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 naïve 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}")
```