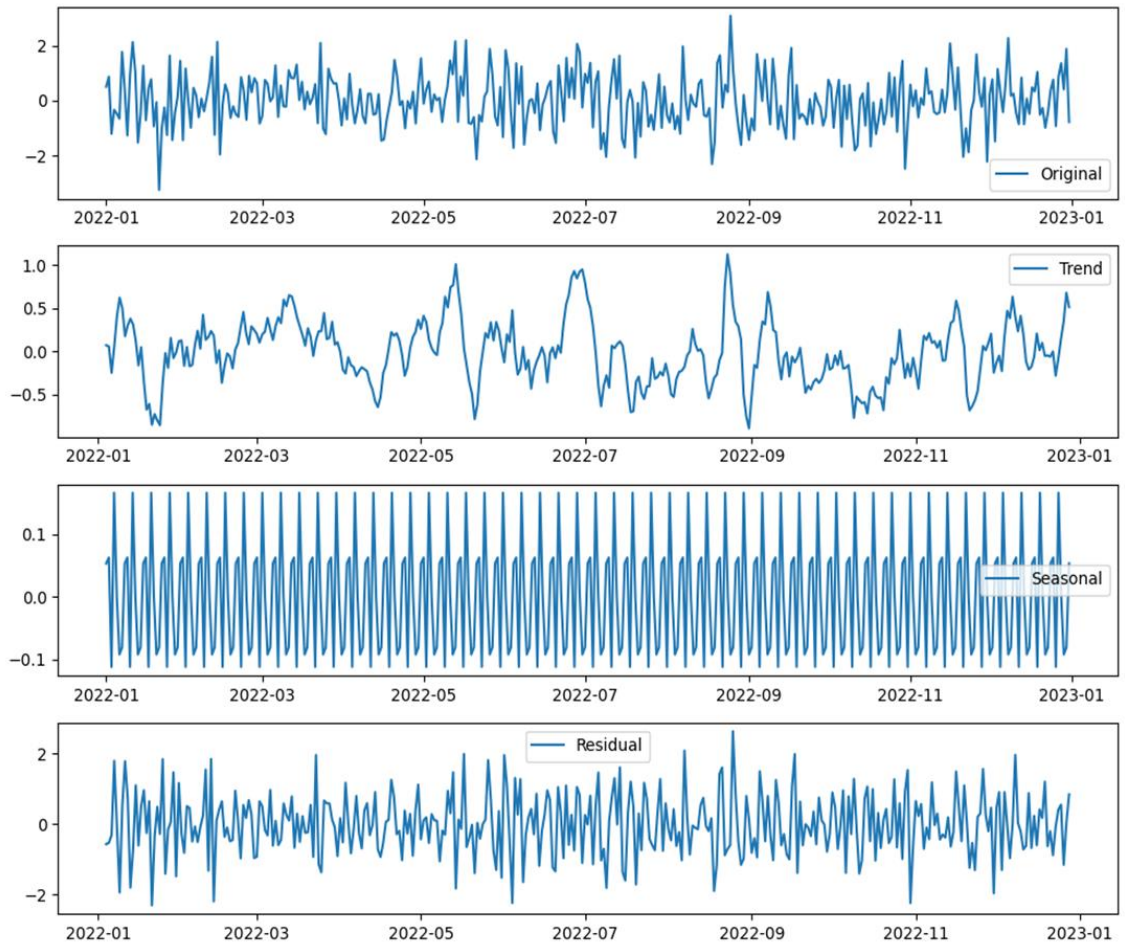
PROGRAM:

```
import pandas as pd
```

```
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.seasonal import seasonal_decompose
# Sample time series data
date_range = pd.date_range(start='2022-01-01', end='2022-12-31', freq='D')
data = np.random.randn(len(date_range))
ts = pd.Series(data, index=date_range)
# Visualize the time series data
plt.figure(figsize=(10, 6))
plt.plot(ts)
plt.title('Sample Time Series Data')
plt.xlabel('Date')
plt.ylabel('Value')
plt.grid(True)
plt.show()
# Decompose the time series into trend, seasonal, and residual components
decomposition = seasonal_decompose(ts, model='additive')
# Plot the decomposed components
trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid
plt.figure(figsize=(10, 8))
plt.subplot(411)
plt.plot(ts, label='Original')
```

```
plt.legend(loc='best')
plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='best')
plt.subplot(413)
plt.plot(seasonal, label='Seasonal')
plt.legend(loc='best')
plt.subplot(414)
plt.plot(residual, label='Residual')
plt.legend(loc='best')
plt.tight_layout()
plt.show()
```

OUTPUT:



RESULT:

The program successfully generates and visualizes a sample time series dataset. The time series is decomposed into its trend, seasonal, and residual components. The resulting plots provide a clear view of each component, helping to understand the underlying patterns and variations within the time series data.