1 Study of Python Basic Libraries such as Numpy and Scipy

Study of Python Basic Libraries – NumPy and SciPy

Python offers a powerful ecosystem of libraries for scientific and numerical computing. Two of the most essential ones in the context of **machine learning and data science** are **NumPy** and **SciPy**.

🔷 1. NumPy (Numerical Python)

Overview:
NumPy is the backbone of numerical computing in Python. It enables efficient manipulation of large arrays and matrices, which are foundational structures in machine learning and data science.

🔧 Key Features for ML:

- **Efficient Array Computation:** Faster than standard Python lists; used for storing datasets and model parameters.
- **Broadcasting:** Enables operations between arrays of different shapes — useful for feature scaling and normalization.
- **Vectorized Operations:** Speeds up training by eliminating loops in data preprocessing and mathematical operations.
- **Random Module:** Used for shuffling data, initializing weights, and generating synthetic datasets.

✨ Example

Program 1: Create a NumPy array and perform basic operations

```
import numpy as np

arr = np.array([10, 20, 30, 40, 50])
print("Original Array:", arr)
print("Mean:", np.mean(arr))
print("Standard Deviation:", np.std(arr))
```

Output:

```
Original Array: [10 20 30 40 50]
Mean: 30.0
Standard Deviation: 14.142135623730951
```

Program 2: Matrix multiplication using NumPy

```
import numpy as np

A = np.array([[1, 2], [3, 4]])
B = np.array([[2, 0], [1, 2]])
C = np.dot(A, B)
print("Matrix Multiplication:\n", C)
```

Output:

```
Matrix Multiplication:
 [[ 4  4]
```

```
 [10  8]]
```

🔷 2. SciPy (Scientific Python)

Overview:
SciPy extends NumPy by adding a collection of algorithms and high-level commands for **scientific and technical computing**. It is widely used for optimization, interpolation, signal processing, and statistical analysis — all critical in ML pipelines.

🔧 Key Features for ML:

- **`scipy.optimize`:** Used in training models where custom loss functions are minimized.
- **`scipy.stats`:** Statistical tests, distributions, and random sampling — essential for hypothesis testing, feature analysis, and data exploration.
- **`scipy.spatial`:** Tools for spatial data, distances, and clustering.
- **`scipy.linalg`:** Efficient solvers for matrix operations (used internally in many ML algorithms).

✨ Example

Program 1: Integration using `scipy.integrate`

```
from scipy import integrate

result, _ = integrate.quad(lambda x: x**2, 0, 3)
print("Integral of x^2 from 0 to 3:", result)
```

Output:

```
Integral of x^2 from 0 to 3: 9.0
```

Program 2: Find root of equation using `scipy.optimize`

```
python
Copy code
from scipy import optimize

root = optimize.root(lambda x: x**2 - 9, x0=1)
print("Root of x^2 - 9:", root.x)
```

Output:

```
less
Copy code
Root of x^2 - 9: [3.]
```

🤔 Why These Libraries Matter in ML:

Use Case	NumPy	SciPy
Data preprocessing	Feature scaling, reshaping, slicing	Statistical analysis, distribution fitting
Model training	Vectorized gradient descent	Loss function optimization
Feature engineering	Matrix transformations	Signal and image feature extraction
Evaluation & validation	Metrics computation, arrays	Hypothesis testing, confidence intervals

---

💡 In Summary:

- **NumPy** provides **fast, flexible data structures** (arrays, matrices) and basic math operations.
- **SciPy** adds **advanced scientific tools** ideal for tasks like optimization, statistics, and signal processing.
- Together, they form the **foundation** of the ML ecosystem, often used beneath higher-level libraries like **scikit-learn**, **TensorFlow**, and **PyTorch**.

---

2 Study of Python Libraries for ML application such as Pandas and Matplotlib.

Study of Python Libraries for ML Applications: Pandas and Matplotlib

Python offers robust libraries that play a crucial role in **data analysis**, **visualization**, and **machine learning workflows**. Two such libraries are **Pandas** and **Matplotlib**. While Pandas is essential for handling and preprocessing data, Matplotlib is a powerful tool for visualizing it.

🐼 Pandas – *Python Data Analysis Library*

📌 Overview:

Pandas provides **high-level data structures** like `Series` (1D) and `DataFrame` (2D) that make it easy to handle structured data. It is especially useful for **data cleaning**, **preprocessing**, and **exploratory data analysis (EDA)** — key steps in any ML pipeline.

✅ Program 1: Creating and accessing a DataFrame

```
python
Copy code
import pandas as pd

data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 22],
    'Score': [85, 90, 78]
}

df = pd.DataFrame(data)
print("DataFrame:\n", df)
print("Names column:\n", df['Name'])
```

🔍 Explanation:

- `pd.DataFrame()` creates a 2D data structure.
- `df['Name']` accesses a specific column.

Output:

```
yaml
Copy code
DataFrame:
      Name  Age  Score
0   Alice   25     85
1     Bob   30     90
2  Charlie   22     78

Names column:
0     Alice
1       Bob
2    Charlie
Name: Name, dtype: object
```

✅ Program 2: Filtering and descriptive statistics

```
python
Copy code
print("People older than 24:\n", df[df['Age'] > 24])
```

```
print("Average Score:", df['Score'].mean())
```

🔍 Explanation:

- Filtering rows based on a condition.
- `mean()` calculates the average score — useful for analysis.

Output:

```
yaml
Copy code
People older than 24:
    Name  Age  Score
0  Alice   25     85
1    Bob   30     90

Average Score: 84.33333333333333
```

📊 Matplotlib – *Data Visualization Library*

📌 Overview:

Matplotlib is a 2D plotting library used for **visualizing data**. It is highly customizable and works well with Pandas and NumPy. Visualization is a key part of ML to understand data patterns, correlations, and trends.

✅ Program 1: Line plot of scores

```
python
Copy code
import matplotlib.pyplot as plt

names = df['Name']
scores = df['Score']

plt.plot(names, scores, marker='o')
plt.title("Student Scores")
plt.xlabel("Name")
plt.ylabel("Score")
plt.grid(True)
plt.show()
```

🔍 Explanation:

- Creates a simple line graph showing how scores vary among students.
- `plt.show()` displays the plot.

✅ Program 2: Bar chart of scores

```
python
Copy code
plt.bar(names, scores, color='skyblue')
plt.title("Score Comparison")
plt.xlabel("Name")
plt.ylabel("Score")
plt.show()
```

🔍 Explanation:

- Bar charts are great for comparing categorical data (e.g., student names vs. scores).

✅ Program 3: Pie chart of scores

```
python
Copy code
plt.pie(scores, labels=names, autopct='%1.1f%%')
plt.title("Score Distribution")
plt.show()
```

🔍 Explanation:

- Pie charts show the proportion of total each student's score contributes.

✅ Conclusion

- **Pandas** is ideal for handling, cleaning, and analyzing datasets — critical for ML model preparation.
- **Matplotlib** enables clear visual representations of data, making trends and insights more accessible.
- Together, they form the foundation of **data wrangling and visualization**, essential components in **machine learning workflows**.

---

3 Write a Python program to implement Simple Linear Regression.

Theory

◈ What is Simple Linear Regression?

Simple Linear Regression is a supervised learning algorithm that models the relationship between a **dependent variable (y)** and a **single independent variable (x)** using a straight line (called the regression line).

◈ Mathematical Equation:

$y=m \cdot x+c$

Where:

- `y` is the predicted output.
- `x` is the input feature.
- `m` is the slope (coefficient).
- `c` is the intercept (bias).

The goal is to find the best-fitting line that minimizes the error between predicted and actual values.

□ Python Code

```
python
Copy code
# Simple Linear Regression using scikit-learn

import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

# Input data: Hours studied vs Scores obtained
X = np.array([[1], [2], [3], [4], [5]])  # Independent variable (Hours)
y = np.array([1.5, 3.2, 4.5, 3.8, 5.0])  # Dependent variable (Scores)

# Splitting into training and test sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Creating the Linear Regression model
model = LinearRegression()

# Fitting the model on training data
model.fit(X_train, y_train)

# Predicting on test data
```

```
y_pred = model.predict(X_test)

# Displaying model parameters
print("Slope (Coefficient):", model.coef_[0])
print("Intercept:", model.intercept_)

# Visualization
plt.scatter(X_train, y_train, color='blue', label='Training Data')
plt.plot(X_train, model.predict(X_train), color='red', label='Regression
Line')
plt.xlabel('Hours Studied')
plt.ylabel('Scores')
plt.title('Simple Linear Regression')
plt.legend()
plt.grid(True)
plt.show()

# Showing prediction results
print("\nTest Data Predictions:")
for i in range(len(X_test)):
    print(f"Input: {X_test[i][0]} hours -> Predicted Score:
{round(y_pred[i], 2)} | Actual Score: {y_test[i]}")
```

✅ Sample Output

```
plaintext
Copy code
Slope (Coefficient): 0.85
Intercept: 1.5749999999999997

Test Data Predictions:
Input: 2 hours -> Predicted Score: 3.27 | Actual Score: 3.2
```

🔹 The regression line is plotted on the training data, showing how well the model fits.

📌 Conclusion

- This program demonstrates **Simple Linear Regression** using Python's **scikit-learn** library.
- It shows how we can **predict** outcomes (like exam scores) based on a **single feature** (hours studied).
- You can extend this to real-world datasets with similar numeric relationships.

---

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

Theory

◈ What is Multiple Linear Regression?

Multiple Linear Regression models the relationship between one **dependent variable (y)** and **two or more independent variables (x_1, x_2, ..., x_n)**.

◈ Equation:

y=b0+b1x1+b2x2+...+bnxn

Where:

- y = predicted house price
- $x_1, x_2, \ldots, x_n$ = features (like size, number of bedrooms, age, etc.)
- b_0 = intercept
- $b_1, b_2, \ldots, b_n$ = regression coefficients

□ Python Code

```
python
Copy code
# Multiple Linear Regression using sklearn for House Price Prediction

import numpy as np
import pandas as pd
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

# ----------------------------
# Step 1: Dataset Preparation
# ----------------------------
# Creating a sample dataset
# Features: [Size (sqft), No. of Bedrooms, Age of House]
data = {
    'Size': [1000, 1500, 1800, 2400, 3000, 3500, 4000, 4100, 4700, 5000],
    'Bedrooms': [2, 3, 3, 4, 4, 5, 5, 5, 6, 6],
    'Age': [10, 5, 3, 8, 12, 6, 5, 3, 2, 4],
    'Price': [250000, 330000, 360000, 420000, 480000, 530000, 590000,
610000, 670000, 700000]
}

df = pd.DataFrame(data)

# ----------------------------
```

```
# Step 2: Splitting into Features and Target
# ----------------------------
X = df[['Size', 'Bedrooms', 'Age']]  # Independent variables
y = df['Price']                      # Dependent variable

# ----------------------------
# Step 3: Train-Test Split
# ----------------------------
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# ----------------------------
# Step 4: Model Training
# ----------------------------
model = LinearRegression()
model.fit(X_train, y_train)

# ----------------------------
# Step 5: Predictions
# ----------------------------
y_pred = model.predict(X_test)

# ----------------------------
# Step 6: Model Evaluation
# ----------------------------
print("Intercept (b0):", round(model.intercept_, 2))
print("Coefficients (b1, b2, b3):", model.coef_)
print("\nMean Squared Error:", round(mean_squared_error(y_test, y_pred),
2))
print("R² Score:", round(r2_score(y_test, y_pred), 4))

# ----------------------------
# Step 7: Visual Comparison
# ----------------------------
plt.figure(figsize=(10,6))
plt.plot(np.arange(len(y_test)), y_test.values, marker='o', label='Actual
Price')
plt.plot(np.arange(len(y_test)), y_pred, marker='x', label='Predicted
Price')
plt.title('Actual vs Predicted House Prices')
plt.xlabel('Test Sample Index')
plt.ylabel('House Price')
plt.legend()
plt.grid(True)
plt.show()
```

✅ Sample Output (Will Vary Slightly)

```
plaintext
Copy code
Intercept (b0): 36113.29
Coefficients (b1, b2, b3): [ 115.55 10255.87 -2415.53]

Mean Squared Error: 8417716.47
R² Score: 0.9746
```

📌 Conclusion

- This code uses **Multiple Linear Regression** to predict house prices based on multiple features.
- The **R^2 score** close to 1 indicates a good fit.
- Can be extended with more features like **location**, **distance to amenities**, etc., for better accuracy.

---

5 Implementation of Decision tree using sklearn.

Overview:

A **Decision Tree** is a supervised learning algorithm used for both **classification** and **regression** tasks. It splits the data into subsets based on the value of input features and makes decisions in a tree-like model of decisions and their possible consequences.

✅ Key Concepts:

- **Root Node**: The first decision node.
- **Decision Node**: Nodes where the data is split.
- **Leaf Node**: Terminal node that gives the output label.
- **Gini Index / Entropy**: Used to decide the best split.

✅ Advantages:

- Easy to understand and visualize
- Handles both numerical and categorical data
- Requires little data preprocessing

✅ Common Criteria:

- **Gini Impurity** (default in sklearn):

  Gini=1−∑i=1npi2Gini = 1 - \sum_{i=1}^{n} p_i^2Gini=1−i=1∑npi2

- **Entropy**:

  Entropy=−∑i=1npilog⁡2(pi)Entropy = - \sum_{i=1}^{n} p_i \log_2(p_i)Entropy=−i=1∑npilog2(pi)

💻 Implementation Using `scikit-learn`

Dataset: Iris Dataset

This classic dataset has 3 classes of iris plants (Setosa, Versicolor, Virginica) and 4 features (sepal length, sepal width, petal length, petal width).

🔧 Code:

```
# Step 1: Import libraries
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, accuracy_score,
confusion_matrix
from sklearn import tree
import matplotlib.pyplot as plt

# Step 2: Load dataset
iris = load_iris()
X = iris.data
y = iris.target

# Step 3: Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Step 4: Initialize and train model
clf = DecisionTreeClassifier(criterion='gini', random_state=42)
clf.fit(X_train, y_train)

# Step 5: Make predictions
y_pred = clf.predict(X_test)

# Step 6: Evaluate model
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
print("\nAccuracy Score:", accuracy_score(y_test, y_pred))

# Step 7: Visualize the tree
plt.figure(figsize=(12,8))
tree.plot_tree(clf, filled=True, feature_names=iris.feature_names,
class_names=iris.target_names)
plt.title("Decision Tree - Iris Dataset")
plt.show()
```

📊 Sample Output:

```
Confusion Matrix:
[[10  0  0]
 [ 0  9  1]
 [ 0  0 10]]
```

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

      setosa       1.00      1.00      1.00        10
  versicolor       1.00      0.90      0.95        10
   virginica       0.91      1.00      0.95        10

    accuracy                           0.97        30

Accuracy Score: 0.9667
```

📝 Conclusion:

Decision Trees offer a simple yet powerful approach for classification tasks. Using the Iris dataset, we can see how feature-based decisions can classify flower species accurately.

---

6 Implementation of KNN using sklearn.

Theory

K-Nearest Neighbors (KNN) is a **supervised learning** algorithm used for both **classification** and **regression** problems. However, it is mostly used for classification.

🔷 How It Works:

1. **Store Training Data:** KNN is a **lazy learner**, meaning it does not learn a model during training. It simply stores the training dataset.
2. **Compute Distance:** When a new input needs to be classified, KNN computes the **distance** between the input and all points in the training data.
3. **Find Neighbors:** It identifies the **'k' nearest neighbors** based on the smallest distances.
4. **Voting (Classification):** For classification, it assigns the class which is **most frequent** among the nearest neighbors.
5. **Averaging (Regression):** For regression, it returns the **average value** of the target variable of the nearest neighbors.

□ Common Distance Metrics:

- **Euclidean Distance:**

  $$d(p,q) = \sqrt{\sum_{i=1}^n (p_i - q_i)^2}$$

- **Manhattan Distance:**

  $$d(p,q) = \sum_{i=1}^n |p_i - q_i|$$

- **Minkowski Distance (general form):**

✅ Advantages:

- Simple and easy to implement.
- No training phase (no model building).
- Works well with a small dataset with low dimensionality.

❌ Disadvantages:

- Slow for large datasets.
- Sensitive to irrelevant or redundant features.
- Requires feature scaling (like normalization).
- Performance depends heavily on the choice of **k**.

🔧 Hyperparameters:

- `n_neighbors`: Number of nearest neighbors (k).
- `metric`: Distance metric (`'euclidean'`, `'manhattan'`, `'minkowski'`, etc.).
- `weights`: `'uniform'` (all neighbors equal) or `'distance'` (closer neighbors have more influence).

💻 Implementation using scikit-learn (sklearn)

```
# Step 1: Import required libraries
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, classification_report

# Step 2: Load the dataset
iris = load_iris()
X = iris.data  # Features
y = iris.target  # Labels
```

```
# Step 3: 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)

# Step 4: Initialize and train KNN model
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

# Step 5: Make predictions
y_pred = knn.predict(X_test)

# Step 6: Evaluate the model
print("Accuracy:", accuracy_score(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

📊 Output Example:

```
Accuracy: 1.0

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         9
           1       1.00      1.00      1.00        10
           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
```

📌 Conclusion

KNN is a foundational classification algorithm that's easy to understand and implement. However, its performance can degrade with high-dimensional or large datasets, and it's important to normalize features and tune the value of **k** carefully.

---

7 Write a program to Implement Support Vector Machines.

Theory

Support Vector Machines (SVM) are supervised learning models used for classification and regression tasks. The core idea is to find the optimal hyperplane that best separates data points of different classes in the feature space.

🔹 Key Concepts:

- **Hyperplane**: A decision boundary that separates different classes.
- **Support Vectors**: Data points that are closest to the hyperplane and influence its position and orientation.
- **Margin**: The distance between the hyperplane and the nearest data points from each class. SVM aims to maximize this margin.

🔹 Types of SVM:

- **Linear SVM**: Used when data is linearly separable.
- **Non-Linear SVM**: Utilizes kernel functions to handle non-linearly separable data.

🔹 Kernel Functions:

- **Linear Kernel**: Suitable for linearly separable data.
- **Polynomial Kernel**: Handles polynomial relationships.
- **Radial Basis Function (RBF) Kernel**: Effective for non-linear data.
- **Sigmoid Kernel**: Similar to neural networks.

✅ Advantages:

- Effective in high-dimensional spaces.
- Works well with clear margin of separation.
- Versatile with different kernel functions.

❌ Disadvantages:

- Not suitable for large datasets due to high training time.
- Less effective when the number of features exceeds the number of samples.
- Requires careful tuning of parameters.

💻 Practical Implementation: SVM on Pima Indians Diabetes Dataset

We'll use the **Pima Indians Diabetes Dataset** from Kaggle, which contains medical data for predicting the onset of diabetes.

📎 Dataset Source:

You can access the dataset here: Pima Indians Diabetes Dataset on Kaggle

🔗 Steps:

1. **Import Libraries**:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import classification_report, confusion_matrix,
accuracy_score
```

2. **Load Dataset**:

```
# Load the dataset
df = pd.read_csv('diabetes.csv')  # Ensure the CSV file is in your working
directory
```

3. **Explore and Preprocess Data**:

```
python
Copy code
# Check for missing values
print(df.isnull().sum())

# Separate features and target
X = df.drop('Outcome', axis=1)
y = df['Outcome']

# Feature scaling
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

4. **Split Data into Training and Testing Sets**:

```
python
Copy code
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y,
test_size=0.2, random_state=42)
```

5. **Train SVM Model**:

```
python
Copy code
# Initialize the SVM classifier with RBF kernel
svm_classifier = SVC(kernel='rbf', C=1, gamma='scale')

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

6. **Make Predictions and Evaluate Model**:

```
python
Copy code
# Predict on test data
y_pred = svm_classifier.predict(X_test)
```

```
# Evaluate the model
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
print("\nAccuracy Score:", accuracy_score(y_test, y_pred))
```

📊 Sample Output:

```
lua
Copy code
Confusion Matrix:
 [[90 10]
 [15 39]]

Classification Report:
              precision    recall  f1-score   support

           0       0.86      0.90      0.88       100
           1       0.80      0.72      0.76        54

    accuracy                           0.84       154
   macro avg       0.83      0.81      0.82       154
weighted avg       0.84      0.84      0.84       154

Accuracy Score: 0.84
```

Note: The above output is illustrative. Actual results may vary based on data splits and parameter tuning.

📝 Conclusion:

Support Vector Machines are powerful tools for classification tasks, especially when dealing with high-dimensional data. By selecting appropriate kernel functions and tuning hyperparameters, SVMs can achieve high accuracy and generalization performance.

---

8 Implementation of Logistic Regression using sklearn

Theory

Logistic Regression is a supervised learning algorithm used for classification tasks. It models the probability that a given input point belongs to a certain class. Unlike linear regression, which predicts continuous outputs, logistic regression predicts probabilities using the logistic (sigmoid) function.

🔷 Sigmoid Function:

σ(z)=11+e−z\sigma(z) = \frac{1}{1 + e^{-z}}σ(z)=1+e−z1

This function maps any real-valued number into the (0, 1) interval, making it suitable for modeling probabilities.

🔹 Decision Boundary:

In logistic regression, the decision boundary is determined by the equation:

β0+β1x1+β2x2+…+βnxn=0\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n = 0β0 +β1x1+β2x2+…+βnxn=0

Where:

- β0\beta_0β0 is the intercept,
- β1,β2,…,βn\beta_1, \beta_2, \ldots, \beta_nβ1,β2,…,βn are the coefficients for the features x1,x2,…,xnx_1, x_2, \ldots, x_nx1,x2,…,xn.

🔹 Cost Function:

Logistic regression uses the **log loss** or **binary cross-entropy** as its cost function:

J(θ)=−1m∑i=1m[y(i)log⁡(hθ(x(i)))+(1−y(i))log⁡(1−hθ(x(i)))]J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right]J(θ)=−m1i=1∑m[y(i)log(hθ(x(i)))+(1−y(i))log(1−hθ(x(i)))]

Where:

- mmm is the number of training examples,
- y(i)y^{(i)}y(i) is the actual label,
- hθ(x(i))h_\theta(x^{(i)})hθ(x(i)) is the predicted probability.

💻 Implementation using scikit-learn

We'll use the **Breast Cancer Wisconsin Dataset** for this implementation. This dataset contains features computed from digitized images of breast mass and is used to predict whether the mass is malignant or benign.

🔧 Steps:

1. **Import Libraries**:

```
python
Copy code
import numpy as np
import pandas as pd
from sklearn.datasets import load_breast_cancer
```

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix,
accuracy_score
```

2. **Load Dataset**:

```
python
Copy code
# Load the dataset
data = load_breast_cancer()
X = data.data
y = data.target
```

3. **Split Data into Training and Testing Sets**:

```
python
Copy code
# Split the dataset 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)
```

4. **Feature Scaling**:

```
python
Copy code
# Standardize features by removing the mean and scaling to unit variance
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

5. **Train Logistic Regression Model**:

```
python
Copy code
# Initialize and train the logistic regression model
model = LogisticRegression(max_iter=10000)
model.fit(X_train_scaled, y_train)
```

6. **Make Predictions and Evaluate Model**:

```
python
Copy code
# Predict on the test set
y_pred = model.predict(X_test_scaled)

# Evaluate the model
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
print("\nAccuracy Score:", accuracy_score(y_test, y_pred))
```

📊 Sample Output:

```
lua
```

```
Copy code
Confusion Matrix:
[[71  1]
 [ 2 40]]

Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.99      0.98        72
           1       0.98      0.95      0.96        42

    accuracy                           0.97       114
   macro avg       0.97      0.97      0.97       114
weighted avg       0.97      0.97      0.97       114

Accuracy Score: 0.9736842105263158
```

Note: The actual output may vary slightly due to randomness in data splitting.

📝 Conclusion:

Logistic Regression is a powerful and interpretable algorithm for binary classification tasks. By applying it to the Breast Cancer Wisconsin Dataset, we can effectively predict whether a tumor is malignant or benign based on various features.

---

9 Implementation of K-Means Clustering

Aim:

To implement the K-Means Clustering algorithm and visualize the clusters formed on a dataset.

Theory:

K-Means is an **unsupervised learning algorithm** used for clustering. It partitions `n` observations into `k` clusters where each observation belongs to the cluster with the nearest mean (centroid). The objective is to minimize the sum of squared distances between points and their respective cluster centers.

Algorithm:

1. Choose the number of clusters `k`.
2. Randomly initialize `k` centroids.
3. Repeat until convergence:
    - Assign each point to the nearest centroid.
    - Recalculate the centroids as the mean of all points in the cluster.
4. Stop when the centroids no longer change significantly or after a fixed number of iterations.

Code:

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

# Generate sample dataset
X, y_true = make_blobs(n_samples=300, centers=4, cluster_std=0.60,
random_state=0)

# Visualize initial data
plt.scatter(X[:, 0], X[:, 1], s=50)
plt.title("Original Data")
plt.show()

# Apply KMeans
k = 4
kmeans = KMeans(n_clusters=k)
kmeans.fit(X)
y_kmeans = kmeans.predict(X)

# Visualize the Clusters
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='viridis')
centers = kmeans.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], c='red', s=200, alpha=0.75,
marker='X')
plt.title("Clusters Formed by K-Means")
plt.show()
```

Sample Output:

- **Graph 1:** Original randomly generated data points.

- **Graph 2:** Clusters formed after applying K-Means with centroids marked in red.

Result:

The K-Means Clustering algorithm was successfully implemented, and the data was grouped into 4 clusters based on their features. The centroids of the clusters were correctly identified and visualized.

---

10 Build an Artificial Neural Network by implementing the Back propagation algorithm

Aim:

To implement an Artificial Neural Network (ANN) using the Backpropagation algorithm and test it on a dataset.

Theory:

Artificial Neural Networks (ANNs) are inspired by the structure and function of the human brain. They consist of layers of neurons: an input layer, one or more hidden layers, and an output layer. Each neuron uses an activation function to produce an output.

Backpropagation is a supervised learning algorithm used for training ANNs. It adjusts the weights by calculating the gradient of the loss function with respect to each weight through the chain rule (Gradient Descent).

Algorithm:

1. **Initialize** the weights and biases with small random values.
2. **Forward Propagation**:
    - Compute input to hidden layer.
    - Apply activation function (e.g., sigmoid).
    - Compute input to output layer.
    - Apply activation function.

3. **Compute Error**:
    - Difference between predicted output and actual output.
4. **Backpropagation**:
    - Calculate gradient of error with respect to output weights.
    - Calculate gradient of error with respect to hidden weights.
    - Update weights and biases using gradients.
5. **Repeat** steps 2–4 for a number of epochs.
6. **Test** the trained model using a test dataset.

Code:

```
python
CopyEdit
import numpy as np
from sklearn.datasets import load_iris
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.model_selection import train_test_split

# Activation functions
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# Load and preprocess dataset
iris = load_iris()
X = iris.data
y = iris.target.reshape(-1, 1)

encoder = OneHotEncoder(sparse_output=False)
y_encoded = encoder.fit_transform(y)

scaler = StandardScaler()
X = scaler.fit_transform(X)

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

# Initialize network parameters
input_neurons = X_train.shape[1]
hidden_neurons = 5
output_neurons = y_encoded.shape[1]
learning_rate = 0.1
epochs = 1000

np.random.seed(1)
weights_input_hidden = np.random.rand(input_neurons, hidden_neurons)
bias_hidden = np.zeros((1, hidden_neurons))
weights_hidden_output = np.random.rand(hidden_neurons, output_neurons)
bias_output = np.zeros((1, output_neurons))

# Training process
for epoch in range(epochs):
    hidden_input = np.dot(X_train, weights_input_hidden) + bias_hidden
    hidden_output = sigmoid(hidden_input)
```

```
    final_input = np.dot(hidden_output, weights_hidden_output) +
bias_output
    final_output = sigmoid(final_input)

    error = y_train - final_output
    d_output = error * sigmoid_derivative(final_output)

    error_hidden = d_output.dot(weights_hidden_output.T)
    d_hidden = error_hidden * sigmoid_derivative(hidden_output)

    weights_hidden_output += hidden_output.T.dot(d_output) * learning_rate
    bias_output += np.sum(d_output, axis=0, keepdims=True) * learning_rate
    weights_input_hidden += X_train.T.dot(d_hidden) * learning_rate
    bias_hidden += np.sum(d_hidden, axis=0, keepdims=True) * learning_rate

    if epoch % 100 == 0:
        loss = np.mean(np.square(error))
        print(f"Epoch {epoch} Loss: {loss:.4f}")

# Testing
hidden_test = sigmoid(np.dot(X_test, weights_input_hidden) + bias_hidden)
final_test = sigmoid(np.dot(hidden_test, weights_hidden_output) +
bias_output)

predictions = np.argmax(final_test, axis=1)
actual = np.argmax(y_test, axis=1)

accuracy = np.mean(predictions == actual) * 100
print(f"\nTest Accuracy: {accuracy:.2f}%")
```

Sample Output:

```
Epoch 0 Loss: 0.2601
Epoch 100 Loss: 0.0925
Epoch 200 Loss: 0.0594
...
Epoch 900 Loss: 0.0258

Test Accuracy: 96.67%
```

Result:

An Artificial Neural Network was successfully implemented using the Backpropagation algorithm and tested on the Iris dataset. The model achieved a test accuracy of **approximately 96.67%**.