Practical 1

A. Revisiting Python Basics & Libraries

```
# Importing essential libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

# 1. NUMPY - Creating and manipulating arrays
arr = np.array([1, 2, 3, 4, 5])  # Create a NumPy array
print("NumPy Array:", arr)

# 2. PANDAS - Creating and displaying a DataFrame
data = {'Name': ['Alice', 'Bob'], 'Age': [25, 30]}
df = pd.DataFrame(data)  # Create a Pandas DataFrame
print("\nPandas DataFrame:\n", df)

# 3. MATPLOTLIB - Simple line plot
plt.plot([1, 2, 3], [4, 5, 6])  # Plot x vs y
plt.title("Simple Plot")  # Add title
plt.show()  # Display plot

# 4. SKLEARN - Simple Linear Regression
X = [[1], [2], [3]]  # Features
y = [2, 4, 6]        # Target
model = LinearRegression().fit(X, y)  # Train model
print("\nSklearn Prediction for 4:", model.predict([[4]]))
```

B. Optical Character Recognition (OCR) using Pytesseract

(i) OCR from an Image File

```
from PIL import Image  # To handle images
import pytesseract     # OCR library

# Provide the path to Tesseract executable (if not in PATH)
# pytesseract.pytesseract.tesseract_cmd = r'C:\Program Files\Tesseract-OCR\tesseract.exe'

# Open an image file
img = Image.open("sample_image.png")  # Replace with your image path

# Extract text from the image
text = pytesseract.image_to_string(img)
print("Extracted Text from Image:\n", text)
```

(ii) OCR from a Multi-Page PDF File

```
import pdf2image              # Convert PDF to images
import pytesseract            # OCR library
from PIL import Image         # To handle images
# Convert PDF pages to images (one image per page)
images = pdf2image.convert_from_path("sample.pdf")  # Replace with your PDF path
# Loop through each page/image and extract text
for i, img in enumerate(images):
    text = pytesseract.image_to_string(img)  # Extract text
    print(f"\nText from Page {i+1}:\n", text)  # Print extracted text
```

Key Notes:

Install required libraries first:

```
pip install numpy pandas matplotlib scikit-learn pillow pytesseract pdf2image
```

Practical 2

1. Simple Linear Regression (One Variable)

(A) Gradient Descent Approach

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

# Sample data (X: feature, y: target)
X = np.array([1, 2, 3, 4, 5])
y = np.array([2, 4, 5, 4, 5])

# Initialize parameters (slope 'm' and intercept 'b')
m, b = 0, 0
learning_rate = 0.01
epochs = 1000

# Gradient Descent
for _ in range(epochs):
    y_pred = m * X + b  # Predicted y
    error = y_pred - y  # Error

    # Compute gradients (derivatives of MSE w.r.t 'm' and 'b')
    grad_m = (2/len(X)) * np.sum(error * X)
    grad_b = (2/len(X)) * np.sum(error)

    # Update parameters
    m -= learning_rate * grad_m
    b -= learning_rate * grad_b

print(f"Final Parameters: m = {m:.2f}, b = {b:.2f}")
```

```
# Plotting
plt.scatter(X, y, color='blue', label='Actual Data')
plt.plot(X, m * X + b, color='red', label='Regression Line')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.show()
```

(B) Normal Equation Approach (Closed-form Solution)

```
import numpy as np

X = np.array([1, 2, 3, 4, 5])
y = np.array([2, 4, 5, 4, 5])

# Add a column of 1s for intercept (b)
X_matrix = np.vstack([X, np.ones(len(X))]).T

# Normal Equation: θ = (X^T X)^-1 X^T y
theta = np.linalg.inv(X_matrix.T @ X_matrix) @ X_matrix.T @ y

m, b = theta[0], theta[1]
print(f"Parameters (Normal Eq.): m = {m:.2f}, b = {b:.2f}")
```

2. Multiple Linear Regression (Multiple Variables)

(A) Gradient Descent Approach

```
import numpy as np
# Sample data (X1, X2: features, y: target)
X = np.array([[1, 2], [2, 4], [3, 6], [4, 8], [5, 10]])
y = np.array([3, 5, 7, 9, 11])
# Add bias term (column of 1s)
X_b = np.c_[np.ones((X.shape[0], 1)), X]  # Shape: (5, 3)

# Initialize parameters (theta = [bias, weight1, weight2])
theta = np.zeros(X_b.shape[1])
learning_rate = 0.01
epochs = 1000
# Gradient Descent
for _ in range(epochs):
    y_pred = X_b @ theta  # Predicted y
    error = y_pred - y

    # Compute gradient (MSE derivative)
    gradient = (2/len(X)) * X_b.T @ error
    # Update theta
    theta -= learning_rate * gradient
print("Final Parameters (Gradient Descent):", theta)

# Predict for new data
new_X = np.array([[6, 12], [7, 14]])
new_X_b = np.c_[np.ones((new_X.shape[0], 1)), new_X]
predictions = new_X_b @ theta
print("Predictions:", predictions)
```

(B) Normal Equation Approach (Closed-form Solution)

```
import numpy as np

X = np.array([[1, 2], [2, 4], [3, 6], [4, 8], [5, 10]])

y = np.array([3, 5, 7, 9, 11])

# Add bias term (column of 1s)

X_b = np.c_[np.ones((X.shape[0], 1)), X]

# Normal Equation: θ = (X^T X)^-1 X^T y

theta = np.linalg.inv(X_b.T @ X_b) @ X_b.T @ y

print("Parameters (Normal Eq.):", theta)

# Predict for new data

new_X = np.array([[6, 12], [7, 14]])

new_X_b = np.c_[np.ones((new_X.shape[0], 1)), new_X]

predictions = new_X_b @ theta

print("Predictions:", predictions)
```

Key Notes:

1. **Gradient Descent**:
    - Iteratively updates parameters to minimize the **Mean Squared Error (MSE)**.
    - Requires tuning **learning rate** and **epochs**.
    - Works well for large datasets.
2. **Normal Equation**:
    - Directly computes the optimal parameters using matrix operations.
    - **No iterations** needed, but **slower for large datasets** (due to matrix inversion).
3. **Multiple Linear Regression**:
    - Extends to **n features** by adding columns to X.
    - The **bias term (intercept)** is added as a column of 1s.
4. **Matrix Operations** (@ operator in NumPy):
    - X.T → Transpose of X.
    - np.linalg.inv() → Matrix inverse.
    - A @ B → Matrix multiplication.

Practical 3

1. Manual Implementation (Without Sklearn)

(A) Simple & Multiple Linear Regression (No Regularization)

(Same as previous answer, but consolidated for clarity.)

Gradient Descent Approach

```
import numpy as np

def gradient_descent(X, y, learning_rate=0.01, epochs=1000):
    # Add bias term (column of 1s)
    X_b = np.c_[np.ones((X.shape[0], 1)), X]
    theta = np.zeros(X_b.shape[1])  # Initialize parameters

    for _ in range(epochs):
        y_pred = X_b @ theta
        error = y_pred - y
        gradient = (2 / len(X)) * X_b.T @ error  # MSE derivative
        theta -= learning_rate * gradient

    return theta

# Example Usage:
X = np.array([[1, 2], [2, 4], [3, 6]])  # Multiple features
y = np.array([3, 5, 7])
theta = gradient_descent(X, y)
print("Manual Gradient Descent Coefficients:", theta)
```

Normal Equation Approach

```
def normal_equation(X, y):
    X_b = np.c_[np.ones((X.shape[0], 1)), X]
    theta = np.linalg.inv(X_b.T @ X_b) @ X_b.T @ y
    return theta
theta = normal_equation(X, y)
print("Manual Normal Equation Coefficients:", theta)
```

(B) Linear Regression with L2 Regularization (Ridge Regression)

Gradient Descent with L2 Penalty

```
def ridge_gradient_descent(X, y, alpha=1.0, learning_rate=0.01, epochs=1000):
    X_b = np.c_[np.ones((X.shape[0], 1)), X]
    theta = np.zeros(X_b.shape[1])
    for _ in range(epochs):
        y_pred = X_b @ theta
        error = y_pred - y
        # Gradient with L2 penalty (exclude bias term)
        gradient = (2 / len(X)) * (X_b.T @ error + alpha * np.r_[0, theta[1:]])
        theta -= learning_rate * gradient

    return theta

theta_ridge = ridge_gradient_descent(X, y, alpha=1.0)
print("Manual Ridge Regression Coefficients:", theta_ridge)
```

Normal Equation with L2 Penalty

```
def ridge_normal_equation(X, y, alpha=1.0):
    X_b = np.c_[np.ones((X.shape[0], 1)), X]
    I = np.eye(X_b.shape[1])
    I[0, 0] = 0  # Don't regularize bias term
    theta = np.linalg.inv(X_b.T @ X_b + alpha * I) @ X_b.T @ y
    return theta

theta_ridge = ridge_normal_equation(X, y, alpha=1.0)
print("Manual Ridge (Normal Eq.) Coefficients:", theta_ridge)
```

2. Sklearn Implementation

(A) Simple & Multiple Linear Regression (No Regularization)

```
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(X, y)

print("Sklearn Coefficients:", model.coef_)
print("Sklearn Intercept:", model.intercept_)
```

(B) Ridge Regression (L2 Regularization)

```
from sklearn.linear_model import Ridge

ridge_model = Ridge(alpha=1.0)
ridge_model.fit(X, y)

print("Sklearn Ridge Coefficients:", ridge_model.coef_)
print("Sklearn Ridge Intercept:", ridge_model.intercept_)
```

(C) Lasso Regression (L1 Regularization)

```
from sklearn.linear_model import Lasso

lasso_model = Lasso(alpha=0.1)
lasso_model.fit(X, y)

print("Sklearn Lasso Coefficients:", lasso_model.coef_)
print("Sklearn Lasso Intercept:", lasso_model.intercept_)
```

(D) ElasticNet (L1 + L2 Regularization)

```
from sklearn.linear_model import ElasticNet

elastic_model = ElasticNet(alpha=0.1, l1_ratio=0.5)
elastic_model.fit(X, y)

print("Sklearn ElasticNet Coefficients:", elastic_model.coef_)
print("Sklearn ElasticNet Intercept:", elastic_model.intercept_)
```

Key Notes:

1. **Manual Implementation**:
    - **Gradient Descent**: Iterative optimization (better for large datasets).
    - **Normal Equation**: Direct solution (faster for small datasets).
    - **L2 Regularization (Ridge)**: Adds alpha * theta² penalty to loss.
2. **Sklearn Implementation**:
    - LinearRegression: No regularization.
    - Ridge: L2 regularization (alpha controls strength).
    - Lasso: L1 regularization (sparse solutions).
    - ElasticNet: Mix of L1 + L2.
3. **When to Use Regularization?**
    - **Ridge (L2)**: When features are correlated.
    - **Lasso (L1)**: For feature selection (zeroes out weak features).
    - **ElasticNet**: Balanced L1 + L2 penalty.

Practical 4

1. Bernoulli Naïve Bayes (Binary Features)

Use Case: Spam detection (words present: 1 or absent: 0)

```
# Import libraries
from sklearn.naive_bayes import BernoulliNB
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# Generate synthetic binary data (1000 samples, 10 features)
X, y = make_classification(n_samples=1000, n_features=10, n_classes=2, random_state=42)

# Split data into training (70%) and testing (30%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Initialize and train Bernoulli Naïve Bayes
bnb = BernoulliNB()  # Default settings
bnb.fit(X_train, y_train)  # Train model

# Evaluate accuracy on test data
print("BernoulliNB Accuracy:", bnb.score(X_test, y_test))  # Output: ~0.87
```

2. Multinomial Naïve Bayes (Discrete Counts)

Use Case: Sentiment analysis (word frequencies)

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import CountVectorizer

# Sample dataset: 4 movie reviews
corpus = [“good movie”, “bad movie”, “great film”, “poor film”]
labels = [1, 0, 1, 0]  # 1=Positive, 0=Negative

# Convert text to word counts (e.g., “good movie” → [1, 1, 0, 0, 0])
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(corpus)  # X = word count matrix

# Train Multinomial Naïve Bayes
mnb = MultinomialNB()
mnb.fit(X, labels)

# Predict sentiment for new text
new_text = [“good film”]
X_new = vectorizer.transform(new_text)  # Convert to word counts
print(“Predicted sentiment (1=Positive, 0=Negative):”, mnb.predict(X_new))  # Output: [1]
```

3. Gaussian Naïve Bayes (Continuous Features)

Use Case: Iris flower classification (sepal/petal measurements)

```
from sklearn.naive_bayes import GaussianNB
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

# Load Iris dataset (150 samples, 4 features)
X, y = load_iris(return_X_y=True)  # X = features, y = labels (0,1,2)

# Split data (70% train, 30% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Train Gaussian Naïve Bayes (assumes features are normally distributed)
gnb = GaussianNB()
gnb.fit(X_train, y_train)

# Evaluate accuracy
print("GaussianNB Accuracy:", gnb.score(X_test, y_test))  # Output: ~0.93
```

Key Notes for Easy Recall

1. **BernoulliNB**:
    - Input: Binary data (0 or 1).
    - Example: [1, 0, 1] = "Word1 present, Word2 absent, Word3 present".
2. **MultinomialNB**:
    - Input: Word counts (e.g., [3, 0, 2] = "Word1 appears 3 times, Word2 appears 0 times...").
3. **GaussianNB**:
    - Input: Continuous numbers (e.g., [5.1, 3.5, 1.4, 0.2] = sepal length, width, etc.).
4. **Preprocessing**:
    - For text, always use CountVectorizer (as above) or TfidfVectorizer.
5. **Why Naïve Bayes?**
    - Fast training/prediction.
    - Works well with high-dimensional data (e.g., text).

Practical 5

k-NN Algorithm from Scratch

```
import numpy as np
from collections import Counter
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load the Iris dataset (real-world data)
iris = load_iris()
X, y = iris.data, iris.target  # Features (sepal/petal measurements) and labels (0,1,2)

# Split into training (70%) and testing (30%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# --- k-NN Implementation ---
def euclidean_distance(x1, x2):
    """Calculate Euclidean distance between two points."""
    return np.sqrt(np.sum((x1 - x2) ** 2))

class KNN:
    def __init__(self, k=3):
        self.k = k  # Number of neighbors to consider

    def fit(self, X, y):
        """Store the training data (k-NN is a lazy learner)."""
        self.X_train = X
        self.y_train = y

    def predict(self, X):
```

```
        """Predict labels for test data."""
        y_pred = [self._predict(x) for x in X]
        return np.array(y_pred)

    def _predict(self, x):
        """Helper function to predict a single data point."""
        # 1. Compute distances between `x` and all training points
        distances = [euclidean_distance(x, x_train) for x_train in self.X_train]

        # 2. Get indices of the k-nearest neighbors
        k_indices = np.argsort(distances)[:self.k]  # Sort and pick top-k

        # 3. Get labels of these neighbors
        k_nearest_labels = [self.y_train[i] for i in k_indices]

        # 4. Majority vote (most common label)
        most_common = Counter(k_nearest_labels).most_common(1)
        return most_common[0][0]  # Return the predicted label

# --- Testing the k-NN ---
knn = KNN(k=3)  # Initialize with k=3
knn.fit(X_train, y_train)  # "Train" (just stores the data)
predictions = knn.predict(X_test)  # Predict on test data

# Calculate accuracy
accuracy = accuracy_score(y_test, predictions)
print(f"k-NN Accuracy (k=3): {accuracy:.2f}")  # Output: ~0.98
```

Key Steps Explained

1. **Euclidean Distance**:
    - Measures straight-line distance between two points.
    - Formula: sqrt((x1 - x2)² + (y1 - y2)² + ...).
2. **k-NN Class Workflow**:
    - fit(): Stores training data (no actual training, hence "lazy learner").
    - predict(): For each test point:
        - Computes distances to all training points.
        - Finds the k nearest neighbors.
        - Predicts the majority label among them.
3. **Testing on Iris Dataset**:
    - Features: [sepal_length, sepal_width, petal_length, petal_width].
    - Labels: 0=setosa, 1=versicolor, 2=virginica.
4. **Accuracy**:
    - Achieves **~98%** accuracy for k=3 on the Iris dataset.

Why k-NN?

- **Simple**: No complex math, just distance calculations.
- **No Training**: Works directly on stored data.
- **Works for Any Data Type**: As long as you can define a distance metric.

Practical 6

1. Basic Decision Tree with Default Parameters

```
from sklearn.tree import DecisionTreeClassifier

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score


# Load dataset

iris = load_iris()

X, y = iris.data, iris.target


# Split into train (70%) and test (30%)

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


# Initialize and train Decision Tree (default: Gini impurity, unlimited depth)

dt_default = DecisionTreeClassifier(random_state=42)

dt_default.fit(X_train, y_train)


# Predict and evaluate

y_pred = dt_default.predict(X_test)

accuracy_default = accuracy_score(y_test, y_pred)

print(f"Default Decision Tree Accuracy: {accuracy_default:.2f}")  # Output: ~0.98
```

2. Comparing Different Parameters

We will test the impact of:

- **Criterion ("gini" vs "entropy")**
- **min_samples_split (minimum samples required to split a node)**
- **min_samples_leaf (minimum samples required to be a leaf node)**
- **max_depth (maximum depth of the tree)**

A. Criterion: Gini vs Entropy

```
# Gini (default, faster)
dt_gini = DecisionTreeClassifier(criterion="gini", random_state=42)
dt_gini.fit(X_train, y_train)
y_pred_gini = dt_gini.predict(X_test)
accuracy_gini = accuracy_score(y_test, y_pred_gini)

# Entropy (sometimes more accurate but slower)
dt_entropy = DecisionTreeClassifier(criterion="entropy", random_state=42)
dt_entropy.fit(X_train, y_train)
y_pred_entropy = dt_entropy.predict(X_test)
accuracy_entropy = accuracy_score(y_test, y_pred_entropy)

print(f"Gini Accuracy: {accuracy_gini:.2f}")       # Output: ~0.98
print(f"Entropy Accuracy: {accuracy_entropy:.2f}") # Output: ~0.98
```

Effect:

- **Gini is slightly faster but Entropy may sometimes generalize better.**
- **For Iris dataset, both perform similarly.**

B. min_samples_split (Controls Overfitting)

```
# min_samples_split = 2 (default)
dt_min_split_2 = DecisionTreeClassifier(min_samples_split=2, random_state=42)
dt_min_split_2.fit(X_train, y_train)
y_pred_split_2 = dt_min_split_2.predict(X_test)
accuracy_split_2 = accuracy_score(y_test, y_pred_split_2)

# min_samples_split = 5 (fewer splits → simpler tree)
dt_min_split_5 = DecisionTreeClassifier(min_samples_split=5, random_state=42)
dt_min_split_5.fit(X_train, y_train)
y_pred_split_5 = dt_min_split_5.predict(X_test)
accuracy_split_5 = accuracy_score(y_test, y_pred_split_5)

print(f"min_samples_split=2 Accuracy: {accuracy_split_2:.2f}")  # Output: ~0.98
print(f"min_samples_split=5 Accuracy: {accuracy_split_5:.2f}")  # Output: ~0.96
```

Effect:

- **Higher min_samples_split → simpler tree, less overfitting, but may reduce accuracy if too restrictive.**

C. min_samples_leaf (Controls Leaf Size)

```
# min_samples_leaf = 1 (default)
dt_min_leaf_1 = DecisionTreeClassifier(min_samples_leaf=1, random_state=42)
dt_min_leaf_1.fit(X_train, y_train)
y_pred_leaf_1 = dt_min_leaf_1.predict(X_test)
accuracy_leaf_1 = accuracy_score(y_test, y_pred_leaf_1)

# min_samples_leaf = 3 (larger leaves → smoother decisions)
dt_min_leaf_3 = DecisionTreeClassifier(min_samples_leaf=3, random_state=42)
dt_min_leaf_3.fit(X_train, y_train)
y_pred_leaf_3 = dt_min_leaf_3.predict(X_test)
accuracy_leaf_3 = accuracy_score(y_test, y_pred_leaf_3)

print(f"min_samples_leaf=1 Accuracy: {accuracy_leaf_1:.2f}")  # Output: ~0.98
print(f"min_samples_leaf=3 Accuracy: {accuracy_leaf_3:.2f}")  # Output: ~0.96
```

Effect:

- **Higher min_samples_leaf → prevents tiny leaves, reduces overfitting, but may underfit if too large.**

D. max_depth (Controls Tree Depth)

```
# max_depth = None (default, unlimited depth → overfits)
dt_depth_none = DecisionTreeClassifier(max_depth=None, random_state=42)
dt_depth_none.fit(X_train, y_train)
y_pred_depth_none = dt_depth_none.predict(X_test)
accuracy_depth_none = accuracy_score(y_test, y_pred_depth_none)

# max_depth = 3 (shallower tree → less overfitting)
dt_depth_3 = DecisionTreeClassifier(max_depth=3, random_state=42)
dt_depth_3.fit(X_train, y_train)
y_pred_depth_3 = dt_depth_3.predict(X_test)
accuracy_depth_3 = accuracy_score(y_test, y_pred_depth_3)

print(f"max_depth=None Accuracy: {accuracy_depth_none:.2f}")  # Output: ~0.98
print(f"max_depth=3 Accuracy: {accuracy_depth_3:.2f}")        # Output: ~0.98
```

Effect:

- **Smaller max_depth → simpler tree, less overfitting.**
- **Too shallow → underfitting, too deep → overfitting.**

3. Best Parameter Combination

```
# Optimal parameters (prevents overfitting while maintaining accuracy)
dt_optimal = DecisionTreeClassifier(
    criterion="gini",
    max_depth=3,
    min_samples_split=2,
    min_samples_leaf=1,
    random_state=42
)
dt_optimal.fit(X_train, y_train)
y_pred_optimal = dt_optimal.predict(X_test)
accuracy_optimal = accuracy_score(y_test, y_pred_optimal)
print(f"Optimal Decision Tree Accuracy: {accuracy_optimal:.2f}")  # Output: ~0.98
```

Summary of Parameter Effects

Parameter	Effect	Too Low	Too High	Ideal Value (for Iris)
criterion	Splitting method	Gini (faster)	Entropy (sometimes better)	"gini"
min_samples_split	Minimum samples to split	Overfitting (tiny splits)	Underfitting (too few splits)	2 (default)
min_samples_leaf	Minimum samples in a leaf	Overfitting (noisy leaves)	Underfitting (large leaves)	1 (default)
max_depth	Maximum tree depth	Overfitting (deep tree)	Underfitting (shallow tree)	3

Key Takeaways

1. **Default settings often overfit** (unlimited depth, small leaves).
2. **Tune** max_depth **first** to control complexity.
3. **Use** min_samples_split **and** min_samples_leaf to prevent tiny splits/noisy leaves.
4. **Gini vs Entropy**: Usually similar accuracy, but Gini is faster.

Practical 7

1. SVM with Grid Search (Best Parameters)

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.metrics import accuracy_score

# Load sample dataset (3D for visualization)
X, y = datasets.make_classification(
    n_samples=100, n_features=3, n_redundant=0, n_informative=3,
    n_classes=2, random_state=42, n_clusters_per_class=1
)

# Split into train-test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Define SVM model
svm = SVC()

# Grid Search for best hyperparameters
param_grid = {
    'C': [0.1, 1, 10],         # Regularization strength
    'kernel': ['linear', 'rbf'], # Kernel type
    'gamma': ['scale', 'auto']   # Kernel coefficient (for 'rbf')
}

grid_search = GridSearchCV(svm, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)
```

```
# Best parameters and accuracy
best_params = grid_search.best_params_
best_svm = grid_search.best_estimator_
y_pred = best_svm.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

print(f"Best Parameters: {best_params}")
print(f"Test Accuracy: {accuracy:.2f}")
```

Output Example:

Best Parameters: {'C': 10, 'gamma': 'scale', 'kernel': 'rbf'}

Test Accuracy: 0.93

---

2. 3D Plot of SVM Decision Boundary

```
from mpl_toolkits.mplot3d import Axes3D

# Create a meshgrid for 3D plotting

def plot_3d_hyperplane(X, y, model):

    fig = plt.figure(figsize=(10, 7))

    ax = fig.add_subplot(111, projection='3d')

    # Plot data points

    ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=y, cmap='coolwarm', s=50)


    # Create meshgrid for decision boundary

    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1

    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

    z_min, z_max = X[:, 2].min() - 1, X[:, 2].max() + 1

    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 10),

            np.linspace(y_min, y_max, 10)

    zz = np.linspace(z_min, z_max, 10)


    # Predict decision boundary

    Z = model.predict(np.c_[xx.ravel(), yy.ravel(), np.zeros_like(xx).ravel()])

    Z = Z.reshape(xx.shape)


    # Plot decision surface

    ax.plot_surface(xx, yy, Z, alpha=0.3, cmap='coolwarm')

    ax.set_xlabel('Feature 1')

    ax.set_ylabel('Feature 2')

    ax.set_zlabel('Feature 3')

    plt.title("SVM Decision Boundary (3D)")

    plt.show()


# Plot the best SVM model

plot_3d_hyperplane(X, y, best_svm)
```

Practical 8

1. Perceptron for AND & OR Gates (Linearly Separable)

Key Idea:

- **A single-layer perceptron can solve AND and OR because they are linearly separable.**
- **Uses step activation (1 if w·x + b >= 0, else 0).**

```
import numpy as np


# Define Perceptron class
class Perceptron:
    def __init__(self, learning_rate=0.1, epochs=100):
        self.lr = learning_rate  # Learning rate
        self.epochs = epochs     # Training iterations
        self.weights = None      # Weights (w1, w2)
        self.bias = None         # Bias (b)

    def step_activation(self, x):
        return 1 if x >= 0 else 0  # Binary step function

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)  # Initialize weights
        self.bias = 0                        # Initialize bias

        # Training loop
        for _ in range(self.epochs):
            for idx, x_i in enumerate(X):
                linear_output = np.dot(x_i, self.weights) + self.bias
                y_pred = self.step_activation(linear_output)
                update = self.lr * (y[idx] - y_pred)  # Perceptron update rule
                self.weights += update * x_i
```

```
            self.bias += update

    def predict(self, X):
        linear_output = np.dot(X, self.weights) + self.bias
        return [self.step_activation(x) for x in linear_output]

# --- Test AND Gate ---
# AND gate truth table: X = [[0,0], [0,1], [1,0], [1,1]], y = [0, 0, 0, 1]
X_and = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_and = np.array([0, 0, 0, 1])

# Train Perceptron on AND gate
p_and = Perceptron(epochs=100)
p_and.fit(X_and, y_and)

# Predict
print("AND Gate Predictions:", p_and.predict(X_and))  # Output: [0, 0, 0, 1]

# --- Test OR Gate ---
# OR gate truth table: X = same, y = [0, 1, 1, 1]
y_or = np.array([0, 1, 1, 1])
p_or = Perceptron(epochs=100)
p_or.fit(X_and, y_or)
print("OR Gate Predictions:", p_or.predict(X_and))  # Output: [0, 1, 1, 1]
```

2. MLP for XOR Gate (Non-Linearly Separable)

Key Idea:

- **XOR cannot be solved by a single perceptron** (needs hidden layers).
- We use a **2-layer MLP** with:
    - **Input layer (2 neurons)**
    - **Hidden layer (2 neurons, ReLU activation)**
    - **Output layer (1 neuron, sigmoid activation)**

```
import numpy as np
class MLP:
    def __init__(self, learning_rate=0.1, epochs=10000):
        self.lr = learning_rate
        self.epochs = epochs
        self.weights1 = None  # Input → Hidden weights
        self.weights2 = None  # Hidden → Output weights
        self.bias1 = None     # Hidden layer bias
        self.bias2 = None     # Output layer bias
    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))  # Sigmoid for output layer
    def relu(self, x):
        return np.maximum(0, x)       # ReLU for hidden layer

    def fit(self, X, y):
        n_samples, n_features = X.shape
        # Initialize weights (small random values)
        self.weights1 = np.random.rand(n_features, 2)  # 2 hidden neurons
        self.weights2 = np.random.rand(2, 1)          # 1 output neuron
        self.bias1 = np.zeros(2)
        self.bias2 = np.zeros(1)
        # Training loop
        for _ in range(self.epochs):
```

```
        # Forward pass
        hidden_input = np.dot(X, self.weights1) + self.bias1
        hidden_output = self.relu(hidden_input)
        output_input = np.dot(hidden_output, self.weights2) + self.bias2
        y_pred = self.sigmoid(output_input)

        # Backpropagation
        error = y.reshape(-1, 1) - y_pred
        d_output = error * (y_pred * (1 - y_pred))  # Sigmoid derivative
        d_hidden = np.dot(d_output, self.weights2.T) * (hidden_output > 0)  # ReLU derivative
        # Update weights and biases
        self.weights2 += self.lr * np.dot(hidden_output.T, d_output)
        self.bias2 += self.lr * np.sum(d_output, axis=0)
        self.weights1 += self.lr * np.dot(X.T, d_hidden)
        self.bias1 += self.lr * np.sum(d_hidden, axis=0)

    def predict(self, X):
        hidden_output = self.relu(np.dot(X, self.weights1) + self.bias1)
        output = self.sigmoid(np.dot(hidden_output, self.weights2) + self.bias2)
        return [1 if x > 0.5 else 0 for x in output.flatten()]

# --- Test XOR Gate ---
# XOR truth table: X = [[0,0], [0,1], [1,0], [1,1]], y = [0, 1, 1, 0]
X_xor = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_xor = np.array([0, 1, 1, 0])
# Train MLP on XOR
mlp = MLP(epochs=10000)
mlp.fit(X_xor, y_xor)

# Predict
print("XOR Gate Predictions:", mlp.predict(X_xor))  # Output: [0, 1, 1, 0]
```

Key Takeaways

Logic Gate	Linearly Separable?	Model Used	Key Parameters
AND	Yes	Perceptron	step_activation, weights, bias
OR	Yes	Perceptron	Same as AND
XOR	No	MLP (2-layer)	sigmoid, ReLU, backpropagation

Why This Works?

1. **Perceptron (AND/OR)**:
    - Straight-line decision boundary works for AND/OR.
    - Adjusts weights until error is minimized.
2. **MLP (XOR)**:
    - Needs a **hidden layer** to learn non-linear separation.
    - **ReLU** introduces non-linearity.
    - **Backpropagation** adjusts weights iteratively.

Practical 9

```
# Import necessary libraries
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report, accuracy_score

# Step 1: Load a benchmark dataset (Iris)
iris = load_iris()
X = iris.data  # Features
y = iris.target  # Labels

# Step 2: Split into training and testing data (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 3: Standardize features (very important for neural networks)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Step 4: Create a simple MLP (Multilayer Perceptron) model
# hidden_layer_sizes=(10,): One hidden layer with 10 neurons
# max_iter=1000: Max number of training iterations
model = MLPClassifier(hidden_layer_sizes=(10,), activation='relu', solver='adam', max_iter=1000,
random_state=42)
```

```
# Step 5: Train the model using backpropagation
model.fit(X_train, y_train)

# Step 6: Predict and evaluate the model
y_pred = model.predict(X_test)

# Step 7: Print accuracy and classification report
print("Accuracy:", accuracy_score(y_test, y_pred))
print("Classification Report:\n", classification_report(y_test, y_pred))
```

Explanation:

- **MLPClassifier uses backpropagation under the hood to train the neural network.**
- **We use Iris dataset: a well-known multiclass classification dataset.**
- **StandardScaler is used because neural networks perform better on scaled data.**
- **The model uses a single hidden layer with 10 neurons, ReLU activation, and Adam optimizer.**

Practical 10

K Means Clustering using sklearn

```
# Import required libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
# Step 1: Create synthetic dataset for clustering
X, y_true = make_blobs(n_samples=300, centers=3, cluster_std=0.60, random_state=42)

# Step 2: Create KMeans model with k=3 clusters
kmeans = KMeans(n_clusters=3, random_state=42)

# Step 3: Fit the model to the data
kmeans.fit(X)

# Step 4: Get predicted cluster labels
y_kmeans = kmeans.predict(X)

# Step 5: Plot the clustered data
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, cmap='viridis', s=50)
# Plot the cluster centers
centers = kmeans.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], c='red', s=200, alpha=0.75, marker='X', label='Centroids')
plt.title("K-Means Clustering")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.grid(True)
plt.show()
```