**1. Implement the k-Nearest Neighbor (k-NN) algorithm to classify a given dataset using Minkowski distance**

for p=3. Evaluate the accuracy of the classifier.

import numpy as np

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

from collections import Counter

def minkowski\_distance(x1, x2, p=3):

return np.sum(np.abs(x1 - x2) \*\* p) \*\* (1 / p)

def k\_nearest\_neighbors(X\_train, y\_train, X\_test, k, p=3):

y\_pred = []

for x\_test in X\_test:

distances = [minkowski\_distance(x\_test, x\_train, p) for x\_train in X\_train]

k\_indices = np.argsort(distances)[:k]

k\_labels = [y\_train[i] for i in k\_indices]

most\_common = Counter(k\_labels).most\_common(1)[0][0]

y\_pred.append(most\_common)

return np.array(y\_pred)

data = load\_iris()

X, y = data.data, data.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

k = 5

y\_pred = k\_nearest\_neighbors(X\_train, y\_train, X\_test, k, p=3)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy of k-NN classifier (k={k}, p=3): {accuracy \* 100:.2f}%")

**2. Implement the Iterative Dichotomiser (ID3) algorithm with entropy as the criterion to build a decision**

from sklearn.tree import DecisionTreeClassifier

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

from sklearn.datasets import load\_iris

data = load\_iris()

X=data.data

y=data.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

dt = DecisionTreeClassifier(criterion='entropy')

dt.fit(X\_train, y\_train)

y\_pred = dt.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Decision Tree (Entropy) Accuracy: {accuracy\*100:.2f}%")

tree using a given dataset. Evaluate the classifier by computing its accuracy.

**3. Implement feature reduction using Principal Component Analysis by at least one dimension for a given dataset. Evaluate the performance of Logistic regression before after applying PCA.**

from sklearn.decomposition import PCA

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score

data = load\_iris()

X = data.data

y = data.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

log\_reg = LogisticRegression(max\_iter=200)

log\_reg.fit(X\_train, y\_train)

y\_pred\_before = log\_reg.predict(X\_test)

accuracy\_before = accuracy\_score(y\_test, y\_pred\_before)

pca = PCA(n\_components=3)

X\_train\_pca = pca.fit\_transform(X\_train)

X\_test\_pca = pca.transform(X\_test)

log\_reg.fit(X\_train\_pca, y\_train)

y\_pred\_after = log\_reg.predict(X\_test\_pca)

accuracy\_after = accuracy\_score(y\_test, y\_pred\_after)

print(f"Logistic Regression Accuracy (before PCA): {accuracy\_before \* 100:.2f}%")

print(f"Logistic Regression Accuracy (after PCA): {accuracy\_after \* 100:.2f}%")

**4. Implement the na ̈ıve Bayesian classifier for a given data set. Compute the accuracy of the classifier, considering few test data. sets.**

from sklearn.naive\_bayes import GaussianNB

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

from sklearn.datasets import load\_iris

data = load\_iris()

X = data.data

y = data.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

nb = GaussianNB()

nb.fit(X\_train, y\_train)

y\_pred = nb.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Naïve Bayes Classifier Accuracy: {accuracy \* 100:.2f}%")

**5. Implement Support Vector Machine (SVM) model for a given data for Kernels :’linear’, ’poly’, ’rbf’, ’sigmoid’. Plot the support vectors and with regions of classes in each case. Evaluate their performance on the test data and suggest the best fitting kernel.**

import numpy as np

import matplotlib.pyplot as plt

from sklearn import datasets

from sklearn.model\_selection import train\_test\_split

from sklearn.svm import SVC

from sklearn.metrics import accuracy\_score

from sklearn.preprocessing import StandardScaler

iris = datasets.load\_iris()

X = iris.data[:, :2]

y = iris.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

def plot\_decision\_boundary(X, y, model, kernel\_name):

h = 0.02 # Step size in the meshgrid

x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h),

np.arange(y\_min, y\_max, h))

Z = model.predict(np.c\_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.75, cmap=plt.cm.coolwarm)

plt.scatter(X[:, 0], X[:, 1], c=y, s=30, edgecolors='k', cmap=plt.cm.coolwarm)

plt.scatter(model.support\_vectors\_[:, 0], model.support\_vectors\_[:, 1],

s=100, facecolors='none', edgecolors='k', linewidths=2, marker='o')

plt.title(f"SVM with {kernel\_name} Kernel")

plt.show()

kernels = ['linear', 'poly', 'rbf', 'sigmoid']

accuracies = {}

for kernel in kernels:

print(f"Training SVM with {kernel} kernel...")

model = SVC(kernel=kernel, random\_state=42)

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

accuracies[kernel] = accuracy

plot\_decision\_boundary(X\_train, y\_train, model, kernel)

for kernel, accuracy in accuracies.items():

print(f"Accuracy with {kernel} kernel: {accuracy \* 100:.2f}%")

best\_kernel = max(accuracies, key=accuracies.get)

print(f"The best fitting kernel is: {best\_kernel}")

**6. Implement Rosenblatt’s perceptron model for the Boolean expression ((p ∧ q) ∨ r) → (p∧ ∼ r) and evaluate its accuracy.**

import numpy as np

# Define the Boolean expression ((p ∧ q) ∨ r) → (p ∧ ∼r)

def boolean\_expression(p, q, r):

lhs = (p and q) or r

rhs = p and not r

return not lhs or rhs

data = []

for p in [0, 1]:

for q in [0, 1]:

for r in [0, 1]:

output = boolean\_expression(p, q, r)

data.append([p, q, r,output])

data = np.array(data)

X=data[:,:-1]

Y=data[:,-1]

class Perceptron:

def \_init\_(self, input\_size, lr=0.1, epochs=1000):

self.weights = np.zeros(input\_size) # Initialize weights to zero

self.bias = 0 # Initialize bias to zero

self.lr = lr # Learning rate

self.epochs = epochs # Number of training epochs

def fit(self, X, y):

for \_ in range(self.epochs):

for xi, target in zip(X, y):

linear\_output = np.dot(xi, self.weights) + self.bias

prediction = self.activation\_function(linear\_output)

error = target - prediction

self.weights += self.lr \* error \* xi

self.bias += self.lr \* error

def activation\_function(self, x):

return 1 if x >= 0 else 0

def predict(self, X):

linear\_output = np.dot(X, self.weights) + self.bias

return np.array([self.activation\_function(output) for output in linear\_output])

perceptron = Perceptron(input\_size=3)

perceptron.fit(X, Y)

predictions = perceptron.predict(X)

accuracy = np.mean(predictions == Y)

print(f"Perceptron Accuracy: {accuracy \* 100:.2f}%")

**7. Implement polynomial regression using Stochastic Gradient Descent for the given dataset, plot the accuracy for different degrees and conclude the best fit.**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import PolynomialFeatures

from sklearn.linear\_model import SGDRegressor

from sklearn.metrics import mean\_squared\_error

from sklearn.model\_selection import train\_test\_split

from sklearn.datasets import load\_iris

data = load\_iris()

X = data.data[:, 0].reshape(-1, 1)

y = data.target.reshape(-1, 1)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

degrees = range(1, 10)

train\_errors = []

test\_errors = []

for degree in degrees:

# Polynomial feature transformation

poly = PolynomialFeatures(degree=degree, include\_bias=False)

X\_train\_poly = poly.fit\_transform(X\_train)

X\_test\_poly = poly.transform(X\_test)

# SGD Regressor

sgd\_reg = SGDRegressor(max\_iter=1000, tol=1e-3, random\_state=42)

sgd\_reg.fit(X\_train\_poly, y\_train.ravel())

# Predictions and errors

y\_train\_pred = sgd\_reg.predict(X\_train\_poly)

y\_test\_pred = sgd\_reg.predict(X\_test\_poly)

train\_errors.append(mean\_squared\_error(y\_train, y\_train\_pred))

test\_errors.append(mean\_squared\_error(y\_test, y\_test\_pred))

# Plot the errors for different polynomial degrees

plt.figure(figsize=(10, 6))

plt.plot(degrees, train\_errors, label="Train Error", marker='o')

plt.plot(degrees, test\_errors, label="Test Error", marker='o')

plt.xlabel("Polynomial Degree")

plt.ylabel("Mean Squared Error")

plt.title("Polynomial Degree vs. Error")

plt.legend()

plt.grid(True)

plt.show()

# Best degree conclusion

best\_degree = degrees[np.argmin(test\_errors)]

print(f"The best polynomial degree is {best\_degree} with test error {min(test\_errors):.2f}.")

**8. Implement multiple linear regression using Stochastic Gradient Descent for the given dataset and compute the accuracy on the test data.**

from sklearn.linear\_model import SGDRegressor

from sklearn.metrics import mean\_squared\_error, r2\_score

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import LabelEncoder

from sklearn.datasets import load\_iris

data=load\_iris()

X=data.data

y=data.target

label\_encoder = LabelEncoder()

y = label\_encoder.fit\_transform(y)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

sgd = SGDRegressor(max\_iter=1000, tol=1e-3, random\_state=42)

sgd.fit(X\_train, y\_train)

y\_pred = sgd.predict(X\_test)

mse = mean\_squared\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

print(f"Mean Squared Error (MSE): {mse:.4f}")

print(f"R-squared (R2) Score: {r2:.4f}")

**9. Implement the Iterative Dichotomiser (ID3) algorithm with Gini as the criterion to build a decision tree for a given dataset. Evaluate the classifier by computing its accuracy.**

from sklearn.datasets import load\_iris

from sklearn.tree import DecisionTreeClassifier, plot\_tree

from sklearn.metrics import accuracy\_score

from sklearn.model\_selection import train\_test\_split

import matplotlib.pyplot as plt

import numpy as np

data=load\_iris()

X=data.data

y=data.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Train Decision Tree using Gini criterion

dt = DecisionTreeClassifier(criterion='gini', random\_state=42)

dt.fit(X\_train, y\_train)

# Predict on test data

y\_pred = dt.predict(X\_test)

# Evaluate accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Decision Tree (Gini) Accuracy: {accuracy\*100:.2f}%")

**10. Implement the ADALINE model using the Delta Rule for binary classification for a given dataset. Evaluate the classifier by computing its accuracy.**

import numpy as np

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.preprocessing import LabelEncoder

class Adaline:

def \_init\_(self, lr=0.01, epochs=100):

self.lr = lr

self.epochs = epochs

def fit(self, X, y):

self.weights = np.zeros(X.shape[1])

self.bias = 0

for \_ in range(self.epochs):

for xi, target in zip(X, y):

y\_pred = np.dot(xi, self.weights) + self.bias

error = target - y\_pred

self.weights += self.lr \* error \* xi

self.bias += self.lr \* error

def predict(self, X):

return (np.dot(X, self.weights) + self.bias >= 0).astype(int)

data=load\_iris()

X=data.data

y=data.target

label\_encoder = LabelEncoder()

y = label\_encoder.fit\_transform(y)

scaler = StandardScaler()

X = scaler.fit\_transform(X)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

model = Adaline(lr=0.01, epochs=1000)

model.fit(X\_train, y\_train)

accuracy = np.mean(model.predict(X\_test) == y\_test)

print("Accuracy:", accuracy)

**11. Implement ADALINE for a regression task using sigmoid activation and delta rule, where the model predicts continuous values for a given dataset. Compute the mean squared error on the test data.**

from sklearn.linear\_model import SGDRegressor

from sklearn.metrics import mean\_squared\_error

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import LabelEncoder

import pandas as pd

import numpy as np

from sklearn.datasets import load\_iris

def sigmoid(x):

return 1 / (1 + np.exp(-x))

data=load\_iris()

X=data.data

y=data.target

y = label\_encoder.fit\_transform(y)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

adaline = SGDRegressor(max\_iter=1000, tol=1e-3, random\_state=42)

adaline.fit(X\_train, y\_train)

y\_pred\_linear = adaline.predict(X\_test)

y\_pred\_sigmoid = sigmoid(y\_pred\_linear)

mse = mean\_squared\_error(y\_test, y\_pred\_sigmoid)

print(f"ADALINE Regression MSE (with Sigmoid Activation): {mse:.4f}")

**12. Implement the MADALINE model for a regression task, where multiple ADALINE neurons are used in the hidden layer to predict continuous values with ReLU activations. Compute the mean squared error on the test data.**

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import LabelEncoder

import pandas as pd

import numpy as np

from sklearn.datasets import load\_iris

def relu(x):

return np.maximum(0, x)

data=load\_iris()

X=data.data

y=data.target

label\_encoder = LabelEncoder()

y = label\_encoder.fit\_transform(y)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

hidden\_layer\_weights = np.random.randn(X\_train.shape[1], 64) # Random weights

hidden\_layer\_bias = np.random.randn(64)

X\_train\_hidden = relu(np.dot(X\_train, hidden\_layer\_weights) + hidden\_layer\_bias)

X\_test\_hidden = relu(np.dot(X\_test, hidden\_layer\_weights) + hidden\_layer\_bias)

regressor = LinearRegression()

regressor.fit(X\_train\_hidden, y\_train)

y\_pred = regressor.predict(X\_test\_hidden)

mse = mean\_squared\_error(y\_test, y\_pred)

print(f"MADALINE Regression MSE (Alternative Approach): {mse:.4f}")

**13. Implement the MADALINE model for a binary classification task, where multiple ADALINE neurons are used in the hidden layers with ReLU activations. Compute the accuracy on the test data.**

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score, classification\_report

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.preprocessing import LabelEncoder

import pandas as pd

import numpy as np

from sklearn.datasets import load\_iris

def relu(x):

return np.maximum(0, x)

data=load\_iris()

X=data.data

y=data.target

y = label\_encoder.fit\_transform(y)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

hidden\_layer\_weights = np.random.randn(X\_train.shape[1], 64) # Random weights

hidden\_layer\_bias = np.random.randn(64)

X\_train\_hidden = relu(np.dot(X\_train, hidden\_layer\_weights) + hidden\_layer\_bias)

X\_test\_hidden = relu(np.dot(X\_test, hidden\_layer\_weights) + hidden\_layer\_bias)

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train\_hidden)

X\_test\_scaled = scaler.transform(X\_test\_hidden)

classifier = LogisticRegression(solver='lbfgs', max\_iter=1000, random\_state=42) # Increased max\_iter

classifier.fit(X\_train\_scaled, y\_train)

y\_pred = classifier.predict(X\_test\_scaled)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"\nMADALINE Classification Accuracy: {accuracy:.4f}")

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred))

**14. Apply Linear Discriminant Analysis (LDA) for feature reduction to improve classification performance of Logistic regression on a given dataset.**

import numpy as np

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.discriminant\_analysis import LinearDiscriminantAnalysis

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

data = load\_iris()

X = data.data

y = data.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

log\_reg = LogisticRegression(max\_iter=1000, random\_state=42)

log\_reg.fit(X\_train, y\_train)

y\_pred\_no\_lda = log\_reg.predict(X\_test)

accuracy\_no\_lda = accuracy\_score(y\_test, y\_pred\_no\_lda)

print("Logistic Regression Accuracy without LDA:", accuracy\_no\_lda)

lda = LinearDiscriminantAnalysis(n\_components=2)

X\_train\_lda = lda.fit\_transform(X\_train, y\_train)

X\_test\_lda = lda.transform(X\_test)

log\_reg\_lda = LogisticRegression(max\_iter=1000, random\_state=42)

log\_reg\_lda.fit(X\_train\_lda, y\_train)

y\_pred\_lda = log\_reg\_lda.predict(X\_test\_lda)

accuracy\_with\_lda = accuracy\_score(y\_test, y\_pred\_lda)

print("Logistic Regression Accuracy with LDA:", accuracy\_with\_lda)

print("\nClassification Report without LDA:")

print(classification\_report(y\_test, y\_pred\_no\_lda))

print("\nClassification Report with LDA:")

print(classification\_report(y\_test, y\_pred\_lda))

**15. Implement the K-means clustering algorithm for a given dataset. Plot the performance graph Inertia vs K.**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

from sklearn.datasets import load\_iris

from sklearn.preprocessing import StandardScaler

# Load dataset (Iris dataset in this case)

data = load\_iris()

X = data.data # Features

# Standardize the data (important for K-means performance)

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

# List to store inertia values for different K

inertia\_values = []

# Try different values of K (number of clusters)

K\_range = range(1, 11) # Try K from 1 to 10 clusters

for k in K\_range:

# Initialize KMeans model with k clusters

kmeans = KMeans(n\_clusters=k,n\_init='auto',random\_state=42)

kmeans.fit(X\_scaled)

# Append the inertia (sum of squared distances) to the list

inertia\_values.append(kmeans.inertia\_)

# Plot Inertia vs K

plt.figure(figsize=(8, 6))

plt.plot(K\_range, inertia\_values, marker='o', linestyle='-', color='b')

plt.title('Inertia vs. Number of Clusters (K)')

plt.xlabel('Number of Clusters (K)')

plt.ylabel('Inertia')

plt.grid(True)

plt.show()

**16. Implement hierarchical clustering for a given dataset without using labels and evaluate the accuracy on the testdata.**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.datasets import load\_iris

from sklearn.preprocessing import StandardScaler

from sklearn.cluster import AgglomerativeClustering

from sklearn.metrics import silhouette\_score

import scipy.cluster.hierarchy as sch

data = load\_iris()

X = data.data

y = data.target

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

clustering = AgglomerativeClustering(n\_clusters=3, metric='euclidean', linkage='ward')

y\_pred = clustering.fit\_predict(X\_scaled)

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

sch.dendrogram(sch.linkage(X\_scaled, method='ward'))

plt.title('Dendrogram for Hierarchical Clustering')

plt.xlabel('Samples')

plt.ylabel('Distance')

plt.show()

sil\_score = silhouette\_score(X\_scaled, y\_pred)

print(f"Silhouette Score: {sil\_score:.2f}")