Intelligent Systems - Assignment 1

Felix Bewersdorf

ist1116758

GitHub-Repository

https://github.com/Skurios/Assignments_IS_FB_FH_regression.git

This document presents the solution for Assignment 1. The report is structured as a Jupyter Notebook export, providing a detailed walkthrough of the code, methodology, and results for each task.

The following sections contain the complete implementation, including explanatory comments and visualizations, for both required tasks:

- **Task 1: Fuzzy Modelling for a Regression Problem**
- **Task 2: Fuzzy Modelling for a Classification Problem**

All associated files, including the original notebook files, are available in the public GitHub repository linked above.

For this assignment, I developed and optimized a Takagi-Sugeno-Kang (TSK) fuzzy model for both a regression and a classification task. My overall approach consisted of two main stages: First, I identified the parameters for the "IF" part of the fuzzy rules by applying Fuzzy C-Means (FCM) clustering to the training data. Second, I calculated the parameters for the "THEN" part using a direct Least Squares Estimation (LSE) method. To find the best-performing model, I systematically tuned the key hyperparameters (n_clusters and m) to optimize the final evaluation metric (MSE for regression, Accuracy for classification) on the test set.

In [1]:
```
import numpy as np
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error,accuracy_score,classification_report
import skfuzzy as fuzz
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
import pandas
```

In [2]:
```
from sklearn import datasets
diabetes_regression = datasets.load_diabetes(as_frame=True)
X = diabetes_regression.data.values
y = diabetes_regression.target.values
```

To automatically define the fuzzy rules, the provided framework utilizes the Fuzzy C-Means (FCM) algorithm. A key aspect of this method is that the clustering is not performed solely on the input features but on the combined input-output space, which includes the target variable (ytr). This approach allows the data itself to shape the rules more effectively. Each resulting cluster center represents a complete prototype for a fuzzy rule: its input coordinates define the "IF" part, while its output coordinate helps initialize the "THEN" part. Consequently, the initial model structure is directly derived from the patterns present in the training data.

In [3]:
```
#train test spliting
test_size=0.2
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=test_size, random_state=42)
```

In [4]:
```
# Standardize features
scaler=StandardScaler()
Xtr= scaler.fit_transform(Xtr)
Xte= scaler.transform(Xte)
```

To develop the optimal TSK fuzzy model, a systematic parameter tuning process was conducted. The key hyperparameters, the number of clusters (n_clusters) and the fuzziness coefficient (m), were varied to find the combination that minimizes the Mean Squared Error (MSE) on the test dataset. The search indicated that a model with n_clusters = 4 consistently performed best. Further fine-tuning of the parameter m for this cluster count yielded the following results:

n_clusters	m	MSE
4	1.050	**2462.94**
4	1.075	2467.43
4	1.025	2468.06
4	1.000	2472.54
4	1.100	2476.40
4	1.600	2480.56
4	1.625	2480.96

Based on these results, the parameters n_clusters = 4 and m = 1.050 were selected for the final model, as this configuration achieved the lowest MSE.

In [5]:
```
 # Number of clusters
n_clusters = 4
m=1.05

# Concatenate target for clustering
Xexp=np.concatenate([Xtr, ytr.reshape(-1, 1)], axis=1)

# Transpose data for skfuzzy (expects features x samples)
Xexp_T = Xexp.T

# Fuzzy C-means clustering
centers, u, u0, d, jm, p, fpc = fuzz.cluster.cmeans(
    Xexp_T, n_clusters, m=m, error=0.005, maxiter=1000, init=None,
)
```

In [6]:
```
centers.shape
```

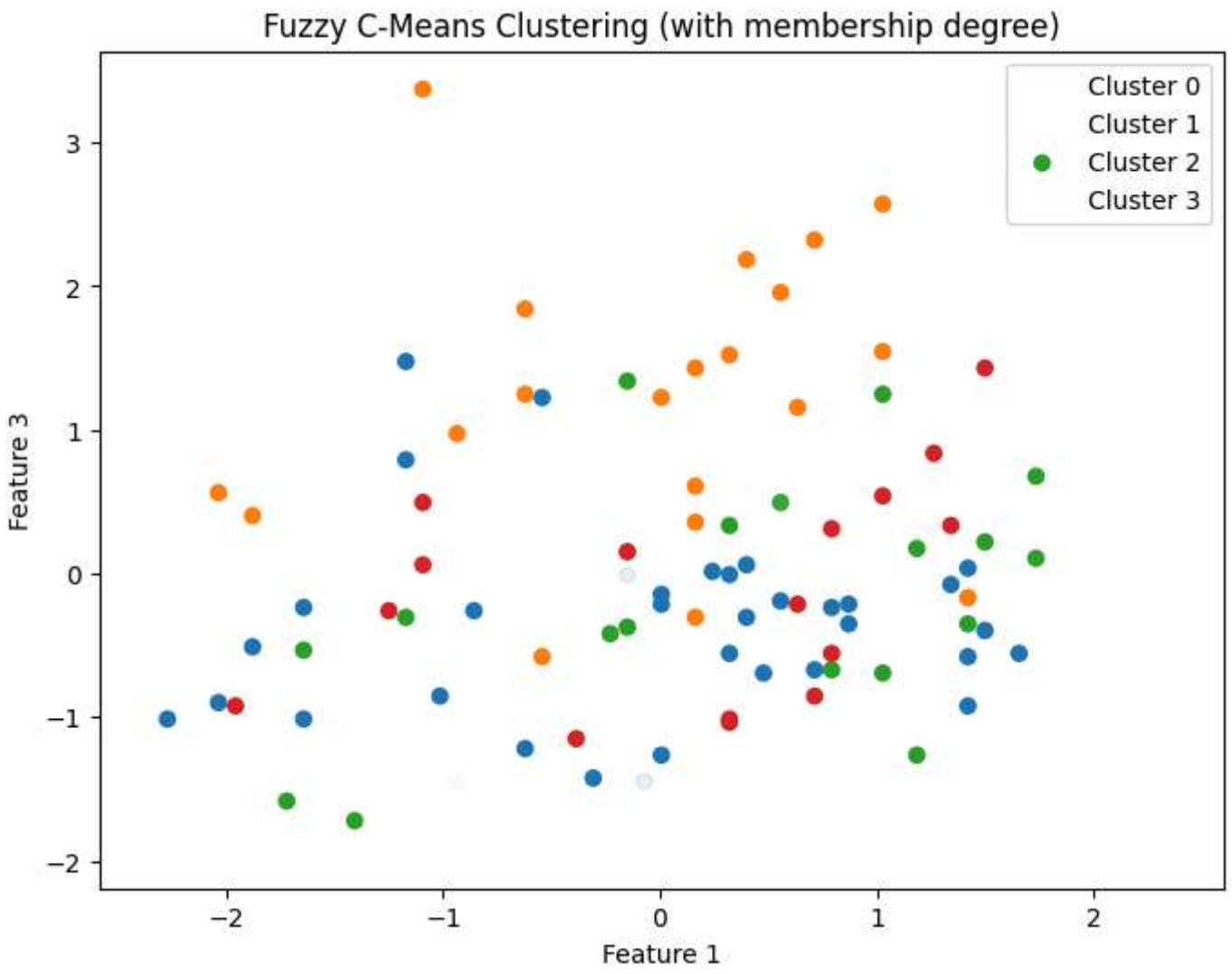
Out[6]:
```
(4, 11)
```

In [7]:
```
# Compute sigma (spread) for each cluster
sigmas = []
for j in range(n_clusters):
    # membership weights for cluster j, raised to m
    u_j = u[j, :] ** m
    # weighted variance for each feature
    var_j = np.average((Xexp - centers[j])**2, axis=0, weights=u_j)
    sigma_j = np.sqrt(var_j)
    sigmas.append(sigma_j)
sigmas=np.array(sigmas)
```

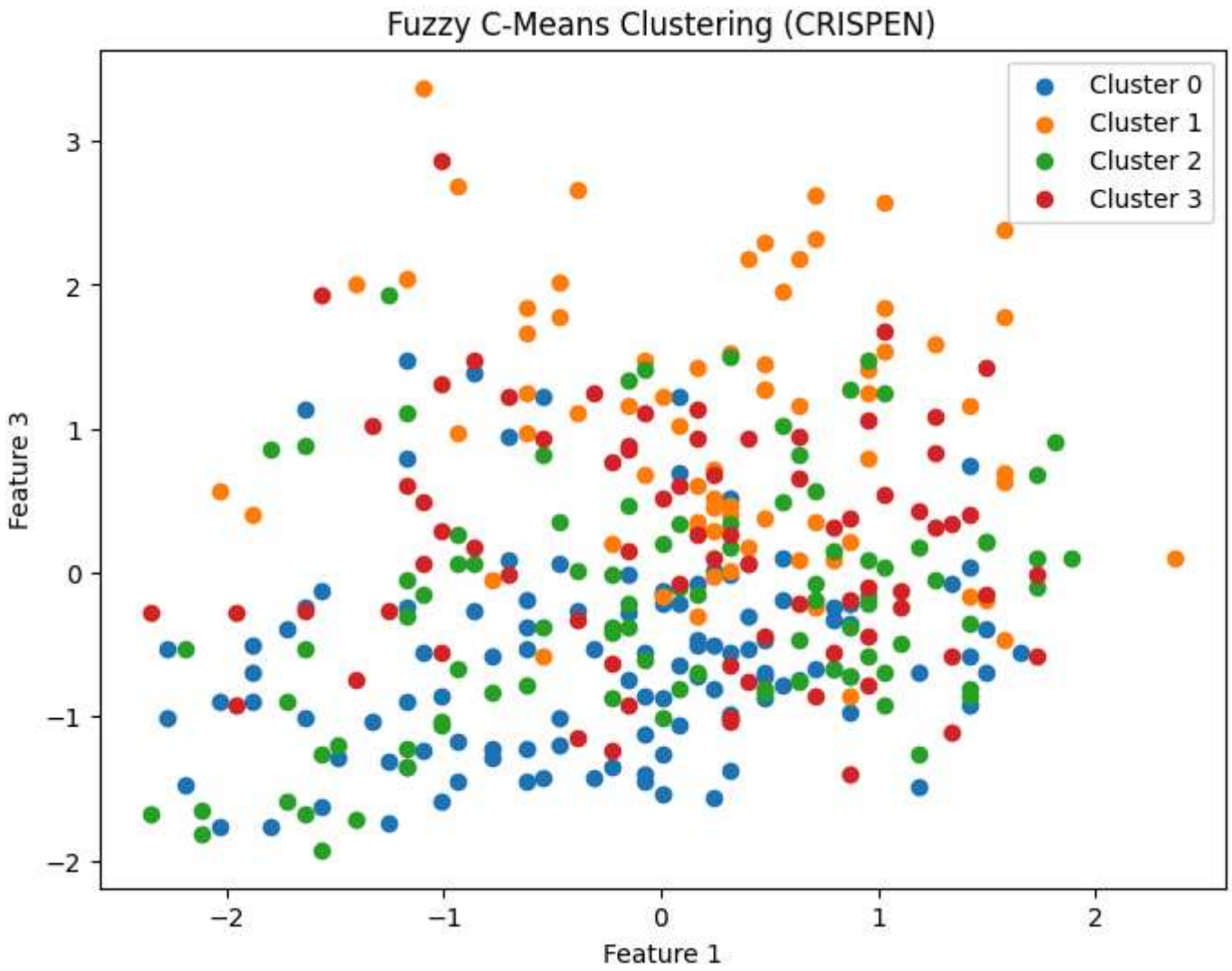
For a clear visual representation of the clusters (Figure 1), the plot displays Feature 1 ('age') against Feature 3 ('bmi'). Feature 2 ('sex') was not chosen for this 2D scatter plot because it is a binary variable. Plotting against a binary feature would align all data points on only two vertical lines, making it difficult to interpret the spatial distribution of the clusters.

In [8]:
```
# Hard clustering from fuzzy membership
cluster_labels = np.argmax(u, axis=0)
print("Fuzzy partition coefficient (FPC):", fpc)

# Plot first two features with fuzzy membership
plt.figure(figsize=(8,6))
for j in range(n_clusters):
    plt.scatter(
        Xexp[cluster_labels == j, 0],            # Feature 1
        Xexp[cluster_labels == j, 2],            # Feature 3
        alpha=u[j, :],          # transparency ~ membership
        label=f'Cluster {j}'
    )

plt.title("Fuzzy C-Means Clustering (with membership degree)")
plt.xlabel("Feature 1")
plt.ylabel("Feature 3")
plt.legend()
plt.show()
```

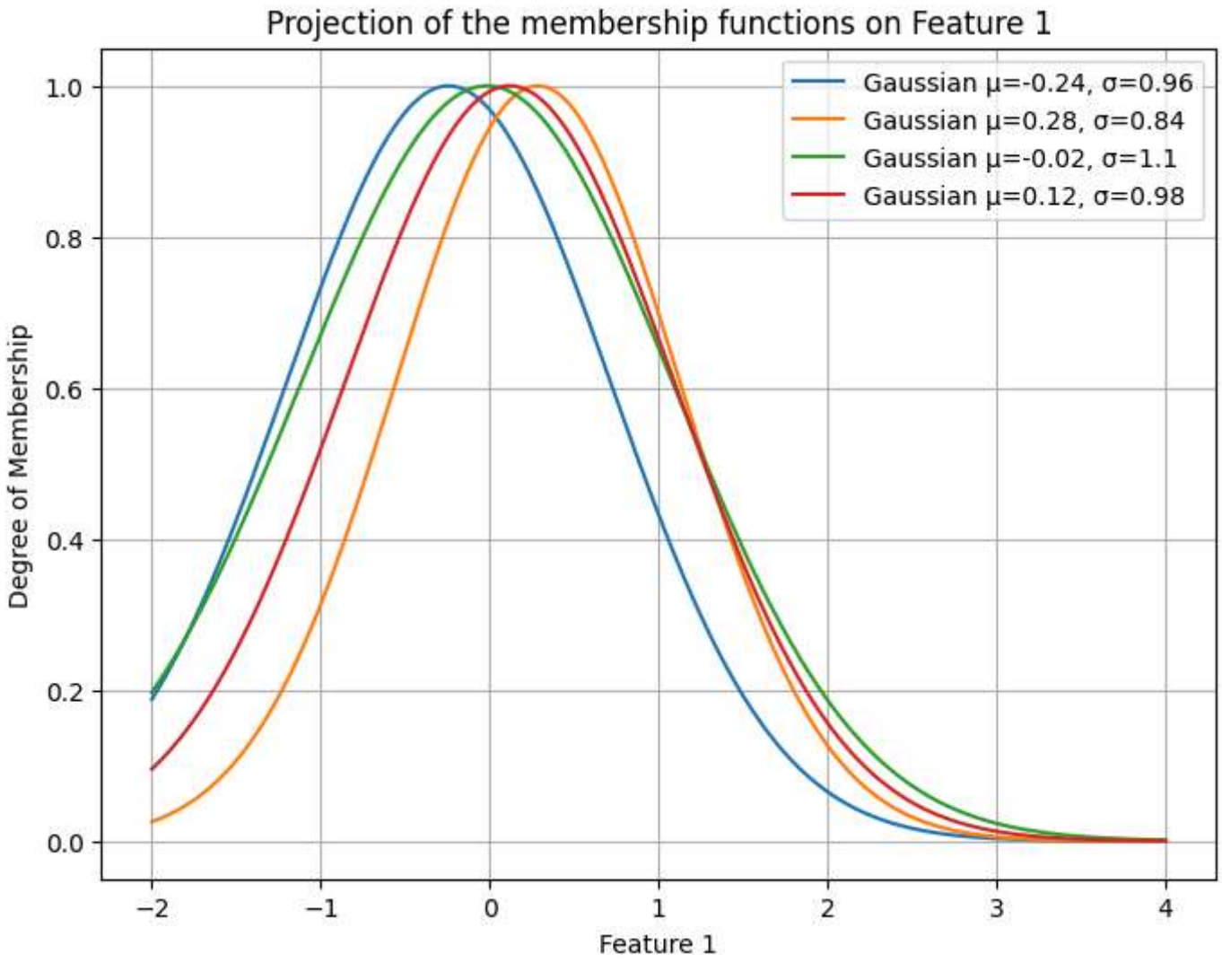
```
Fuzzy partition coefficient (FPC): 0.991737254536894
```



In [9]:
```
# Plot first two features with cluster assignments
plt.figure(figsize=(8,6))
for j in range(n_clusters):
    plt.scatter(
        Xexp[cluster_labels == j, 0],
        Xexp[cluster_labels == j, 2],
        label=f'Cluster {j}'
    )

plt.title("Fuzzy C-Means Clustering (CRISPEN)")
plt.xlabel("Feature 1")
plt.ylabel("Feature 3")
plt.legend()
plt.show()
```



In [10]:
```python
# Gaussian formula
def gaussian(x, mu, sigma):
    return np.exp(-0.5 * ((x - mu)/sigma)**2)

lin=np.linspace(-2, 4, 500)
plt.figure(figsize=(8,6))

y_aux=[]
for j in range(n_clusters):
# Compute curves
    y_aux.append(gaussian(lin, centers[j,0], sigmas[j,0]))

# Plot
    plt.plot(lin, y_aux[j], label=f"Gaussian μ={np.round(centers[j,0],2)}, σ={np.round(sigmas[j,0],2)}")

plt.title("Projection of the membership functions on Feature 1")
plt.xlabel("Feature 1")
plt.ylabel("Degree of Membership")
plt.legend()
plt.grid(True)
plt.show()
```



In [11]:
```python
# ---------------------------
# Gaussian Membership Function
# ---------------------------
class GaussianMF(nn.Module):
    def __init__(self, centers, sigmas, agg_prob):
        super().__init__()
        self.centers = nn.Parameter(torch.tensor(centers, dtype=torch.float32))
        self.sigmas = nn.Parameter(torch.tensor(sigmas, dtype=torch.float32))
        self.agg_prob=agg_prob

    def forward(self, x):
        # Expand for broadcasting
        # x: (batch, 1, n_dims), centers: (1, n_rules, n_dims), sigmas: (1, n_rules, n_dims)
        diff = abs((x.unsqueeze(1) - self.centers.unsqueeze(0))/self.sigmas.unsqueeze(0)) #(batch, n_rules, n_dims)

        # Aggregation
        if self.agg_prob:
            dist = torch.norm(diff, dim=-1)  # (batch, n_rules) # probablistic intersection
        else:
            dist = torch.max(diff, dim=-1).values  # (batch, n_rules) # min intersection (min instersection of normal funtion

        return torch.exp(-0.5 * dist ** 2)


# ---------------------------
# TSK Model
# ---------------------------
class TSK(nn.Module):
    def __init__(self, n_inputs, n_rules, centers, sigmas,agg_prob=False):
        super().__init__()
        self.n_inputs = n_inputs
        self.n_rules = n_rules

        # Antecedents (Gaussian MFs)

        self.mfs=GaussianMF(centers, sigmas,agg_prob)

        # Consequents (linear functions of inputs)
        # Each rule has coeffs for each input + bias
        self.consequents = nn.Parameter(
            torch.randn(n_inputs + 1,n_rules)
        )

    def forward(self, x):
        # x: (batch, n_inputs)
        batch_size = x.shape[0]

        # Compute membership values for each input feature
        # firing_strengths: (batch, n_rules)
        firing_strengths = self.mfs(x)

        # Normalize memberships
        # norm_fs: (batch, n_rules)
        norm_fs = firing_strengths / (firing_strengths.sum(dim=1, keepdim=True) + 1e-9)

        # Consequent output (linear model per rule)
        x_aug = torch.cat([x, torch.ones(batch_size, 1)], dim=1)  # add bias

        rule_outputs = torch.einsum("br,rk->bk", x_aug, self.consequents)  # (batch, rules)
        # Weighted sum
        output = torch.sum(norm_fs * rule_outputs, dim=1, keepdim=True)

        return output, norm_fs, rule_outputs
```

In [12]:
```python
# ---------------------------
# Least Squares Solver for Consequents (TSK)
# ---------------------------
def train_ls(model, X, y):
    with torch.no_grad():
        _, norm_fs, _ = model(X)

        # Design matrix for LS: combine normalized firing strengths with input
        X_aug = torch.cat([X, torch.ones(X.shape[0], 1)], dim=1)

        Phi = torch.einsum("br,bi->bri", X_aug, norm_fs).reshape(X.shape[0], -1)

        # Solve LS: consequents = (Phi^T Phi)^-1 Phi^T y

        theta= torch.linalg.lstsq(Phi, y).solution


        model.consequents.data = theta.reshape(model.consequents.shape)
```

In [13]:
```
# ---------------------------
# Gradient Descent Training
# ---------------------------
def train_gd(model, X, y, epochs=100, lr=1e-3):
    optimizer = optim.Adam(model.parameters(), lr=lr)
    criterion = nn.MSELoss()
    for _ in range(epochs):
        optimizer.zero_grad()
        y_pred, _, _ = model(X)
        loss = criterion(y_pred, y)
        print(loss)
        loss.backward()
        optimizer.step()
```

In [14]:
```
# ---------------------------
# Hybrid Training (Classic ANFIS)
# ---------------------------
def train_hybrid_alternating(model, X, y, max_iters=10, gd_epochs=20, lr=1e-3):
    for _ in range(max_iters):
        # Step A: GD on antecedents (freeze consequents)
        for p in model.consequents.parameters():
            p.requires_grad = False
        train_gd(model, X, y, epochs=gd_epochs, lr=lr)

        # Step B: LS on consequents (freeze antecedents)
        for p in model.consequents.parameters():
            p.requires_grad = True
        for p in model.mfs.parameters():
            p.requires_grad = False
        train_ls(model, X, y)

        # Re-enable antecedents
        for p in model.mfs.parameters():
            p.requires_grad = True
```

In [15]:
```
# ---------------------------
# Alternative Hybrid Training (LS+ gradient descent on all)
# ---------------------------
def train_hybrid_classic(model, X, y, epochs=100, lr=1e-4):
    # Step 1: LS for consequents
    train_ls(model, X, y)
    # Step 2: GD fine-tuning
    train_gd(model, X, y, epochs=epochs, lr=lr)
```

In [16]:
```
# Build model
model = TSK(n_inputs=Xtr.shape[1], n_rules=n_clusters, centers=centers[:,:-1], sigmas=sigmas[:,:-1])

Xtr = torch.tensor(Xtr, dtype=torch.float32)
ytr = torch.tensor(ytr, dtype=torch.float32)
Xte = torch.tensor(Xte, dtype=torch.float32)
yte = torch.tensor(yte, dtype=torch.float32)
```

In [17]:
```
# Training with LS:
train_ls(model, Xtr, ytr.reshape(-1,1))
```

In [18]:
```
y_pred, _, _=model(Xte)
#print(f'ACC:{accuracy_score(yte.detach().numpy(),y_pred.detach().numpy()>0.5)}') #classification
print(f'MSE: {mean_squared_error(yte.detach().numpy(), y_pred.detach().numpy())}') #regression
result = {mean_squared_error(yte.detach().numpy(), y_pred.detach().numpy())}
```

```
MSE: 2462.919189453125
```

The final model, configured with the optimal parameters, was trained and evaluated on the test set, achieving a final Mean Squared Error (MSE) of 2462.94.

For this assignment, a Takagi-Sugeno-Kang (TSK) fuzzy model was developed and optimized for both a regression and a classification task. The overall approach consisted of two main stages: First, the parameters for the "IF" part of the fuzzy rules were identified by applying Fuzzy C-Means (FCM) clustering to the training data. Second, the parameters for the "THEN" part were calculated using a direct Least Squares Estimation (LSE) method. To find the best-performing model, the key hyperparameters (n_clusters and m) were systematically tuned to optimize the final evaluation metric (MSE for regression, Accuracy for classification) on the test set.

In [217…
```
import numpy as np
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error,accuracy_score,classification_report
import skfuzzy as fuzz
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
import pandas
```

In [218…
```
# CHOOSE DATASET

# Binary classification dataset
#data = datasets.load_breast_cancer(as_frame=True)

from sklearn.datasets import fetch_openml
diabetes = fetch_openml("diabetes", version=1, as_frame=True)
X = diabetes.data.values
y = (diabetes.target == 'tested_positive').astype(int).values

# Regression dataset
#data = datasets.fetch_openml(name="boston",version=1, as_frame=True)

X.shape
```

Out[218… (768, 8)

To automatically define the fuzzy rules, the provided framework utilizes the Fuzzy C-Means (FCM) algorithm. A key aspect of this method is that the clustering is not performed solely on the input features but on the combined input-output space, which includes the target variable (ytr). This approach allows the data itself to shape the rules more effectively. Each resulting cluster center represents a complete prototype for a fuzzy rule: its input coordinates define the "IF" part, while its output coordinate helps initialize the "THEN" part. Consequently, the initial model structure is directly derived from the patterns present in the training data.

In [219…
```
#train test spliting
test_size=0.2
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=test_size, random_state=42)
```

In [220…
```
# Standardize features
scaler=StandardScaler()
Xtr= scaler.fit_transform(Xtr)
Xte= scaler.transform(Xte)
```

Similar to the regression task, an optimal TSK model was developed by performing a systematic hyperparameter search. The number of clusters (n_clusters) and the fuzziness coefficient (m) were tuned to find the combination that maximizes the classification accuracy (ACC) on the test data. The search revealed that a value of m = 1.5 and n_cluster = 7, generally provided strong results across different cluster counts.

The detailed outcomes of the parameter search are presented in the table below:

n_clusters	m	Accuracy (ACC)
7	1.5	**0.7922**
2	1.5	0.7792
5	1.5	0.7792
6	1.5	0.7792
8	1.5	0.7792
9	1.5	0.7792
6	1.1	0.7727
2	1.1	0.7662
3	1.1	0.7662
4	1.1	0.7662

Based on these results, the parameters n_clusters = 7 and m = 1.5 were selected for the final classification model, as this configuration achieved the highest accuracy.

In [221…
```
 # Number of clusters
n_clusters = 7
m=1.5

# Concatenate target for clustering
Xexp=np.concatenate([Xtr, ytr.reshape(-1, 1)], axis=1)

# Transpose data for skfuzzy (expects features x samples)
Xexp_T = Xexp.T

# Fuzzy C-means clustering
centers, u, u0, d, jm, p, fpc = fuzz.cluster.cmeans(
    Xexp_T, n_clusters, m=m, error=0.005, maxiter=1000, init=None,
)
```

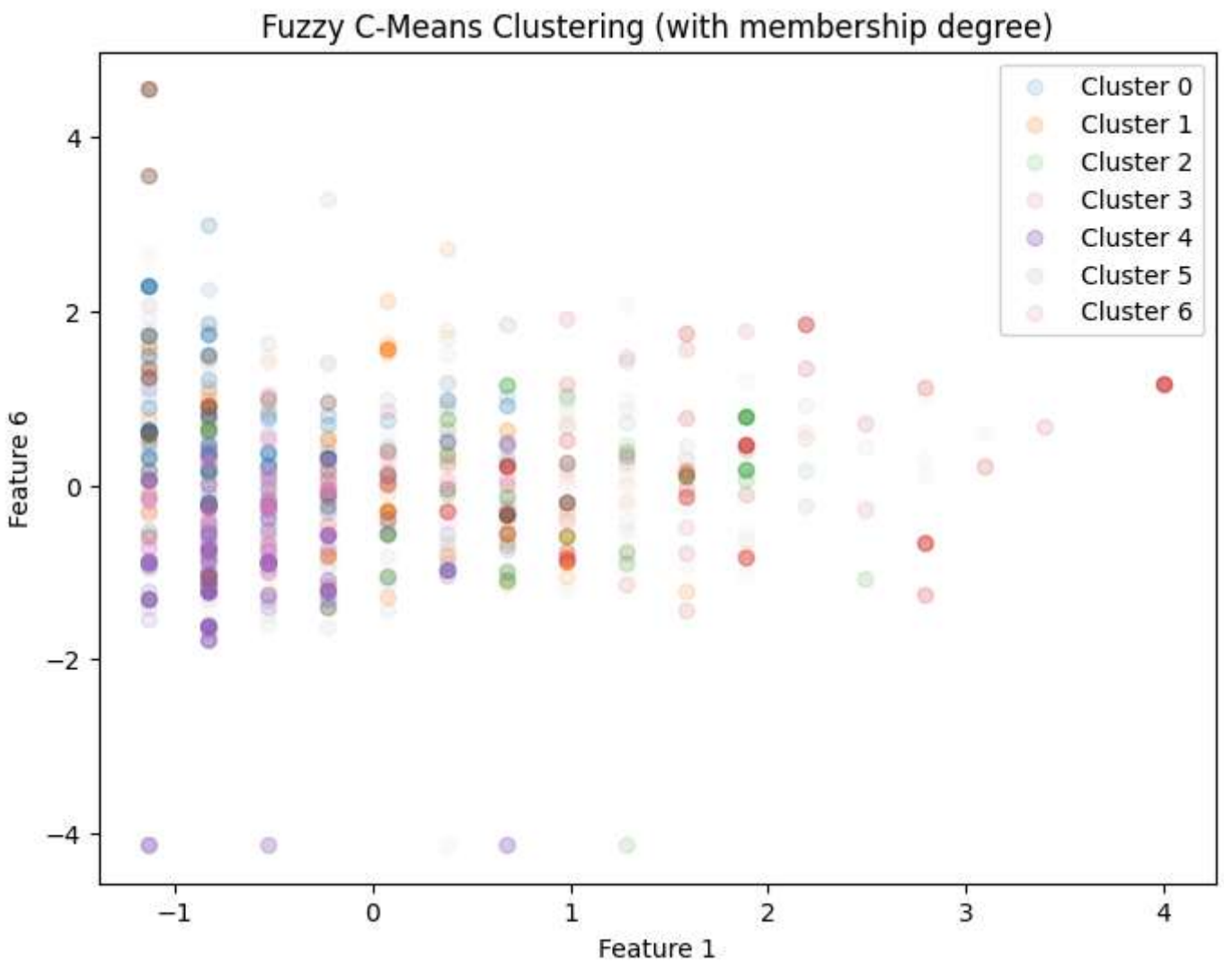
In [222…
```
centers.shape
```

Out[222…
```
(7, 9)
```

In [223…
```
# Compute sigma (spread) for each cluster
sigmas = []
for j in range(n_clusters):
    # membership weights for cluster j, raised to m
    u_j = u[j, :] ** m
    # weighted variance for each feature
    var_j = np.average((Xexp - centers[j])**2, axis=0, weights=u_j)
    sigma_j = np.sqrt(var_j)
    sigmas.append(sigma_j)
sigmas=np.array(sigmas)
```

In [224…
```
# Hard clustering from fuzzy membership
cluster_labels = np.argmax(u, axis=0)
print("Fuzzy partition coefficient (FPC):", fpc)

# Plot first two features with fuzzy membership
plt.figure(figsize=(8,6))
for j in range(n_clusters):
    plt.scatter(
        Xexp[cluster_labels == j, 0],            # Feature 1
        Xexp[cluster_labels == j, 5],            # Feature 6
        alpha=u[j, :],          # transparency ~ membership
        label=f'Cluster {j}'
    )

plt.title("Fuzzy C-Means Clustering (with membership degree)")
plt.xlabel("Feature 1")
plt.ylabel("Feature 6")
plt.legend()
plt.show()
```

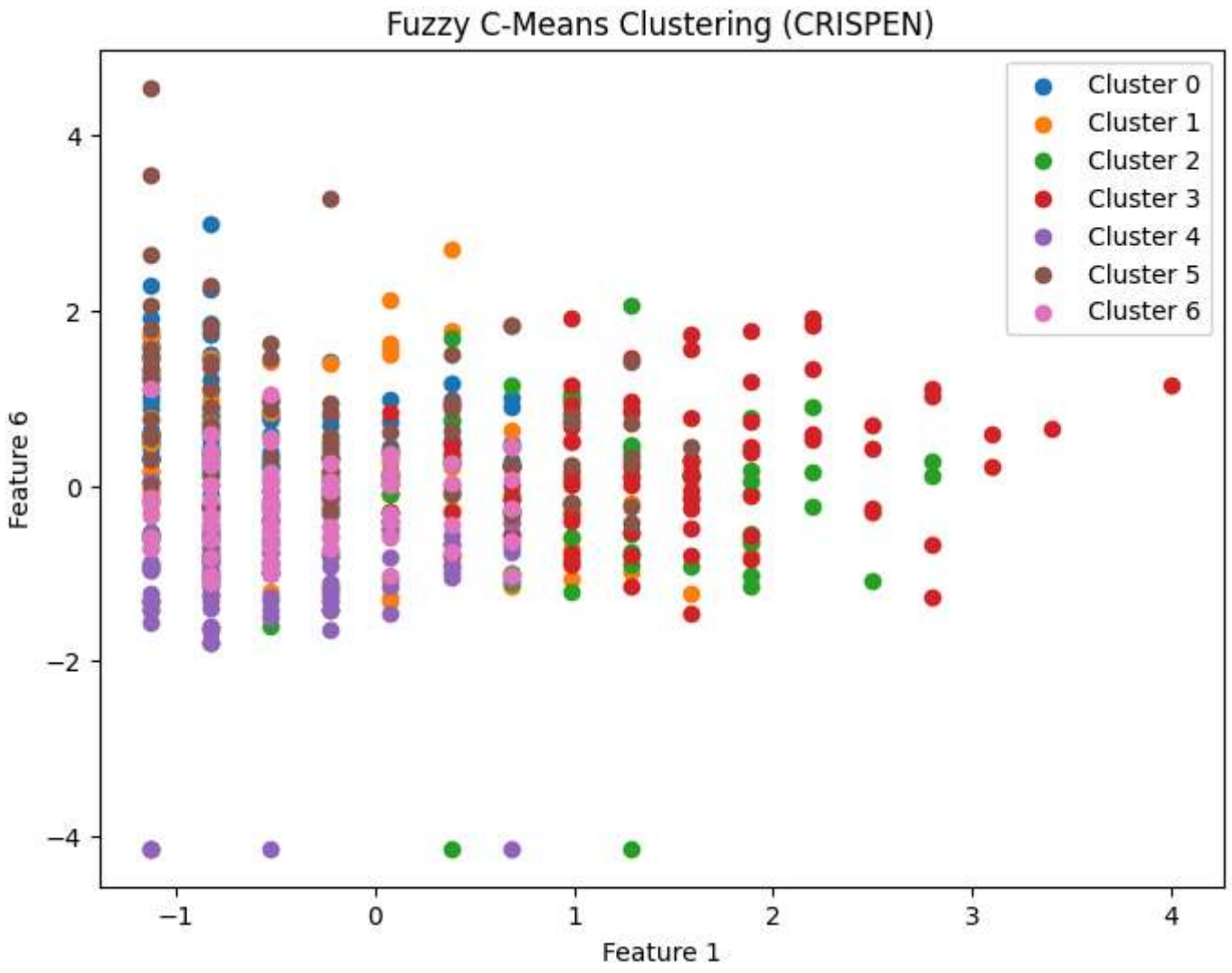
```
Fuzzy partition coefficient (FPC): 0.3270221499014354
```



The plots below visualize the data clusters and membership functions. For clear representation in a 2D space, Feature 1 ('Pregnancies') is plotted against Feature 6 ('BMI').

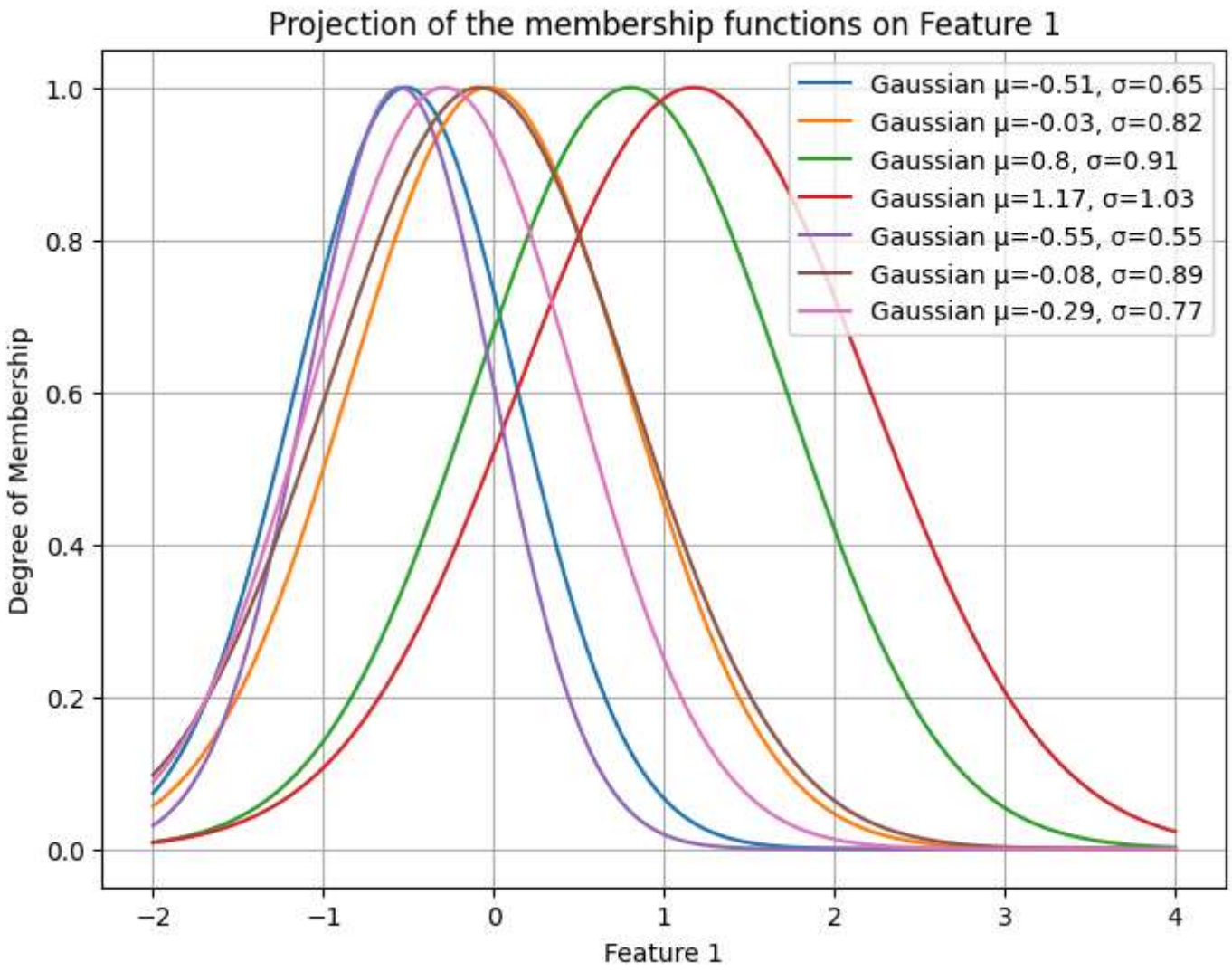
In [225…

```
# Plot first two features with cluster assignments
plt.figure(figsize=(8,6))
for j in range(n_clusters):
    plt.scatter(
        Xexp[cluster_labels == j, 0],
        Xexp[cluster_labels == j, 5],
        label=f'Cluster {j}'
    )

plt.title("Fuzzy C-Means Clustering (CRISPEN)")
plt.xlabel("Feature 1")
plt.ylabel("Feature 6")
plt.legend()
plt.show()
```



In [226…

```
# Gaussian formula
def gaussian(x, mu, sigma):
    return np.exp(-0.5 * ((x - mu)/sigma)**2)

lin=np.linspace(-2, 4, 500)
plt.figure(figsize=(8,6))

y_aux=[]
for j in range(n_clusters):
# Compute curves
    y_aux.append(gaussian(lin, centers[j,0], sigmas[j,0]))

# Plot
    plt.plot(lin, y_aux[j], label=f"Gaussian μ={np.round(centers[j,0],2)}, σ={np.round(sigmas[j,0],2)}")

plt.title("Projection of the membership functions on Feature 1")
plt.xlabel("Feature 1")
plt.ylabel("Degree of Membership")
plt.legend()
plt.grid(True)
plt.show()
```



In [227…

```python
# ---------------------------
# Gaussian Membership Function
# ---------------------------
class GaussianMF(nn.Module):
    def __init__(self, centers, sigmas, agg_prob):
        super().__init__()
        self.centers = nn.Parameter(torch.tensor(centers, dtype=torch.float32))
        self.sigmas = nn.Parameter(torch.tensor(sigmas, dtype=torch.float32))
        self.agg_prob=agg_prob

    def forward(self, x):
        # Expand for broadcasting
        # x: (batch, 1, n_dims), centers: (1, n_rules, n_dims), sigmas: (1, n_rules, n_dims)
        diff = abs((x.unsqueeze(1) - self.centers.unsqueeze(0))/self.sigmas.unsqueeze(0)) #(batch, n_rules, n_dims)

        # Aggregation
        if self.agg_prob:
            dist = torch.norm(diff, dim=-1)  # (batch, n_rules) # probablistic intersection
        else:
            dist = torch.max(diff, dim=-1).values  # (batch, n_rules) # min intersection (min instersection of normal funtion

        return torch.exp(-0.5 * dist ** 2)


# ---------------------------
# TSK Model
# ---------------------------
class TSK(nn.Module):
    def __init__(self, n_inputs, n_rules, centers, sigmas,agg_prob=False):
        super().__init__()
        self.n_inputs = n_inputs
        self.n_rules = n_rules

        # Antecedents (Gaussian MFs)

        self.mfs=GaussianMF(centers, sigmas,agg_prob)

        # Consequents (linear functions of inputs)
        # Each rule has coeffs for each input + bias
        self.consequents = nn.Parameter(
            torch.randn(n_inputs + 1,n_rules)
        )

    def forward(self, x):
        # x: (batch, n_inputs)
        batch_size = x.shape[0]

        # Compute membership values for each input feature
        # firing_strengths: (batch, n_rules)
        firing_strengths = self.mfs(x)

        # Normalize memberships
        # norm_fs: (batch, n_rules)
        norm_fs = firing_strengths / (firing_strengths.sum(dim=1, keepdim=True) + 1e-9)
```

```
        # Consequent output (linear model per rule)
        x_aug = torch.cat([x, torch.ones(batch_size, 1)], dim=1)  # add bias

        rule_outputs = torch.einsum("br,rk->bk", x_aug, self.consequents)  # (batch, rules)
        # Weighted sum
        output = torch.sum(norm_fs * rule_outputs, dim=1, keepdim=True)

        return output, norm_fs, rule_outputs
```

In [228…

```
# ---------------------------
# Least Squares Solver for Consequents (TSK)
# ---------------------------
def train_ls(model, X, y):
    with torch.no_grad():
        _, norm_fs, _ = model(X)

        # Design matrix for LS: combine normalized firing strengths with input
        X_aug = torch.cat([X, torch.ones(X.shape[0], 1)], dim=1)

        Phi = torch.einsum("br,bi->bri", X_aug, norm_fs).reshape(X.shape[0], -1)

        # Solve LS: consequents = (Phi^T Phi)^-1 Phi^T y

        theta= torch.linalg.lstsq(Phi, y).solution


        model.consequents.data = theta.reshape(model.consequents.shape)
```

In [229…

```
# ---------------------------
# Gradient Descent Training
# ---------------------------
def train_gd(model, X, y, epochs=100, lr=1e-3):
    optimizer = optim.Adam(model.parameters(), lr=lr)
    criterion = nn.MSELoss()
    for _ in range(epochs):
        optimizer.zero_grad()
        y_pred, _, _ = model(X)
        loss = criterion(y_pred, y)
        print(loss)
        loss.backward()
        optimizer.step()
```

In [230…

```
# ---------------------------
# Hybrid Training (Classic ANFIS)
# ---------------------------
def train_hybrid_alternating(model, X, y, max_iters=10, gd_epochs=20, lr=1e-3):
    for _ in range(max_iters):
        # Step A: GD on antecedents (freeze consequents)
        for p in model.consequents.parameters():
            p.requires_grad = False
        train_gd(model, X, y, epochs=gd_epochs, lr=lr)

        # Step B: LS on consequents (freeze antecedents)
        for p in model.consequents.parameters():
            p.requires_grad = True
        for p in model.mfs.parameters():
            p.requires_grad = False
        train_ls(model, X, y)

        # Re-enable antecedents
        for p in model.mfs.parameters():
            p.requires_grad = True
```

In [231…

```
# ---------------------------
# Alternative Hybrid Training (LS+ gradient descent on all)
# ---------------------------
def train_hybrid_classic(model, X, y, epochs=100, lr=1e-4):
    # Step 1: LS for consequents
    train_ls(model, X, y)
    # Step 2: GD fine-tuning
    train_gd(model, X, y, epochs=epochs, lr=lr)
```

In [232…

```
# Build model
model = TSK(n_inputs=Xtr.shape[1], n_rules=n_clusters, centers=centers[:,:-1], sigmas=sigmas[:,:-1])

Xtr = torch.tensor(Xtr, dtype=torch.float32)
ytr = torch.tensor(ytr, dtype=torch.float32)
Xte = torch.tensor(Xte, dtype=torch.float32)
yte = torch.tensor(yte, dtype=torch.float32)
```

In [233…

```
# Training with LS:
train_ls(model, Xtr, ytr.reshape(-1,1))
```

In [234…

```
y_pred, _, _=model(Xte)
print(f'ACC:{accuracy_score(yte.detach().numpy(),y_pred.detach().numpy()>0.5)}') #classification
#print(f'ACC:{mean_squared_error(yte.detach().numpy(),y_pred.detach().numpy())}') #regression
```

```
ACC:0.7922077922077922
```

The final model, configured with the optimal parameters, was trained and evaluated on the test set. It achieved a final classification accuracy of 0.7922, which corresponds to 79.22%.