K-Nearest Neighbors (KNN)

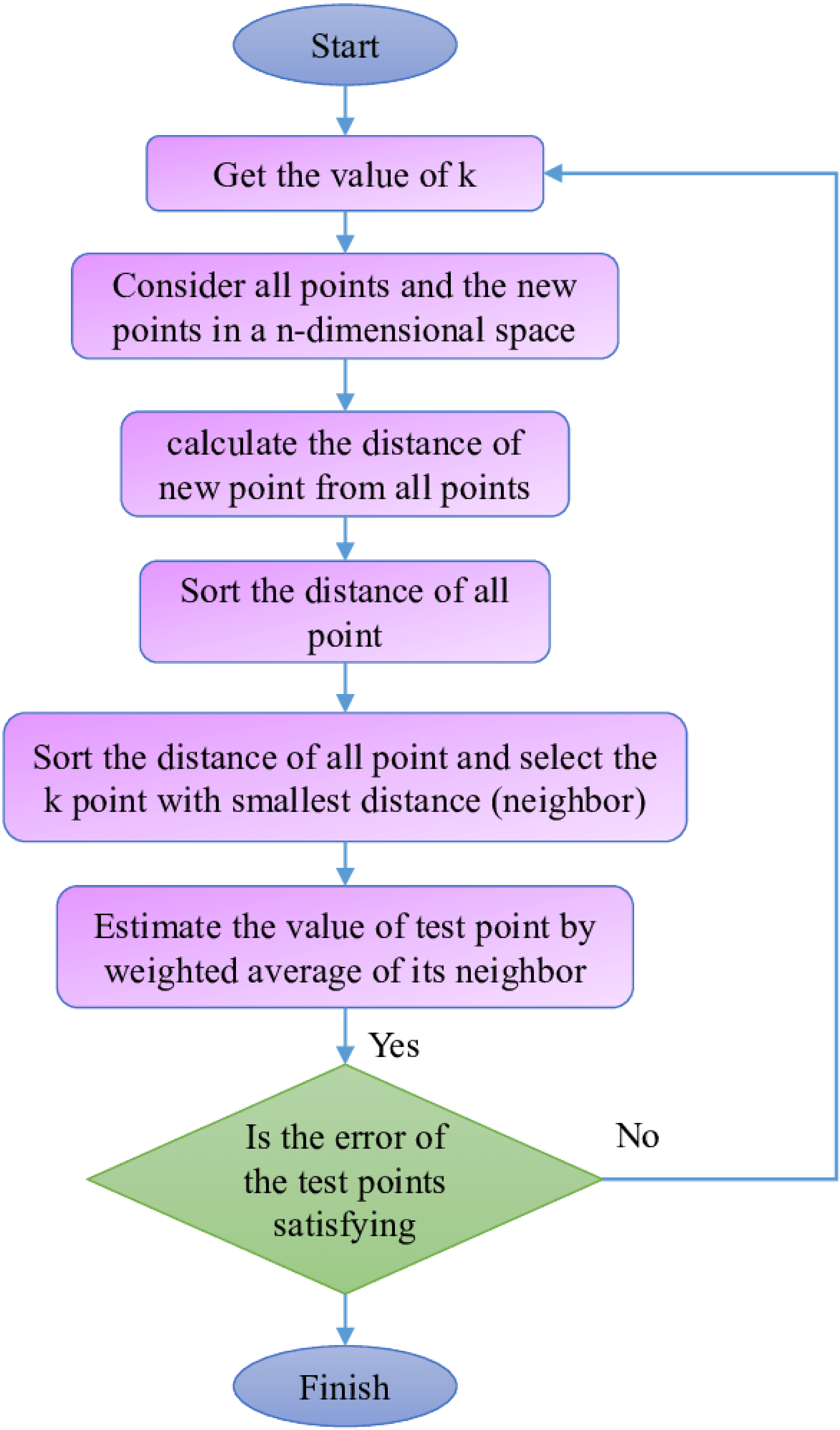
1. Description of Algorithm

K-Nearest Neighbors (KNN) is a simple, non-parametric supervised learning algorithm used for classification and regression.

- **Working Principle**: KNN classifies a data point based on the majority vote of its (k)-nearest neighbors.
- **Applications**: Image recognition, recommendation systems, and medical diagnosis.
- **Advantages**: Simple and intuitive, no training phase.
- **Disadvantages**: Computationally expensive, sensitive to noisy data.

---

2. Flow Diagram


Start
Get the value of k
Consider all points and the new points in a n-dimensional space
calculate the distance of new point from all points
Sort the distance of all point
Sort the distance of all point and select the k point with smallest distance (neighbor)
Estimate the value of test point by weighted average of its neighbor
Yes
Is the error of the test points satisfying
No
Finish

3. Mathematical Model

- **Distance Metric**: Euclidean distance is most commonly used:

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

- **Prediction**:
  - **Classification**: Majority vote among k nearest neighbors.
  - **Regression**: Mean of k nearest neighbors.

---

4. Python Implementation

Dataset: Iris Dataset (classification example).

We use the Iris dataset available from `sklearn.datasets`.

```
# Import Libraries
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.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, classification_report

# Load Dataset
iris = load_iris()
X, y = iris.data, iris.target

# Split Dataset into Training and Testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Standardize the Data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# KNN Classifier
k = 5  # Number of Neighbors
knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(X_train, y_train)
```

```
# Make Predictions
y_pred = knn.predict(X_test)

# Evaluate the Model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
print("Classification Report:\n", classification_report(y_test, y_pred))
```

---

5. Dataset File and Output

The Iris dataset is part of `sklearn`

```
Accuracy: 1.00
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45

PS D:\MS_Things\UET_DS\Semester_1\Advance_machine_learning\Assignements\Assignmetn2>
```

Naïve Bayes (NB)

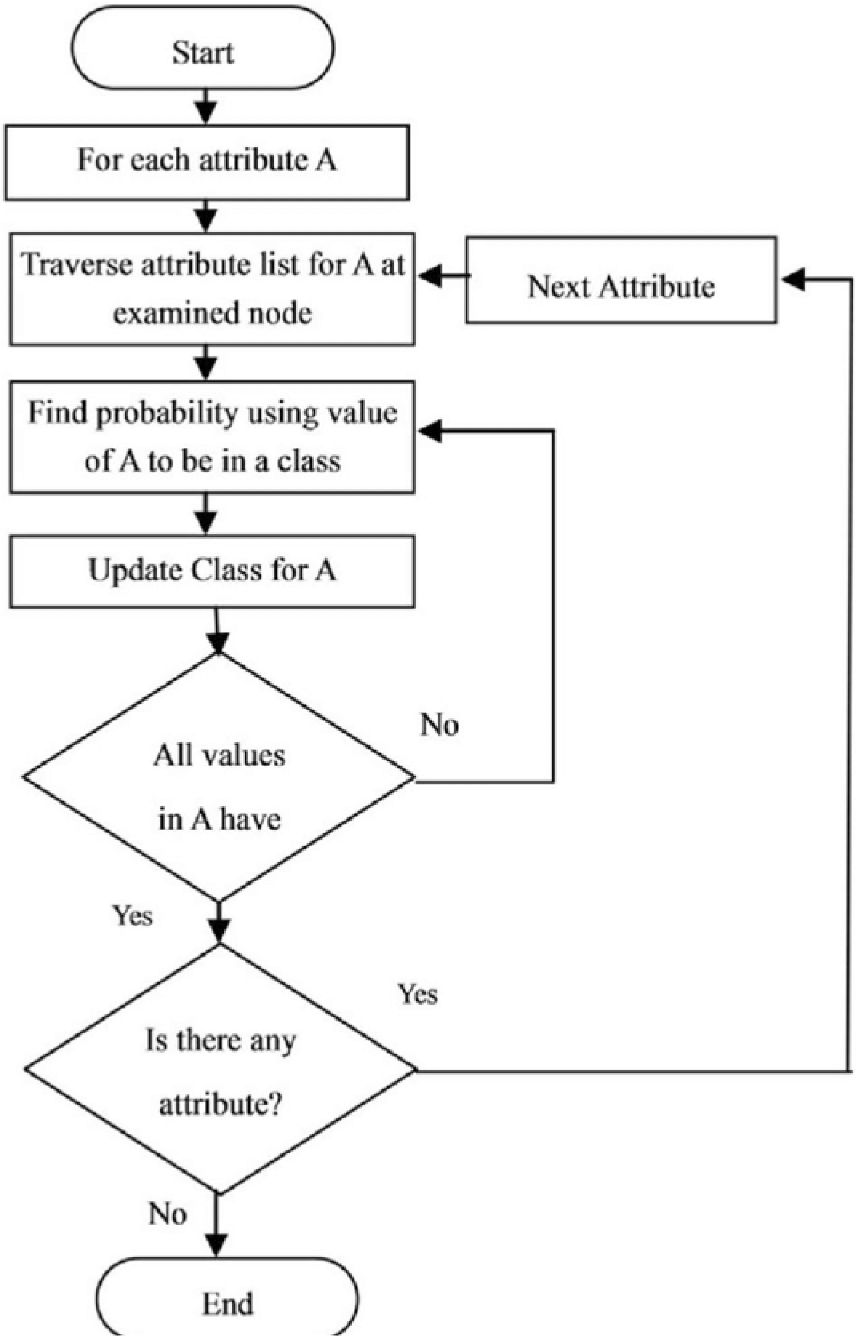
1. Description of Algorithm

Naïve Bayes is a probabilistic supervised learning algorithm based on Bayes' Theorem.

- **Assumption**: All features are independent (hence "naïve").
- **Working Principle**: It calculates the probability of each class given the input features and selects the class with the highest probability.
- **Applications**: Spam filtering, sentiment analysis, and document classification.
- **Advantages**: Fast, simple, and effective for large datasets.
- **Disadvantages**: The independence assumption rarely holds in real-world data.

---

2. Flow Diagram



3. Mathematical Model

Bayes' Theorem:

$$P(y|X) = \frac{P(X|y)P(y)}{P(X)}$$

Where:

- $P(y|X)$: Posterior probability (probability of class y given data X).
- $P(X|y)$: Likelihood (probability of data X given class y).
- $P(y)$: Prior probability of class y.
- $P(X)$: Evidence (probability of the data).

- $P(X)$: Evidence (probability of the data).

For Naïve Bayes, we assume independence between features:

$$P(X|y) = P(x_1|y) \cdot P(x_2|y) \cdot \ldots \cdot P(x_n|y)$$

4. Python Implementation

Dataset: SMS Spam Classification Dataset.

```python
# Import Libraries
import numpy as np
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, classification_report

# Load Dataset
categories = ['alt.atheism', 'sci.space']  # Using two categories for simplicity
newsgroups = fetch_20newsgroups(subset='all', categories=categories)
X, y = newsgroups.data, newsgroups.target

# Convert Text to Numerical Data
vectorizer = CountVectorizer(stop_words='english')
X = vectorizer.fit_transform(X)

# Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Train Naïve Bayes Model
nb = MultinomialNB()
nb.fit(X_train, y_train)

# Predictions
y_pred = nb.predict(X_test)

# Evaluation
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
print("Classification Report:\n", classification_report(y_test, y_pred))
```

---

5. Dataset File and Output

The dataset is available in `sklearn.datasets`

```
...   Accuracy: 1.00
      Classification Report:
                    precision    recall  f1-score   support

                 0       1.00      0.99      1.00       237
                 1       0.99      1.00      1.00       299

          accuracy                           1.00       536
         macro avg       1.00      1.00      1.00       536
      weighted avg       1.00      1.00      1.00       536
```

Support Vector Machine (SVM)

---

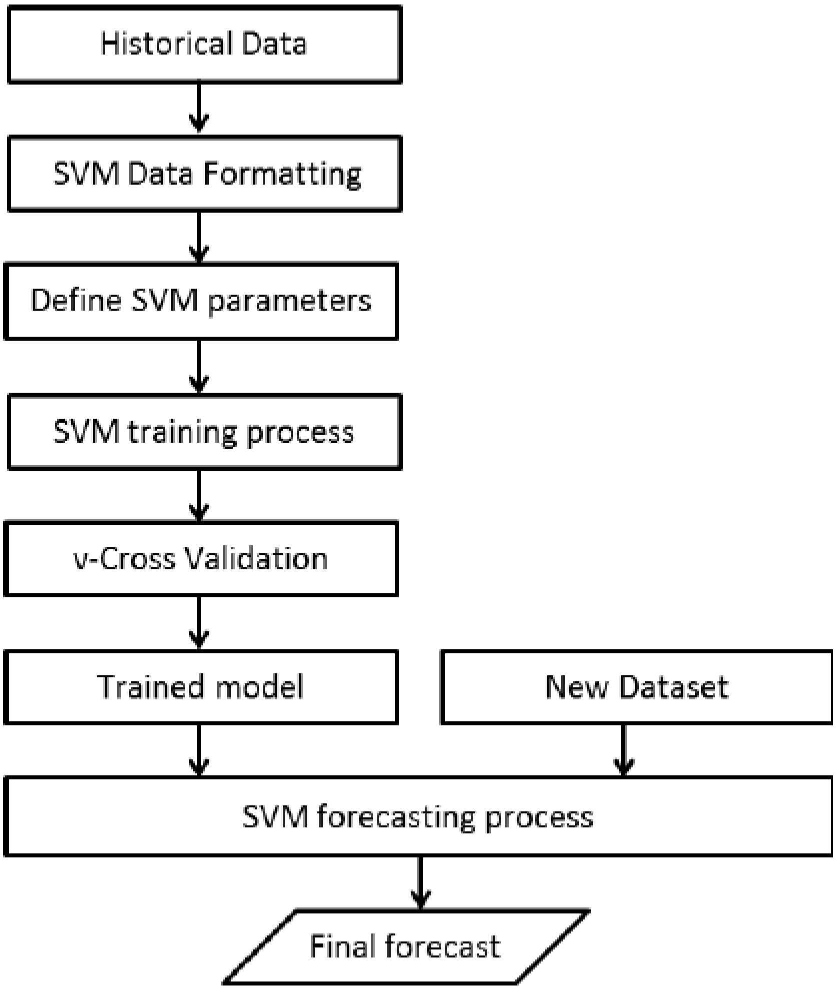
1. Description of Algorithm

Support Vector Machine (SVM) is a supervised learning algorithm used for classification and regression.

- **Working Principle**: It finds the hyperplane that best separates classes with the maximum margin.
- **Applications**: Image classification, bioinformatics, and text categorization.
- **Advantages**: Effective for high-dimensional data, works well with a clear margin of separation.
- **Disadvantages**: Not suitable for large datasets, sensitive to the choice of kernel.

---

2. Flow Diagram



3. Mathematical Model

- **Objective**: Maximize the margin between two classes while minimizing classification errors.
- **Optimization Problem**:

$$\min \frac{1}{2}||w||^2 \quad \text{subject to } y_i(w \cdot x_i + b) \geq 1$$

- **Kernel Trick**: Maps the data into a higher-dimensional space to make it linearly separable.

---

4. Python Implementation

Dataset: Iris Dataset (classification example).

```
# Import Libraries
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report

# Load Dataset
iris = datasets.load_iris()
```

```python
X, y = iris.data, iris.target

# Select Only Two Classes for Binary Classification
X = X[y != 2]
y = y[y != 2]

# Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Standardize the Data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Train SVM Model
svm = SVC(kernel='linear', C=1.0, random_state=42)
svm.fit(X_train, y_train)

# Predictions
y_pred = svm.predict(X_test)

# Evaluation
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
print("Classification Report:\n", classification_report(y_test, y_pred))
```

---

5. Dataset File and Ouptut

The **Iris Dataset** is included in `sklearn` and does not require external download.

```
Accuracy: 1.00
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        17
           1       1.00      1.00      1.00        13

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

Artificial Neural Network (ANN)

---

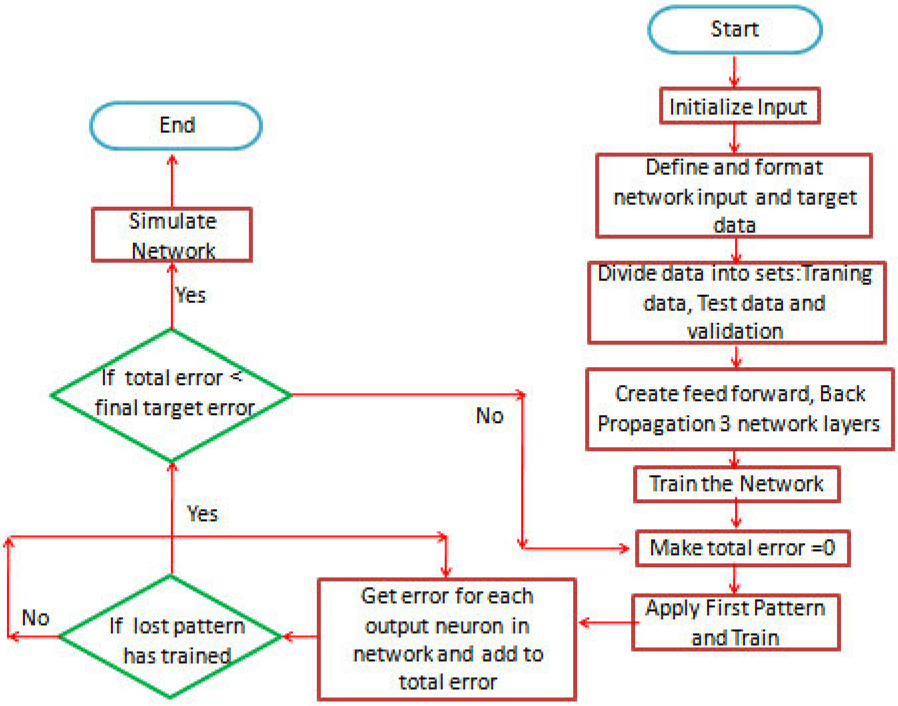
1. Description of Algorithm

An Artificial Neural Network (ANN) is inspired by the structure and function of the human brain.

- **Working Principle**: It consists of interconnected layers of neurons that transform input data to output predictions by learning weights through backpropagation.
- **Applications**: Image recognition, speech processing, natural language processing.
- **Advantages**: Can model complex patterns and nonlinear relationships.
- **Disadvantages**: Requires large datasets, computationally expensive, prone to overfitting.

---

2. Flow Diagram



3. Mathematical Model

1. **Forward Pass**:

$$a^{(l)} = f(W^{(l)} \cdot a^{(l-1)} + b^{(l)})$$

   Where $a^{(l)}$ is the activation of layer l, $W^{(l)}$ are the weights, and f is the activation function.

2. **Loss Function**:
   - Mean Squared Error for regression:

$$L = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

   - Cross-Entropy Loss for classification:

$$L = -\frac{1}{n} \sum_{i=1}^{n} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

3. **Backpropagation**: Update weights using gradient descent:

$$W^{(l)} = W^{(l)} - \eta \frac{\partial L}{\partial W^{(l)}}$$

   Where η is the learning rate.

---

4. Python Implementation

Dataset: MNIST Dataset (Digit Classification).

```
# Import Libraries
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

# Load Dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Preprocess Data
X_train, X_test = X_train / 255.0, X_test / 255.0  # Normalize
y_train, y_test = to_categorical(y_train), to_categorical(y_test)  # One-hot
encoding

# Define ANN Model
model = Sequential([
    Flatten(input_shape=(28, 28)),  # Input Layer
    Dense(128, activation='relu'),  # Hidden Layer
    Dense(10, activation='softmax')  # Output Layer
])

# Compile Model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=
['accuracy'])

# Train Model
model.fit(X_train, y_train, epochs=5, batch_size=32)

# Evaluate Model
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {accuracy:.2f}")
```

---

5. Dataset File and Output

The **MNIST dataset** is included in `tensorflow.keras.datasets` and does not require external downloads.

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 10s 1us/step
Epoch 1/5
1875/1875 [==============================] - 5s 2ms/step - loss: 0.2570 - accuracy: 0.9268
Epoch 2/5
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1108 - accuracy: 0.9679
Epoch 3/5
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0750 - accuracy: 0.9774
Epoch 4/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0570 - accuracy: 0.9826
Epoch 5/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0441 - accuracy: 0.9863
313/313 [==============================] - 1s 2ms/step - loss: 0.0788 - accuracy: 0.9762
Test Accuracy: 0.98
```

K-Means Clustering

---

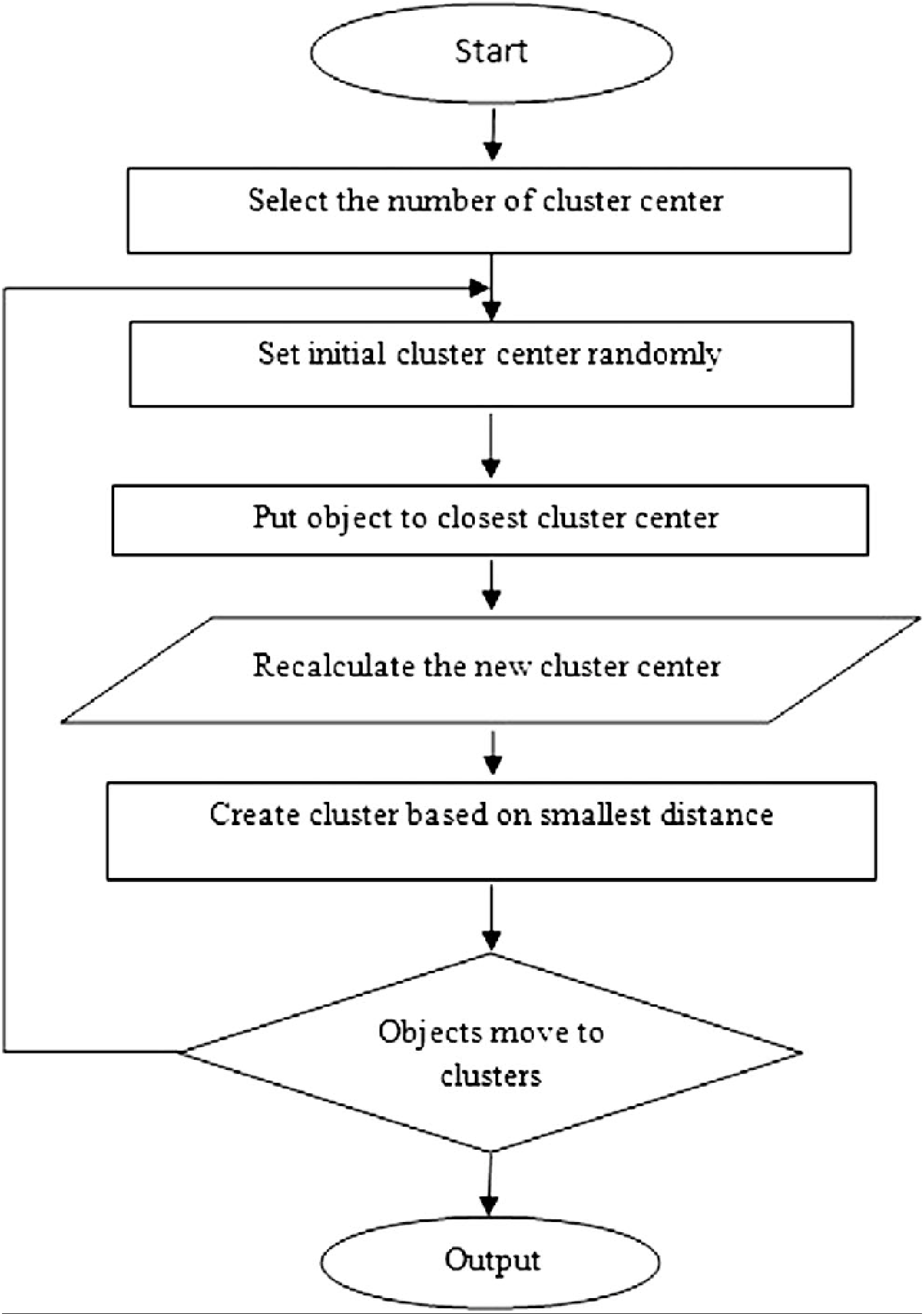
1. Description of Algorithm

K-Means is an unsupervised clustering algorithm used to group data into (k) clusters.

- **Working Principle**: It minimizes the variance within clusters by iteratively updating cluster centroids.
- **Applications**: Customer segmentation, image compression, and document clustering.
- **Advantages**: Simple and scalable.
- **Disadvantages**: Sensitive to initialization and the value of (k), struggles with non-spherical clusters.

---

2. Flow Diagram



3. Mathematical Model

1. **Cluster Assignment**:

   Assign each data point x_i to the cluster with the nearest centroid:

$$C_i = \arg\min_k ||x_i - \mu_k||^2$$

   Where μ_k is the centroid of cluster k.

2. **Centroid Update**:

   Update centroids as the mean of points in each cluster:

$$\mu_k = \frac{1}{|C_k|} \sum_{x \in C_k} x$$

3. **Stopping Criterion**:

   Stop when centroids no longer change or after a maximum number of iterations.

3. **Stopping Criterion**:
   Stop when centroids no longer change or after a maximum number of iterations.

---

4. Python Implementation

Dataset: Iris Dataset (unsupervised clustering example).

```
# Import Libraries
from sklearn.cluster import KMeans
from sklearn import datasets
import matplotlib.pyplot as plt
import seaborn as sns

# Load Dataset
iris = datasets.load_iris()
X = iris.data

# Apply K-Means Clustering
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans.fit(X)
labels = kmeans.labels_

# Visualize Clustering (2D Plot)
plt.figure(figsize=(8, 6))
sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=labels, palette="viridis", s=100)
plt.title("K-Means Clustering on Iris Dataset")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend(title="Cluster")
plt.show()
```

5. Dataset File and Output

The **Iris dataset** is included in `sklearn.datasets` and does not require external downloads.

Hierarchical Clustering

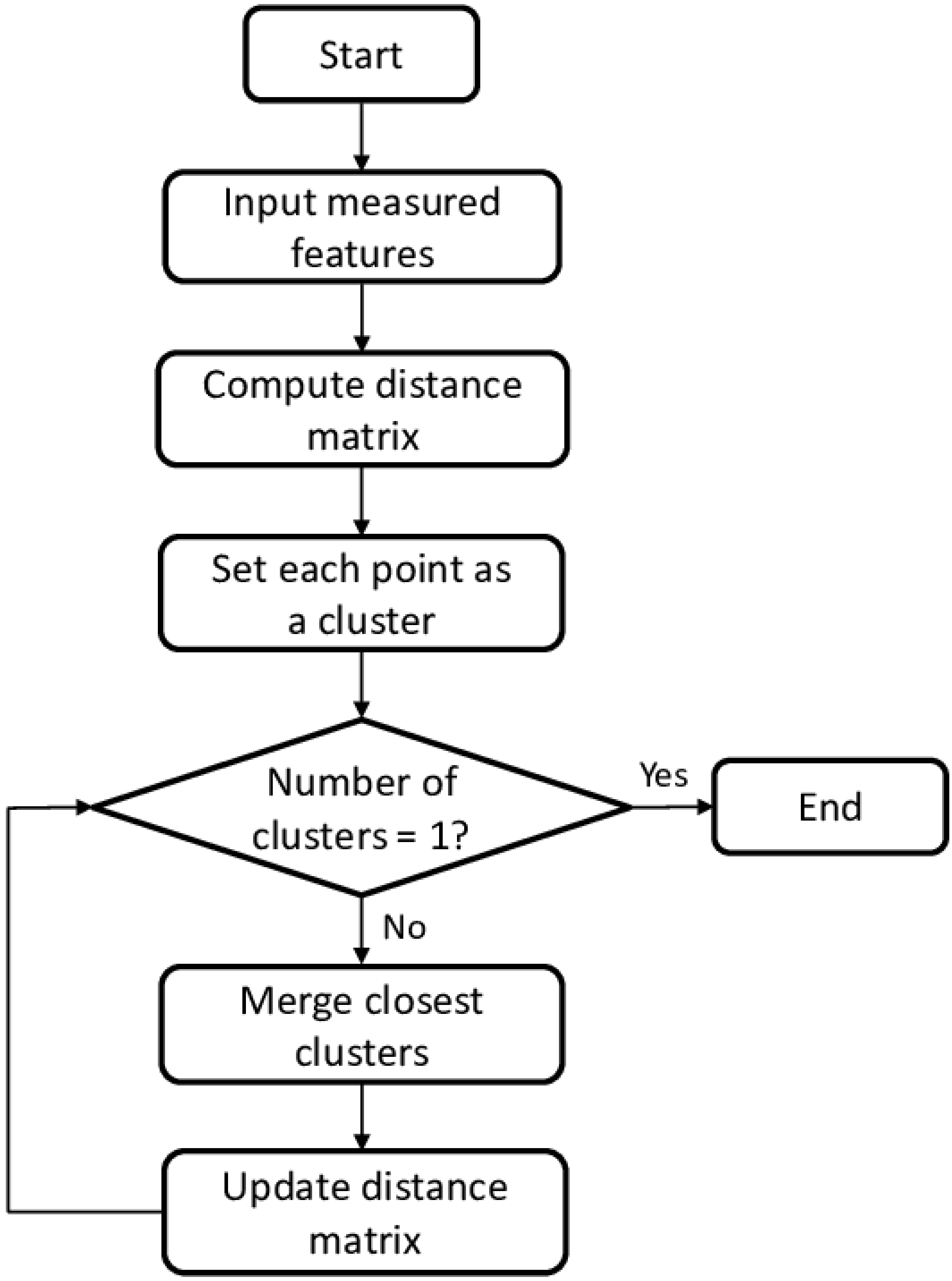
1. Description of Algorithm

Hierarchical Clustering is an unsupervised learning algorithm that builds a hierarchy of clusters by either:

1. **Agglomerative (Bottom-Up)**: Each data point starts as its own cluster and merges until one cluster is formed.
2. **Divisive (Top-Down)**: Starts with one cluster containing all points and splits recursively into smaller clusters.

- **Applications**: Gene sequence analysis, customer segmentation, document clustering.
- **Advantages**: No need to predefine the number of clusters, provides a dendrogram for better insights.
- **Disadvantages**: Computationally expensive for large datasets.

2. Flow Diagram



3. Mathematical Model

1. **Distance Metrics:**
   - **Euclidean Distance:**

$$d(x, y) = \sqrt{\sum (x_i - y_i)^2}$$

   - **Manhattan Distance:**

$$d(x, y) = \sum |x_i - y_i|$$

2. **Linkage Criteria:**

- **Single Linkage**: Nearest neighbor distance between clusters.
- **Complete Linkage**: Farthest neighbor distance between clusters.
- **Average Linkage**: Mean distance between all points in clusters.

3. **Dendrogram**: A tree-like diagram representing the hierarchical structure of clusters.

4. Python Implementation

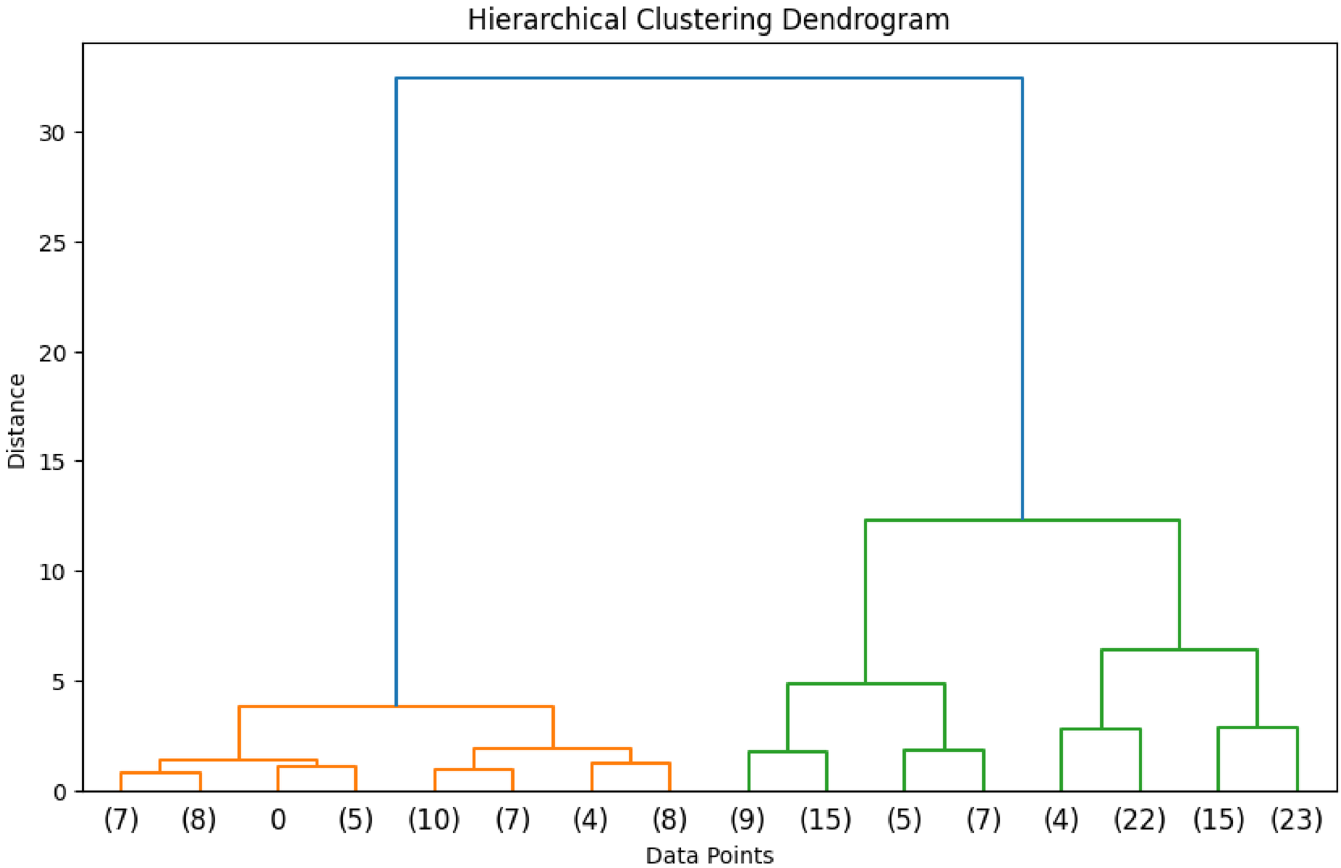
Dataset: Iris Dataset (unsupervised clustering example).

```python
# Import Libraries
from sklearn.datasets import load_iris
from scipy.cluster.hierarchy import dendrogram, linkage
from scipy.spatial.distance import pdist
import matplotlib.pyplot as plt

# Load Dataset
iris = load_iris()
X = iris.data

# Calculate Linkage Matrix
linkage_matrix = linkage(X, method='ward')  # Ward's method minimizes variance within clusters

# Plot Dendrogram
plt.figure(figsize=(10, 6))
dendrogram(linkage_matrix, truncate_mode='level', p=3, labels=iris.target)
plt.title("Hierarchical Clustering Dendrogram")
plt.xlabel("Data Points")
plt.ylabel("Distance")
plt.show()
```

---

5. Dataset File and Output

The **Iris dataset** is included in `sklearn.datasets` and does not require external downloads.



Principal Component Analysis (PCA)

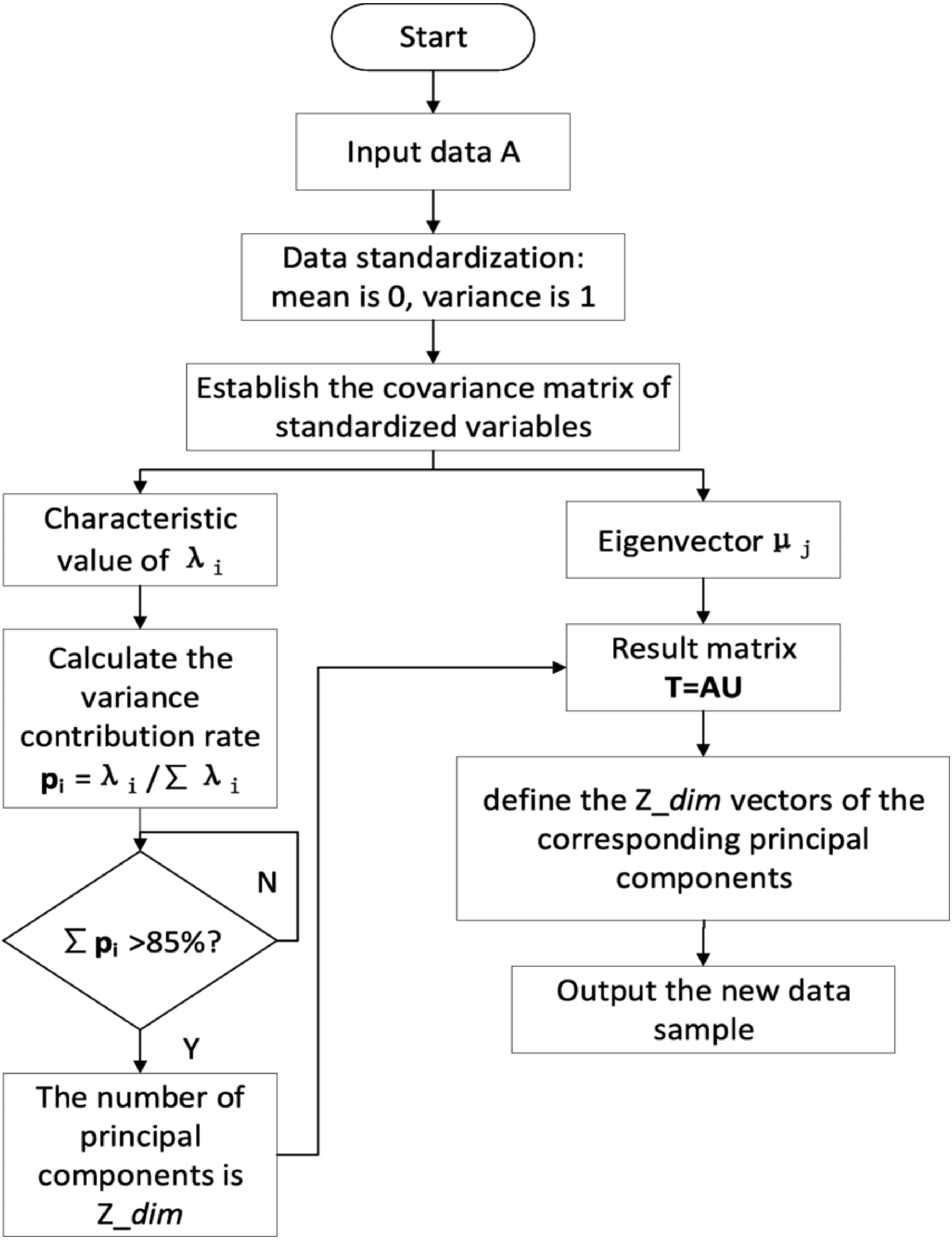
1. Description of Algorithm

Principal Component Analysis (PCA) is a dimensionality reduction technique used to transform high-dimensional data into a lower-dimensional space while preserving as much variance as possible.

- **Working Principle**: It identifies directions (principal components) in which the data varies the most and projects data onto these axes.
- **Applications**: Data visualization, noise reduction, feature extraction.
- **Advantages**: Simplifies data without much information loss.
- **Disadvantages**: Sensitive to scaling, assumes linearity.

---

2. Flow Diagram


Start
Input data A
Data standardization: mean is 0, variance is 1
Establish the covariance matrix of standardized variables
Characteristic value of λ i
Eigenvector μ j
Calculate the variance contribution rate
pi = λ i / Σ λ i
Result matrix
T=AU
define the Z_dim vectors of the corresponding principal components
N
Σ pi >85%?
Output the new data sample
Y
The number of principal components is Z_dim

3. Mathematical Model

1. **Standardization**: Center data by subtracting the mean and dividing by standard deviation:

$$Z = \frac{X - \mu}{\sigma}$$

2. **Covariance Matrix**:

$$C = \frac{1}{n} Z^T Z$$

3. **Eigen Decomposition**: Compute eigenvalues and eigenvectors of the covariance matrix.

4. **Projection**: Project data onto the top k eigenvectors:

$$Z_{PCA} = Z \cdot W$$

Where W contains the top k eigenvectors.

---

4. Python Implementation

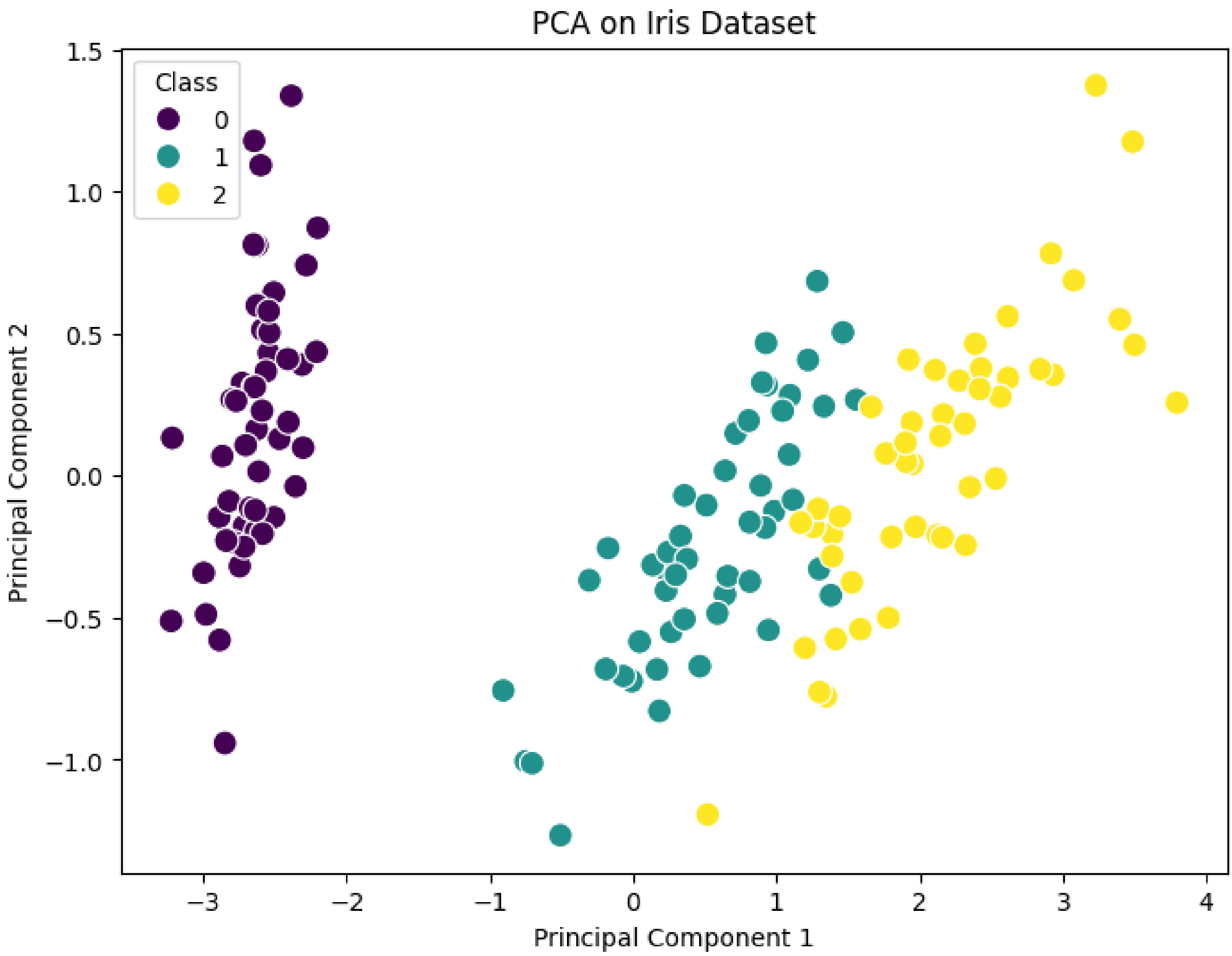
Dataset: Iris Dataset (dimensionality reduction example).

```
# Import Libraries
from sklearn.decomposition import PCA
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
import seaborn as sns

# Load Dataset
iris = load_iris()
X, y = iris.data, iris.target

# Apply PCA
pca = PCA(n_components=2)  # Reduce to 2 dimensions for visualization
X_pca = pca.fit_transform(X)

# Plot PCA Results
plt.figure(figsize=(8, 6))
sns.scatterplot(x=X_pca[:, 0], y=X_pca[:, 1], hue=y, palette='viridis', s=100)
plt.title("PCA on Iris Dataset")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.legend(title="Class")
plt.show()
```

---

5. Dataset File and Output

The **Iris dataset** is included in `sklearn.datasets` and does not require external downloads.



Reinforcement Learning (Q-Learning)

---

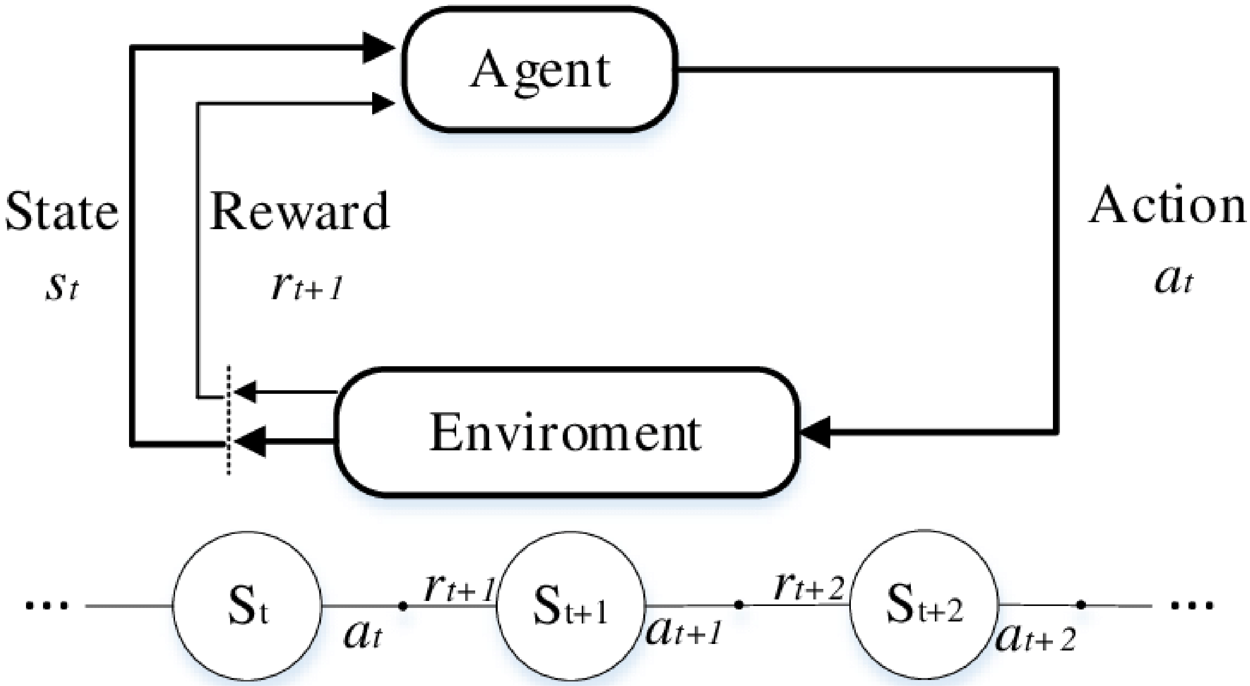
1. Description of Algorithm

Q-Learning is a model-free reinforcement learning algorithm used to learn an optimal policy for an agent interacting with an environment by using rewards.

- **Working Principle**: It uses a Q-table to store the expected utility of taking a given action in a given state.
- **Applications**: Game playing, robotics, autonomous systems.
- **Advantages**: Does not require a model of the environment, works for discrete spaces.
- **Disadvantages**: Inefficient for large state spaces, requires tuning of hyperparameters.

---

2. Flow Diagram



3. Mathematical Model

- **Bellman Equation:**

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a} Q(s',a') - Q(s,a)\right]$$

Where:

- $Q(s,a)$: Q-value for state s and action a
- α: Learning rate
- γ: Discount factor
- r: Reward for action a in state s
- s': Next state

4. Python Implementation

Environment: FrozenLake (OpenAI Gym).

```
import numpy as np
import random

# Define the environment
grid_size = 4
```

```
goal_state = (3, 3)
obstacles = [(1, 1), (2, 2)]
actions = ['up', 'down', 'left', 'right']

# Helper functions
def is_valid_state(state):
    return (
        0 <= state[0] < grid_size and
        0 <= state[1] < grid_size and
        state not in obstacles
    )

def get_next_state(state, action):
    if action == 'up':
        next_state = (state[0] - 1, state[1])
    elif action == 'down':
        next_state = (state[0] + 1, state[1])
    elif action == 'left':
        next_state = (state[0], state[1] - 1)
    elif action == 'right':
        next_state = (state[0], state[1] + 1)
    else:
        next_state = state

    return next_state if is_valid_state(next_state) else state

def get_reward(state):
    return 10 if state == goal_state else -1

# Initialize Q-Table
q_table = {}
for i in range(grid_size):
    for j in range(grid_size):
        q_table[(i, j)] = {a: 0 for a in actions}

# Training parameters
episodes = 500
learning_rate = 0.1
discount_factor = 0.9
epsilon = 0.1

# Q-Learning algorithm
for episode in range(episodes):
    state = (0, 0)  # Start state
    done = False

    while not done:
        # Choose action: ε-Greedy
        if random.uniform(0, 1) < epsilon:
            action = random.choice(actions)
        else:
            action = max(q_table[state], key=q_table[state].get)

        # Take action
```

```
        next_state = get_next_state(state, action)
        reward = get_reward(next_state)

        # Update Q-value
        q_table[state][action] += learning_rate * (
            reward + discount_factor * max(q_table[next_state].values()) -
q_table[state][action]
        )

        # Move to next state
        state = next_state

        # Check if goal is reached
        if state == goal_state:
            done = True

print("Q-Table after training:")
for state, actions in q_table.items():
    print(state, actions)

# Test the policy
print("\nTesting Optimal Policy:")
state = (0, 0)
path = [state]

while state != goal_state:
    action = max(q_table[state], key=q_table[state].get)
    state = get_next_state(state, action)
    path.append(state)

print("Optimal Path:", path)
```

---

5. Dataset and Output

This is custom dataset

```
Q-Table after training:
(0, 0) {'up': -0.9118628592883264, 'down': 1.80979999999998, 'left': -0.1444590790238352, 'right': -2.1199025107974063}
(0, 1) {'up': -1.798713721018097, 'down': -1.737660653908409, 'left': -1.6870615838083391, 'right': -0.6399944385122414}
(0, 2) {'up': -0.9640418879973849, 'down': -0.965447653582764, 'left': -1.2828526442924901, 'right': 1.5567401981145639}
(0, 3) {'up': -0.41199317955240256, 'down': 4.341863975362333, 'left': -0.5254557836312632, 'right': -0.44857403083725556}
(1, 0) {'up': -0.361176909860538, 'down': 3.1219999999999857, 'left': 0.708677152991894, 'right': 0.3211472309597183}
(1, 1) {'up': 0, 'down': 0, 'left': 0, 'right': 0}
(1, 2) {'up': -0.6552228548400392, 'down': -0.48843811000000004, 'left': -0.490099501, 'right': 0.5899618096216224}
(1, 3) {'up': -0.107281128739345, 'down': 7.287380786986903, 'left': -0.32185359100000005, 'right': 0.2840951055399753}
(2, 0) {'up': 0.1735146325249885, 'down': 4.579999999999986, 'left': 1.6320026256268978, 'right': 1.8026332590508576}
(2, 1) {'up': -0.490099501, 'down': 5.335997263635556, 'left': 0.7202810390822216, 'right': -0.4921261388455904}
(2, 2) {'up': 0, 'down': 0, 'left': 0, 'right': 0}
(2, 3) {'up': 0.3660257881692694, 'down': 9.835767967317393, 'left': 0, 'right': 1.4299621975994143}
(3, 0) {'up': 2.040732969178272, 'down': 3.618400691331263, 'left': 3.0638307416756962, 'right': 6.19999999999989}
(3, 1) {'up': 1.481503044450837, 'down': 5.1081444792785105, 'left': 3.1186690992524784, 'right': 7.999999999999991}
(3, 2) {'up': 5.465409103242514, 'down': 4.846085506515803, 'left': 4.5838151229851976, 'right': 9.999999999999993}
(3, 3) {'up': 0, 'down': 0, 'left': 0, 'right': 0}

Testing Optimal Policy:
Optimal Path: [(0, 0), (1, 0), (2, 0), (3, 0), (3, 1), (3, 2), (3, 3)]
```