1.Write a python program to compute Central Tendency Measures: Mean, Median, Mode Measure of Dispersion: Variance, Standard Deviation

Code:

```
import statistics as st

try:
    # Input list of integers
    x = list(map(int, input("Enter numbers separated by space: ").split())
    # Central Tendency Measures
    mean = st.mean(x)
    median = st.median(x)
    mode = st.mode(x)
    multimode = st.multimode(x)  # Returns all modes (if multiple)
    # Measures of Dispersion
    var = st.variance(x)
    std = st.stdev(x)
    # Display Results
    print("Mean:", mean)
    print("Median:", median)
    print("Mode:", mode)
    print("Multimode:", multimode)
    print("Variance:", var)
    print("Standard Deviation:", std)
except st.StatisticsError:
    print("Statistical operations require at least one data point.")
except ValueError:
    print("Only numerical values are allowed.")
```

Output:

```
======= RESTART: C:/Users/M.Hema Sıra Ramya/OneDrive/Desktop/ML lab/ml.py ===
1 2 3 4 2 1
Mean:  2.1666666666666665
Median:  2.0
Mode:  1
Variance:  1.3666666666666667
Standard deviation :  1.1690451944500122
Multimode:  [1, 2]
```

2.) Study of Python Basic Libraries

Includes: statistics, math, numpy, scipy

🔢 1. Statistics Library

The statistics module provides basic statistical operations for numeric data, such as mean, median, and standard deviation.
Useful in simple data analysis tasks, teaching, or when NumPy/Pandas is not available.

✅ Important Functions:

- mean(): Calculates the average of numeric data.
- median(): Finds the middle value of a dataset.
- mode(): Identifies the most frequent value.
- stdev(): Computes sample standard deviation.
- variance(): Calculates how spread out the values are.

💻 Example Code:

```
import statistics

data = [10, 20, 20, 40, 50]

print("Mean:", statistics.mean(data))

print("Median:", statistics.median(data))

print("Mode:", statistics.mode(data))

print("Standard Deviation:", statistics.stdev(data))

print("Variance:", statistics.variance(data))
```

```
Mean: 28
Median: 20
Mode: 20
Standard Deviation: 16.431676725154983
Variance: 270
```

📐 2. Math Library

The math module provides access to mathematical functions like powers, roots, trigonometry, and constants like π and e.

It’s a built-in library. Useful in geometry, trigonometry, scientific simulations, and financial formulas.

✅ **Important Functions:**

- sqrt(): Returns the square root of a number.
- pow(): Raises a number to the power of another.
- factorial(): Calculates the factorial of a number.
- sin(): Computes sine of an angle (in radians).
- log(): Returns the natural logarithm of a number.

💻 **Example Code:**

```
import math

print("Square root of 16:", math.sqrt(16))

print("2 to the power 3:", math.pow(2, 3))

print("Factorial of 5:", math.factorial(5))

print("Sine of 90 degrees:", math.sin(math.radians(90)))

print("Natural log of e:", math.log(math.e))
```

```
Square root of 16: 4.0
2 to the power 3: 8.0
Factorial of 5: 120
Sine of 90 degrees: 1.0
Natural log of e: 1.0
```

🔣 **3. NumPy Library**

Common in machine learning, data science, numerical methods, and simulations.

✅ **Important Functions:**

- array(): Creates an array for numerical computation.
- arange(): Generates evenly spaced numbers within a range.
- reshape(): Changes the shape of an array without changing data.
- mean(): Calculates average of array elements.
- dot(): Performs dot product of two arrays.
- linspace(): Generates evenly spaced numbers over a specified interval.

💻 **Example Code:**

```
import numpy as np

a = np.array([1, 2, 3])

b = np.array([4, 5, 6])

print("Array A:", a)

print("Array B:", b)

print("Dot product:", np.dot(a, b))

print("Mean of A:", np.mean(a))

print("Reshaped:", np.reshape(np.arange(6), (2, 3)))

print("Linspace:", np.linspace(1, 5, 8))
```

```
Array A: [1 2 3]
Array B: [4 5 6]
Dot product: 32
Mean of A: 2.0
Reshaped: [[0 1 2]
 [3 4 5]]
Linspace:  [1.         1.57142857 2.14285714 2.71428571 3.28571429 3.85714286
 4.42857143 5.        ]
```

🧪 4. SciPy Library

🔹 a. scipy.special

Use when dealing with scientific or mathematical computations involving special functions. Useful in advanced math, physics, or engineering problems.

✅ **Important Functions:**

- gamma(): Computes the Gamma function.
- erf(): Computes the error function.
- comb(): Calculates combinations (n choose k).
- perm(): Calculates permutations.

💻 **Example Code:**

```
from scipy import special

print("Gamma(5):", special.gamma(5))

print("Erf(1):", special.erf(1))
```

```
print("Combinations (5C2):", special.comb(5, 2))
```

```
Gamma(5): 24.0
Erf(1): 0.8427007929497148
Combinations (5C2): 10.0
```

🔹 b. scipy.linalg

Use for performing linear algebra operations.
Useful in solving equations, matrix operations, and numerical analysis.

✅ Important Functions:

- inv(): Computes the inverse of a matrix.
- det(): Calculates the determinant of a matrix.
- eig(): Returns eigenvalues and eigenvectors.
- eigvals(): Returns only the eigenvalues.
- solve(): Solves systems of linear equations.

💻 Example Code:

```
from scipy import linalg

import numpy as np

A = np.array([[1, 2], [3, 4]])

print("Inverse:\n", linalg.inv(A))

print("Determinant:", linalg.det(A))

print("Eigenvalues:", linalg.eigvals(A))
```

```
Inverse:
 [[-2.   1. ]
 [ 1.5 -0.5]]
Determinant: -2.0
Eigenvalues: [-0.37228132+0.j  5.37228132+0.j]
```

🔹 c. scipy.interpolate

Use for estimating unknown values between known data points.
Useful in data smoothing, graphics, and numerical modeling.

✅ Important Functions:

- interp1d(): Creates a function for linear or spline interpolation.
- griddata(): Interpolates unstructured data.
- splrep(): Computes B-spline representation of 1D curve.
- splev(): Evaluates B-spline or its derivatives.
- lagrange(): Performs Lagrange polynomial interpolation.

💻 **Example Code:**

```
from scipy.interpolate import interp1d
import numpy as np
x = np.array([0, 1, 2, 3])
y = np.array([0, 2, 4, 6])
f = interp1d(x, y)
print("Interpolated value at 1.5:", f(1.5))
```

```
Interpolated value at 1.5: 3.0
```

🔹 **d. scipy.optimize**

Use for finding minima, maxima, or roots of functions.
Useful in operations research, machine learning, and engineering optimization.

✅ **Important Functions:**

- minimize(): Minimizes a scalar function.
- root(): Finds a root of a function.
- linprog(): Solves linear programming problems.
- curve_fit(): Fits a curve to data points.
- least_squares(): Solves nonlinear least-squares problems.

💻 **Example Code:**

```
from scipy.optimize import minimize
f = lambda x: x**2 + 3*x + 2
res = minimize(f, x0=0)
print("Minimum at:", res.x)
```

```
Minimum at: [-1.50000001]
```

🔹 e. scipy.ndimage

Use for image processing and multi-dimensional data filtering.
Useful in computer vision, image analysis, and scientific imaging.

✅ **Important Functions:**

- gaussian_filter(): Applies Gaussian smoothing filter.
- sobel(): Computes the Sobel gradient of image.
- median_filter(): Applies median filter to reduce noise.
- rotate(): Rotates an image array.
- zoom(): Zooms in or out on an image.

💻 **Example Code:**

```
from scipy import ndimage

import numpy as np

data = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

blurred = ndimage.gaussian_filter(data, sigma=1)

print("Blurred:\n", blurred)
```

Output:

```
Blurred:
 [[2 3 3]
 [4 5 5]
 [5 6 6]]
```

3.) Study of Python Libraries for ML Applications

Libraries: Pandas, Matplotlib

🐼 1. Pandas

Pandas is a powerful Python library for data manipulation and analysis using labeled data structures.
Used in data cleaning, exploration, and preprocessing in machine learning workflows and data science tasks.

✅ Important Functions:

- read_csv(): Reads a CSV file into a DataFrame.
- head(): Displays the first few rows of a DataFrame.
- describe(): Summarizes statistics of numerical columns.
- drop(): Removes specified rows or columns.
- fillna(): Replaces missing values with a specified value.

💻 Example Code:

python

CopyEdit

```
import pandas as pd

df = pd.read_csv("data.csv")

print(df.head())

print(df.describe())

df = df.drop(columns=["UnwantedColumn"])

df = df.fillna(0)
```

📝 Output:

Data.csv

Name	Age	Marks	UnwantedColumn
Alice	20	85	
Bob	22	78	extra
Charlie	21	90	extra
David	23		extra
Eva	20	88	extra

	Name	Age	Marks	UnwantedColumn
0	Alice	20	85.0	NaN
1	Bob	22	78.0	extra
2	Charlie	21	90.0	extra
3	David	23	NaN	extra
4	Eva	20	88.0	extra

	Age	Marks
count	5.00000	4.000000
mean	21.20000	85.250000
std	1.30384	5.251984
min	20.00000	78.000000
25%	20.00000	83.250000
50%	21.00000	86.500000
75%	22.00000	88.500000
max	23.00000	90.000000

📊 2. Matplotlib

Matplotlib is a plotting library used to create static, animated, and interactive visualizations in Python.
Used for visualizing data trends, distributions, and comparisons during analysis and ML model evaluation.

✅ Important Functions:

- plot(): Creates a line plot of data.
- scatter(): Makes a scatter plot of points.
- bar(): Generates bar charts.
- hist(): Plots a histogram to show data distribution.
- show(): Displays the plot window.

💻 Example Code:

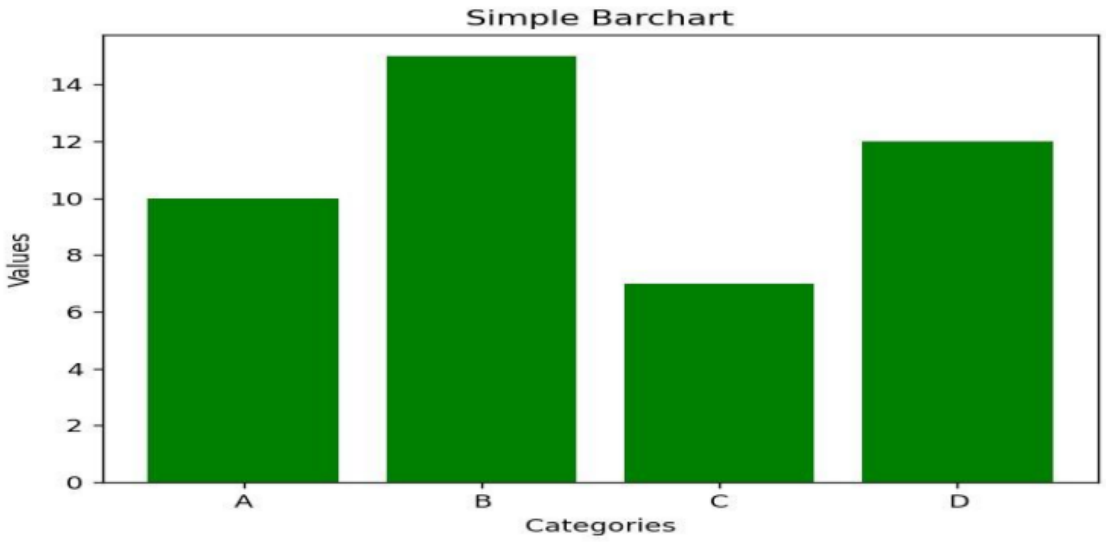
python

CopyEdit

```
import matplotlib.pyplot as plt

x = ['A', 'B', 'C', 'D']

y = [10, 15, 7, 12]
```

```
plt.bar(x, y, color='green')
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Simple Barchart')
plt.show()
```

📄 **Output:**



4.Write a Python program to implement Simple Linear Regression

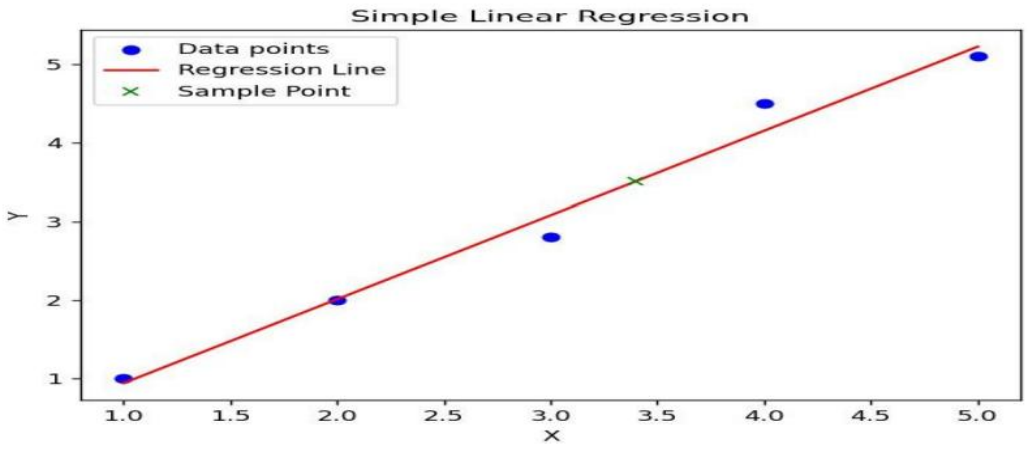
Code:

```
import numpy as np
import matplotlib.pyplot as plt


# Function to calculate slope (m) and intercept (b)
def simple_linear_regression(x, y):
    Xmean = np.mean(x)
    Ymean = np.mean(y)
    numerator = np.sum((x - Xmean) * (y - Ymean))
    denominator = np.sum((x - Xmean) ** 2)
    m = numerator / denominator
    b = Ymean - m * Xmean
    return m, b
# Function to make predictions
def predict(x, m, b):
    return m * x + b
# Input values
x = list(map(float, input("Enter X values separated by space: ").split()))
y = list(map(float, input("Enter Y values separated by space: ").split()))
# Convert to numpy arrays
x = np.array(x)
y = np.array(y)
# Get slope and intercept
m, b = simple_linear_regression(x, y)
# Predict values for all X
prediction = predict(x, m, b)
```

```
# Plot original data points
plt.scatter(x, y, color='blue', label='Data points')
# Plot regression line
plt.plot(x, prediction, color='red', label='Regression Line')
# Predict for a sample value
sample_x = 3.4
predicted_y = predict(sample_x, m, b)
print(f'Predicted Y for X = {sample_x}: {predicted_y}')
# Mark the predicted point
plt.plot(sample_x, predicted_y, 'gx', label='Sample Point', markersize=10)
# Plot settings
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Simple Linear Regression')
plt.legend()
plt.grid(True)
plt.show()
```

Output :

```
======= RESTART: C:/Users/M.Hema Siri Ramya/OneDrive/Desktop/ML lab/ml.py
1 2 3 4 5
1 2 2.8 4.5 5.1
```



5. Implementation of Multiple Linear Regression for House Price Prediction using sklearn

---

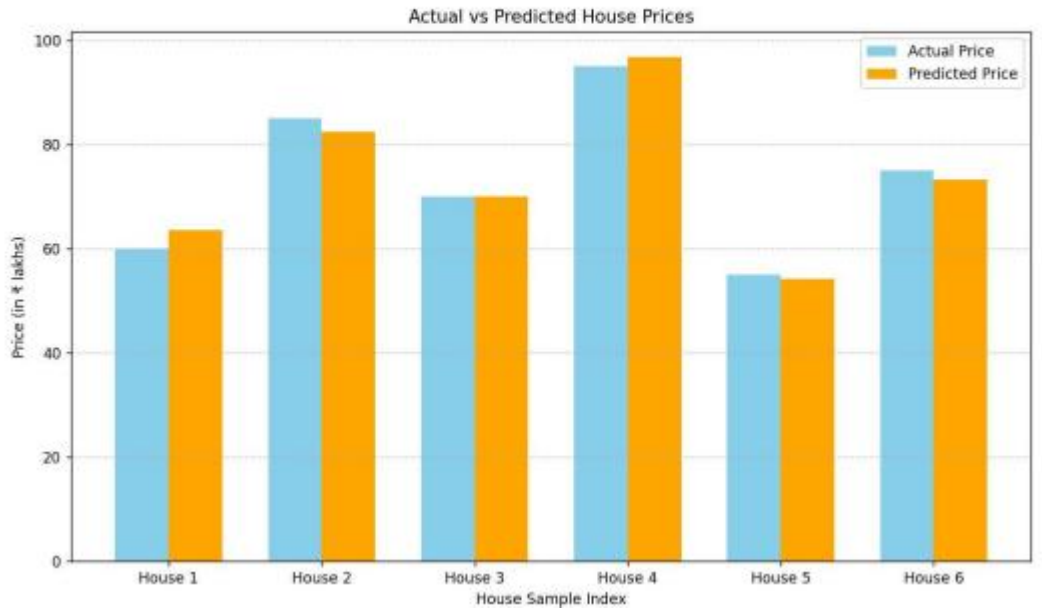
💻 **Code:**

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
# Features: [Area (sqft), Bedrooms, Age of house (years)]
X = np.array([
    [1500, 3, 10],
    [2000, 4, 5],
    [1700, 3, 8],
    [2500, 4, 2],
    [1400, 2, 15],
    [1800, 3, 7]
])
# Target: House prices (in ₹ lakhs)
y = np.array([60, 85, 70, 95, 55, 75])
model = LinearRegression()
model.fit(X, y)
y_pred = model.predict(X)
mse = mean_squared_error(y, y_pred)
print(f"Mean Squared Error (MSE): {mse:.2f}")
new_house = np.array([[2300, 4, 3]])
predicted_price = model.predict(new_house)
print(f"Predicted price for [2300 sqft, 4 bedrooms, 3 yrs old]: ₹{predicted_price[0]:.2f} lakhs")
plt.figure(figsize=(10, 6))
```

```
bar_width = 0.35

index = np.arange(len(y))

plt.bar(index, y, bar_width, label='Actual Price', color='skyblue')

plt.bar(index + bar_width, y_pred, bar_width, label='Predicted Price', color='orange')

plt.xlabel('House Sample Index')

plt.ylabel('Price (in ₹ lakhs)')

plt.title('Actual vs Predicted House Prices')

plt.xticks(index + bar_width / 2, [f'House {i+1}' for i in range(len(y))])

plt.legend()

plt.grid(axis='y', linestyle='--', alpha=0.7)

plt.tight_layout()

plt.show()
```

Output:

```
Mean Squared Error (MSE): 4.41
Predicted price for [2300 sqft, 4 bedrooms, 3 yrs old]: ₹91.18 lakhs
```



6. Implementation of Decision Tree Using sklearn and Its Parameter Tuning

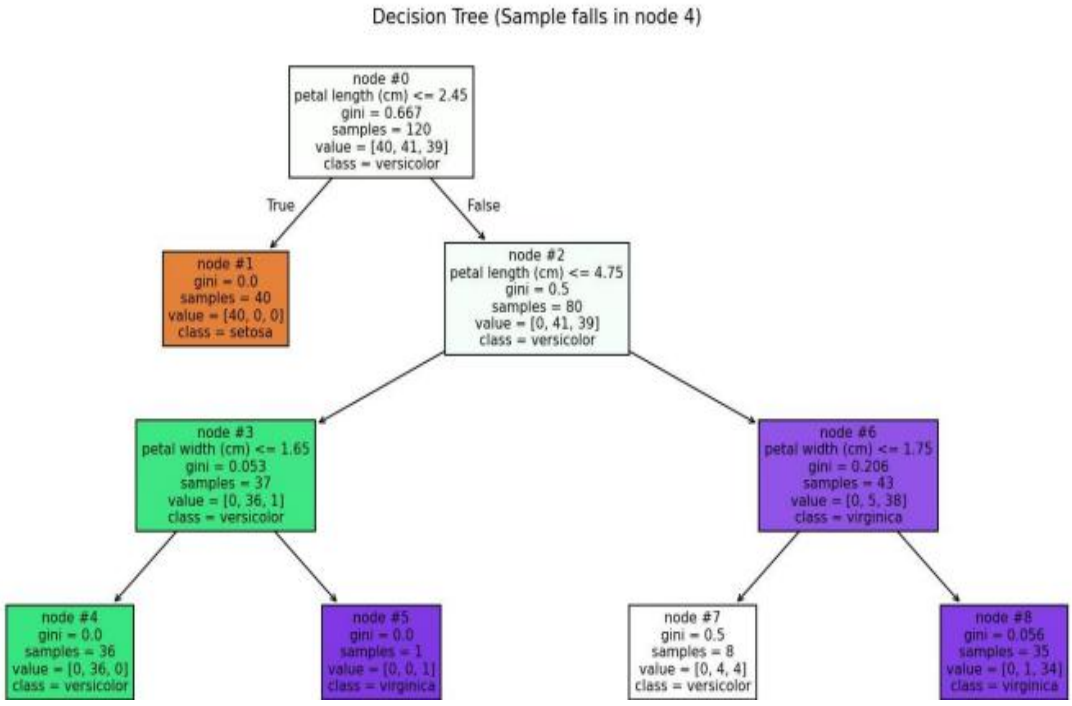
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
import numpy as np

# Load Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize Decision Tree with base parameters
dt = DecisionTreeClassifier(criterion='gini', random_state=42)

# Parameter grid for tuning
params = {
    'max_depth': [2, 3, 5],
    'criterion': ['gini', 'entropy']
}

# Grid Search for best parameters
grid = GridSearchCV(dt, params, cv=5)
grid.fit(X_train, y_train)
```

```
# Best Decision Tree model
best_dt = grid.best_estimator_
y_pred = best_dt.predict(X_test)

# Evaluate performance
print('Accuracy: ', accuracy_score(y_test, y_pred))
print('Best Parameters: ', grid.best_params_)
# Sample to test
sample = np.array([[5.1, 3.5, 3.65, 0.2]])
predicted_class = best_dt.predict(sample)[0]
predicted_name = iris.target_names[predicted_class]
print(f"Sample: {sample}")
print(f"Predicted Class: {predicted_class} ({predicted_name})")
# Trace the sample's path through the tree
node_indicator = best_dt.decision_path(sample)
leaf_id = best_dt.apply(sample)[0]
print(f"Sample reached leaf node: {leaf_id}")
print(f"Sample path: {node_indicator}")
node_index = node_indicator.indices
print('Path:', end='')
for i in node_index:
    print(str(i) + '->', end='')
print('reached')
# Plot the decision tree and highlight the node
plt.figure(figsize=(16, 5))
plot_tree(
    best_dt,
```

```
    filled=True,
    feature_names=iris.feature_names,
    class_names=iris.target_names,
    node_ids=True
)
plt.title(f"Decision Tree (Sample falls in node {leaf_id})")
plt.show()
```

Output:

```
Accuracy:  1.0
Best Parameters:  {'criterion': 'gini', 'max_depth': 3}
Sample: [[5.1  3.5  3.65 0.2 ]]
Predicted Class: 1 (versicolor)
Sample reached leaf node: 4
Sample path: <Compressed Sparse Row sparse matrix of dtype 'int64'
        with 4 stored elements and shape (1, 9)>
  Coords        Values
  (0, 0)        1
  (0, 2)        1
  (0, 3)        1
  (0, 4)        1
Path:0->2->3->4->reached
```



7. Implementation of KNN Using sklearn with Parameter Tuning

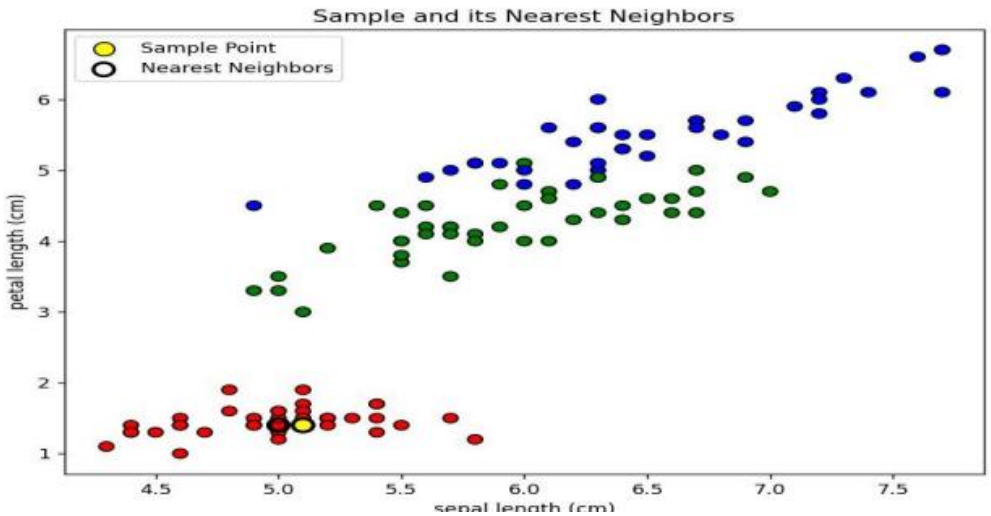
```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import accuracy_score
from sklearn.neighbors import KNeighborsClassifier
import matplotlib.pyplot as plt

# Load the Iris dataset
data = load_iris()
X = data.data
y = data.target

# Split the dataset into training and testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train KNN with default parameters
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print('Accuracy before tuning: ', accuracy)

# Grid Search for best parameters
param_grid = {
    'n_neighbors': [3, 5, 7, 9],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan']
```

```
}

grid = GridSearchCV(
    estimator=KNeighborsClassifier(),
    param_grid=param_grid,
    cv=5
)
grid.fit(X_train, y_train)

print('Best Parameters: ', grid.best_params_)
print('Best Cross-Validation Score: ', grid.best_score_)

# Use best estimator from grid search
best_knn_model = grid.best_estimator_
y_pred_tuned = best_knn_model.predict(X_test)
tuned_accuracy = accuracy_score(y_test, y_pred_tuned)
print('Accuracy after parameter tuning: ', tuned_accuracy)

# Sample prediction
sample = np.array([[5.1, 3.5, 1.4, 0.2]])
predicted_class = best_knn_model.predict(sample)
print('Predicted Class: ', data.target_names[predicted_class[0]])

# Visualization of nearest neighbors
from matplotlib.colors import ListedColormap

feature_x = 0  # sepal length
feature_y = 2  # petal length
```

```
X_vis = X_train[:, [feature_x, feature_y]]
y_vis = y_train
cmap_bold = ListedColormap(['red', 'green', 'blue'])

# Retrain KNN on 2D features
knn_vis = KNeighborsClassifier(**grid.best_params_)
knn_vis.fit(X_vis, y_vis)

sample_2d = sample[:, [feature_x, feature_y]]
neighbors = knn_vis.kneighbors(sample_2d, return_distance=False)

# Plot
plt.figure(figsize=(8, 6))
plt.scatter(X_vis[:, 0], X_vis[:, 1], c=y_vis, cmap=cmap_bold, edgecolor='k', s=50)
plt.scatter(sample_2d[0, 0], sample_2d[0, 1], c='yellow', edgecolor='k', s=100, label='Sample Point')
plt.scatter(
    X_vis[neighbors[0], 0], X_vis[neighbors[0], 1],
    facecolors='none', edgecolors='black',
    s=100, linewidths=2, label='Nearest Neighbors'
)
plt.xlabel(data.feature_names[feature_x])
plt.ylabel(data.feature_names[feature_y])
plt.title('Sample and its Nearest Neighbors')
plt.legend()
plt.show()
```

Output:

```
Accuracy before tuning:  1.0

Best Parameters:  {'metric': 'euclidean', 'n_neighbors': 3, 'weights': 'uniform'
}
Best Cross-Validation Score:  0.9583333333333334
Accuracy after parameter tuning:  1.0
Predicted Class:  setosa
```



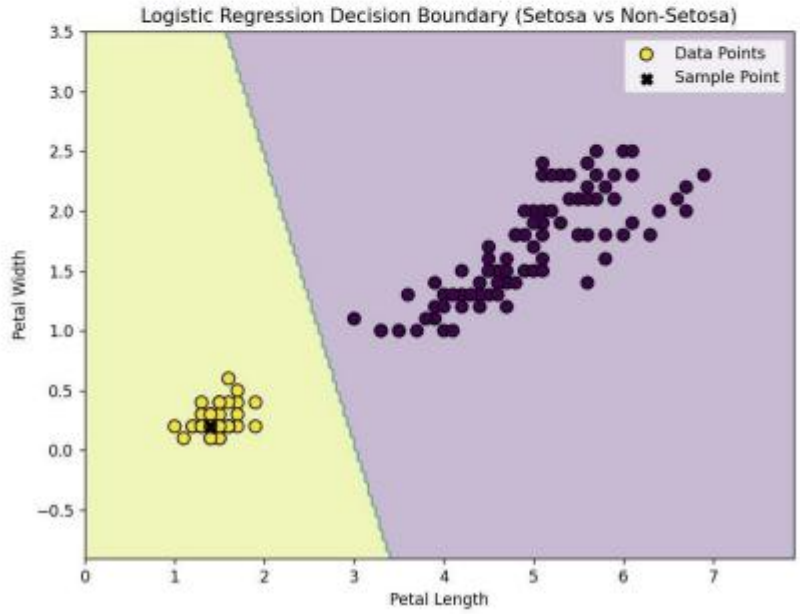
8. Implementation of Logistic Regression using sklearn

```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import load_iris

from sklearn.linear_model import LogisticRegression

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score, classification_report


# Load the Iris dataset

iris = load_iris()

X = iris.data

y = iris.target
```

```
# Binary classification: Setosa = 1, Others = 0
y_binary = (y == 0).astype(int)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y_binary, test_size=0.2,
random_state=42)

# Model training
model = LogisticRegression()
model.fit(X_train, y_train)

# Evaluation
print("Model Evaluation:")
y_pred = model.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))
print("Classification Report:\n", classification_report(y_test, y_pred))

# Sample testing
sample = [[5.1, 3.5, 1.4, 0.2]]
predicted_class = model.predict(sample)
probability = model.predict_proba(sample)

print("Sample Evaluation:")
print("Predicted class (0 = Non-Setosa, 1 = Setosa):", predicted_class[0])
print("Probability [Non-Setosa, Setosa]:", probability[0])

# --------------------
# 🌸 Visualization using petal length and width only
```

```
X_plot = X[:, 2:4]
y_plot = y_binary
sample_plot = [sample[0][2], sample[0][3]]  # Petal length and width of sample

# Train model again for plotting (using 2 features only)
model_plot = LogisticRegression()
model_plot.fit(X_plot, y_plot)

# Create mesh grid for decision boundary
x_min, x_max = X_plot[:, 0].min() - 1, X_plot[:, 0].max() + 1
y_min, y_max = X_plot[:, 1].min() - 1, X_plot[:, 1].max() + 1
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 200),
                     np.linspace(y_min, y_max, 200))

Z = model_plot.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Plot
plt.figure(figsize=(8, 6))
plt.contourf(xx, yy, Z, alpha=0.3, cmap='viridis')
plt.scatter(X_plot[:, 0], X_plot[:, 1], c=y_plot, edgecolor='k', cmap='viridis', s=60, label='Data Points')
plt.scatter(sample_plot[0], sample_plot[1], color='black', s=50, marker='X', label='Sample Point')
plt.xlabel("Petal Length")
plt.ylabel("Petal Width")
plt.title("Logistic Regression Decision Boundary (Setosa vs Non-Setosa)")
plt.legend()
plt.show()
```

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

           0       1.00      1.00      1.00        20
           1       1.00      1.00      1.00        10

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30

Sample Evaluation:
Predicted class (0 = Non-Setosa, 1 = Setosa): 1
Probability [Non-Setosa, Setosa]: [0.02010505 0.97989495]
```



9. Implementation of K-Means Clustering using sklearn

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

# Generating synthetic dataset with 4 clusters
x, y = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)
```

```
# Before clustering
plt.scatter(x[:,0], x[:,1], s=50)
plt.title('Dataset before clustering')
plt.show()

# K-Means clustering
kmeans = KMeans(n_clusters=4)
kmeans.fit(x)
centers = kmeans.cluster_centers_
labels = kmeans.labels_

# Visualizing the clustered output
plt.scatter(x[:,0], x[:,1], c=labels, s=50, cmap='viridis')
plt.scatter(centers[:,0], centers[:,1], c='red', s=200, alpha=0.75, marker='X')

# Adding cluster numbers near centers
for i, center in enumerate(centers):
    plt.text(center[0], center[1], f'Cluster {i}', color='black', fontsize=17, ha='left', va='top')

plt.title('K-Means Clustering with sample testing')

# --------------------
# Sample Testing
sample = np.array([[0.5, 1.5], [3.0, 6.0], [-1.5, 2.5]])
plt.scatter(sample[0, 0], sample[0, 1], c='brown', s=200, marker='P')

# Predicting clusters for new samples
```

```
predicted_labels = kmeans.predict(sample)

for i, new_sample in enumerate(sample):
    print(f'Sample {new_sample} : {predicted_labels[i]}')

# Print cluster centers
print('Centers: ', centers)

plt.show()
```

Output: