Intelligent Systems - Assignment 2

Joel Janson (ist1116925)

Corresponding GitHub: https://github.com/joeljanson19/intelligent_systems.git

ANFIS for Dataset 1 (Regression)

In [313…

```
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 [313…

```
# CHOOSE DATASET

# 1. Regression
diabetes = datasets.load_diabetes(as_frame=True)
X = diabetes.data.values
y = diabetes.target.values

# 2. Classification
# diabetes = datasets.fetch_openml(name="diabetes", version=1, as_frame=True)
# X = diabetes.data.values
# y = diabetes.target.astype(str).map({'tested_positive': 1, 'tested_negative': 0}).values
```

In [313…

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

In [313…

```
# Standardize features
scaler=StandardScaler()
Xtr= scaler.fit_transform(Xtr)
Xte= scaler.transform(Xte)
```

Clustering Hyperparameters

- `n_clusters` – Number of fuzzy clusters. Controls model complexity.
- `m` – Fuzziness coefficient in fuzzy c-means. Higher `m` makes clusters fuzzier (wider membership finctions). `m=1` corresponds to hard clustering.

After tuning the model, the following values were chosen for the different datasets.

Regression: `n_clusters` =2, `m` =2

Classification: `n_clusters` =2, `m` =3

In [313…

```
 # Number of clusters
n_clusters = 2
m = 2

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

# 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 [314…

```
centers.shape
```

Out[3140]:

```
(2, 11)
```

In [314…

```
# 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)
```

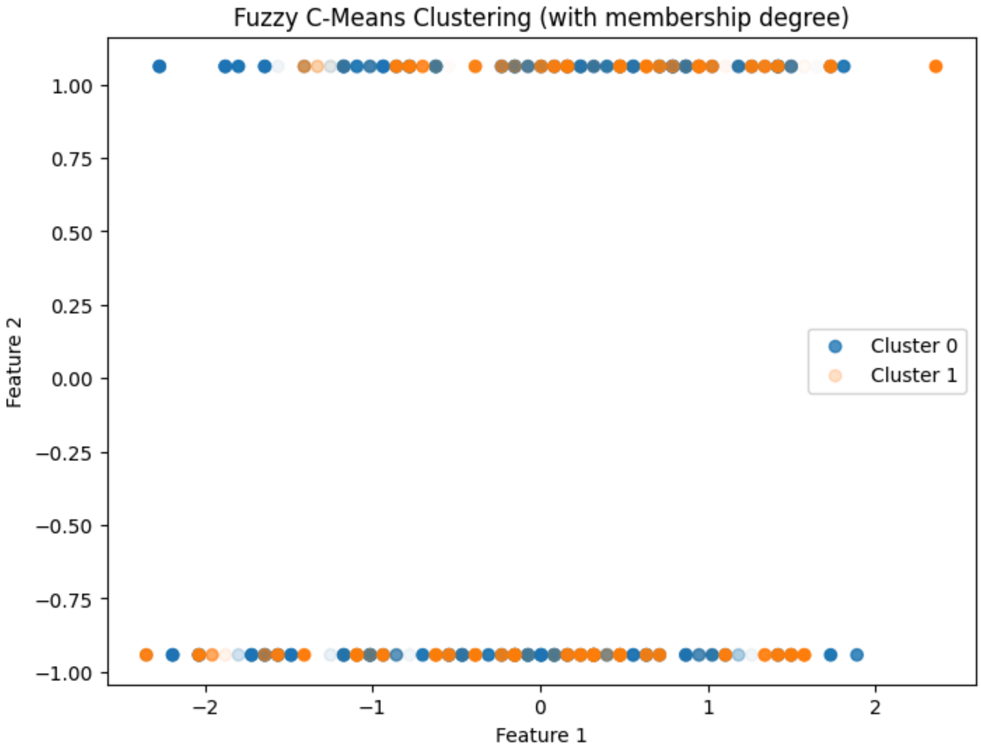
In [314…

```
# 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, 1],            # Feature 2
        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 2")
plt.legend()
plt.show()
```

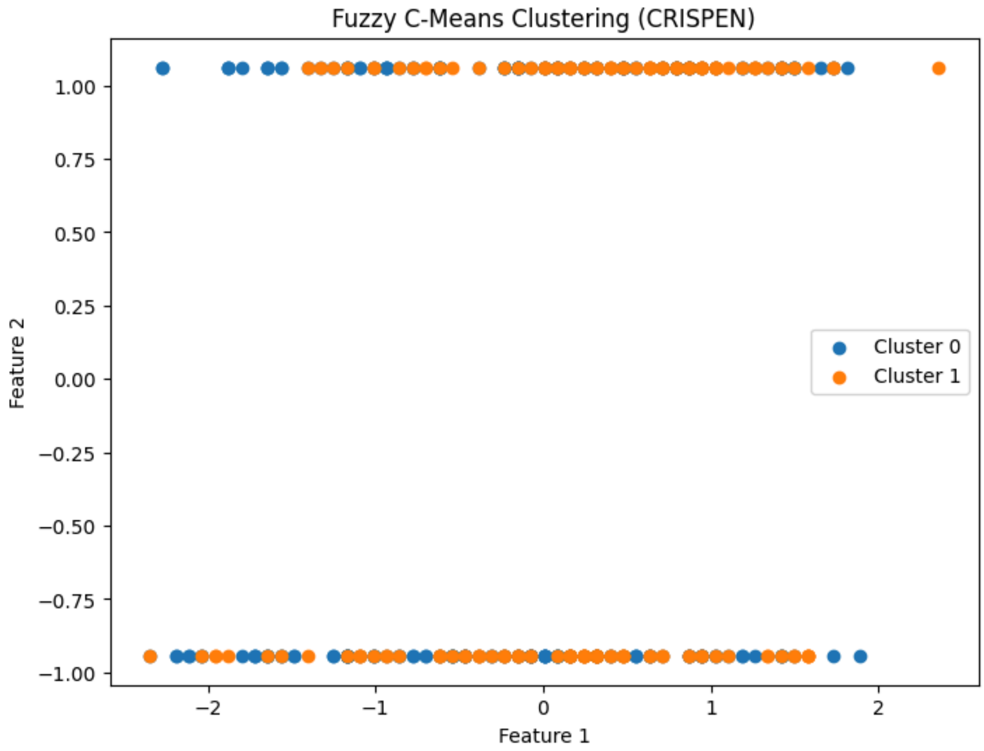
```
Fuzzy partition coefficient (FPC): 0.8556202662979464
```



In [314…

```
# 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, 1],
        label=f'Cluster {j}'
    )

plt.title("Fuzzy C-Means Clustering (CRISPEN)")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.show()
```



In [314…

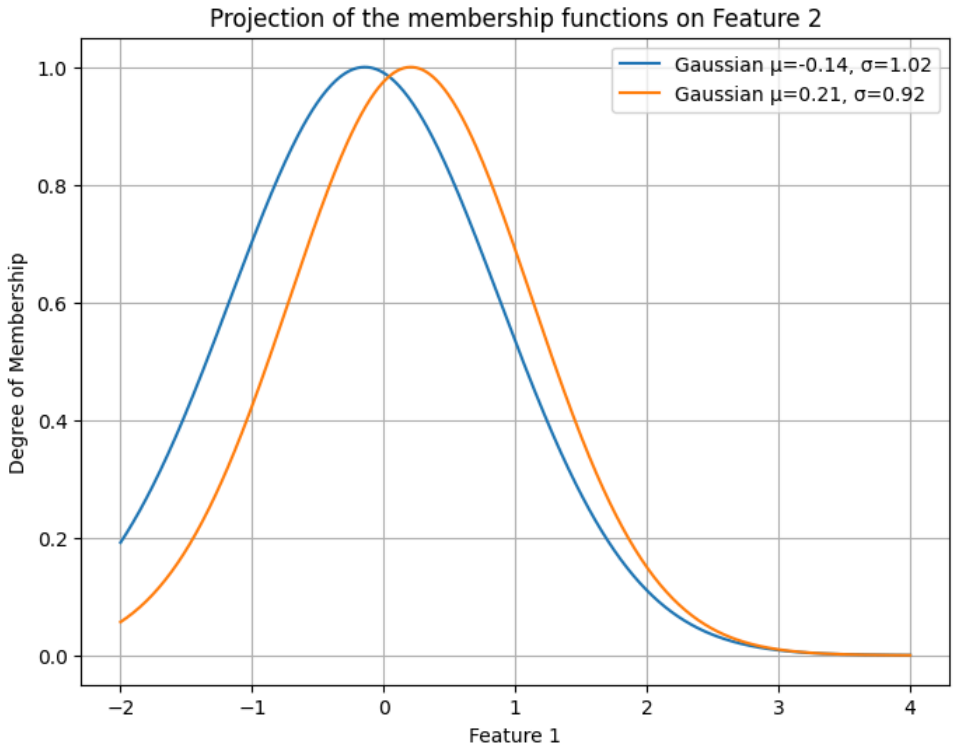
```
# Gaussian formula
def gaussian(x, mu, sigma):
    return np.exp(-0.5 * ((x - mu)/sigma)**2)

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

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

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

plt.title("Projection of the membership functions on Feature 2")
plt.xlabel("Feature 1")
plt.ylabel("Degree of Membership")
plt.legend()
plt.grid(True)
plt.show()
```



In [314…

```
# --------------------------
# 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)
```

```python
        # 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 [314…

```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 [314…

```python
# --------------------------
# 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 [314…

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

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

        # Re-enable antecedents
        model.mfs.requires_grad = True
```

In [314…

```python
# --------------------------
# Alternative Hybrid Training (LS+ gradient descent on all)
# --------------------------
def train_hybrid(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 [315…

```python
# 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)
```

Hyperparameters

- `max_iters` – Number of hybrid training cycles, alternating between gradient descent and least squares. (Default value = 10)
- `gd_epochs` – Number of epochs for the gradient descent step in each cycle. (Default value = 20)

- `lr` – Learning rate. Controls the size of parameter updates during gradient descent. (Default value = 0.001)
  - Higher `lr` → faster convergence but risk of instability or finding incorrect local minima.
  - Lower `lr` → more stable but slower training.

After tuning the model, the following values were chosen for the different datasets:
Regression: `max_iters` =7, `gd_epochs` =10, `lr` =0.001
Classification: `max_iters` =8, `gd_epochs` =15, `lr` =0.001

Important to note is that these values are not necessarily fully optimized. Training the model over and over with the same hyperparameters can give very different performance.

In [315…

```
# Training with LS:
train_hybrid_anfis(model, Xtr, ytr.reshape(-1,1), max_iters=7, gd_epochs=10, lr=0.001)
```

```
tensor(2683.3884, grad_fn=<MseLossBackward0>)
tensor(2682.0874, grad_fn=<MseLossBackward0>)
tensor(2680.8679, grad_fn=<MseLossBackward0>)
tensor(2679.6721, grad_fn=<MseLossBackward0>)
tensor(2678.5134, grad_fn=<MseLossBackward0>)
tensor(2677.3999, grad_fn=<MseLossBackward0>)
tensor(2676.3171, grad_fn=<MseLossBackward0>)
tensor(2675.2329, grad_fn=<MseLossBackward0>)
tensor(2674.1265, grad_fn=<MseLossBackward0>)
tensor(2673.0283, grad_fn=<MseLossBackward0>)
tensor(2671.2566, grad_fn=<MseLossBackward0>)
tensor(2670.0039, grad_fn=<MseLossBackward0>)
tensor(2668.7588, grad_fn=<MseLossBackward0>)
tensor(2667.5090, grad_fn=<MseLossBackward0>)
tensor(2666.2705, grad_fn=<MseLossBackward0>)
tensor(2665.0300, grad_fn=<MseLossBackward0>)
tensor(2663.7864, grad_fn=<MseLossBackward0>)
tensor(2662.5991, grad_fn=<MseLossBackward0>)
tensor(2661.4299, grad_fn=<MseLossBackward0>)
tensor(2660.2698, grad_fn=<MseLossBackward0>)
tensor(2658.4751, grad_fn=<MseLossBackward0>)
tensor(2657.2271, grad_fn=<MseLossBackward0>)
tensor(2655.9822, grad_fn=<MseLossBackward0>)
tensor(2654.7556, grad_fn=<MseLossBackward0>)
tensor(2653.5576, grad_fn=<MseLossBackward0>)
tensor(2652.3677, grad_fn=<MseLossBackward0>)
tensor(2651.1943, grad_fn=<MseLossBackward0>)
tensor(2650.0317, grad_fn=<MseLossBackward0>)
tensor(2648.8853, grad_fn=<MseLossBackward0>)
tensor(2647.7439, grad_fn=<MseLossBackward0>)
tensor(2645.9561, grad_fn=<MseLossBackward0>)
tensor(2644.6604, grad_fn=<MseLossBackward0>)
tensor(2643.3940, grad_fn=<MseLossBackward0>)
tensor(2642.1252, grad_fn=<MseLossBackward0>)
tensor(2640.8643, grad_fn=<MseLossBackward0>)
tensor(2639.6135, grad_fn=<MseLossBackward0>)
tensor(2638.3657, grad_fn=<MseLossBackward0>)
tensor(2637.1221, grad_fn=<MseLossBackward0>)
tensor(2635.8845, grad_fn=<MseLossBackward0>)
tensor(2634.6501, grad_fn=<MseLossBackward0>)
tensor(2632.5056, grad_fn=<MseLossBackward0>)
tensor(2631.1006, grad_fn=<MseLossBackward0>)
tensor(2629.7109, grad_fn=<MseLossBackward0>)
tensor(2628.2129, grad_fn=<MseLossBackward0>)
tensor(2626.6597, grad_fn=<MseLossBackward0>)
tensor(2625.1287, grad_fn=<MseLossBackward0>)
tensor(2623.6079, grad_fn=<MseLossBackward0>)
tensor(2622.1182, grad_fn=<MseLossBackward0>)
tensor(2620.6514, grad_fn=<MseLossBackward0>)
tensor(2619.1772, grad_fn=<MseLossBackward0>)
tensor(2616.5967, grad_fn=<MseLossBackward0>)
tensor(2614.9443, grad_fn=<MseLossBackward0>)
tensor(2613.2996, grad_fn=<MseLossBackward0>)
tensor(2611.6951, grad_fn=<MseLossBackward0>)
tensor(2610.1038, grad_fn=<MseLossBackward0>)
tensor(2608.5288, grad_fn=<MseLossBackward0>)
tensor(2606.9470, grad_fn=<MseLossBackward0>)
tensor(2605.4126, grad_fn=<MseLossBackward0>)
tensor(2603.8984, grad_fn=<MseLossBackward0>)
tensor(2602.4277, grad_fn=<MseLossBackward0>)
tensor(2599.8008, grad_fn=<MseLossBackward0>)
tensor(2598.1375, grad_fn=<MseLossBackward0>)
tensor(2596.4534, grad_fn=<MseLossBackward0>)
tensor(2594.7935, grad_fn=<MseLossBackward0>)
tensor(2593.1572, grad_fn=<MseLossBackward0>)
tensor(2591.5618, grad_fn=<MseLossBackward0>)
tensor(2589.9949, grad_fn=<MseLossBackward0>)
tensor(2588.4705, grad_fn=<MseLossBackward0>)
tensor(2586.9829, grad_fn=<MseLossBackward0>)
tensor(2585.5586, grad_fn=<MseLossBackward0>)
```

We print `mean_squared_error` and `accuracy_score` as indications of the performance of the model for regression and classification respectively.

In [315…

```
y_pred, _, _=model(Xte)
# Performance metric for regression
print(f'MSE:{mean_squared_error(yte.detach().numpy(),y_pred.detach().numpy())}')

# Performance metric for classification
# print(f'ACC:{accuracy_score(yte.detach().numpy(),y_pred.detach().numpy()>0.5)}')
```

```
MSE:2491.41162109375
```

ANFIS for Dataset 2 (Classification)

In [311…

```
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 [311…

```
# CHOOSE DATASET

# 1. Regression
# diabetes = datasets.load_diabetes(as_frame=True)
# X = diabetes.data.values
# y = diabetes.target.values

# 2. Classification
diabetes = datasets.fetch_openml(name="diabetes", version=1, as_frame=True)
X = diabetes.data.values
y = diabetes.target.astype(str).map({'tested_positive': 1, 'tested_negative': 0}).values
```

In [311…

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

In [311…

```
# Standardize features
scaler=StandardScaler()
Xtr= scaler.fit_transform(Xtr)
Xte= scaler.transform(Xte)
```

Clustering Hyperparameters

- `n_clusters` – Number of fuzzy clusters. Controls model complexity.
- `m` – Fuzziness coefficient in fuzzy c-means. Higher `m` makes clusters fuzzier (wider membership finctions). `m=1` corresponds to hard clustering.

After tuning the model, the following values were chosen for the different datasets.
Regression: `n_clusters` =2, `m` =2
Classification: `n_clusters` =2, `m` =3

In [312…

```
 # Number of clusters
n_clusters = 2
m = 3

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

# 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 [312…

```
centers.shape
```

Out[3121]: (2, 9)

In [312…

```
# 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)
```

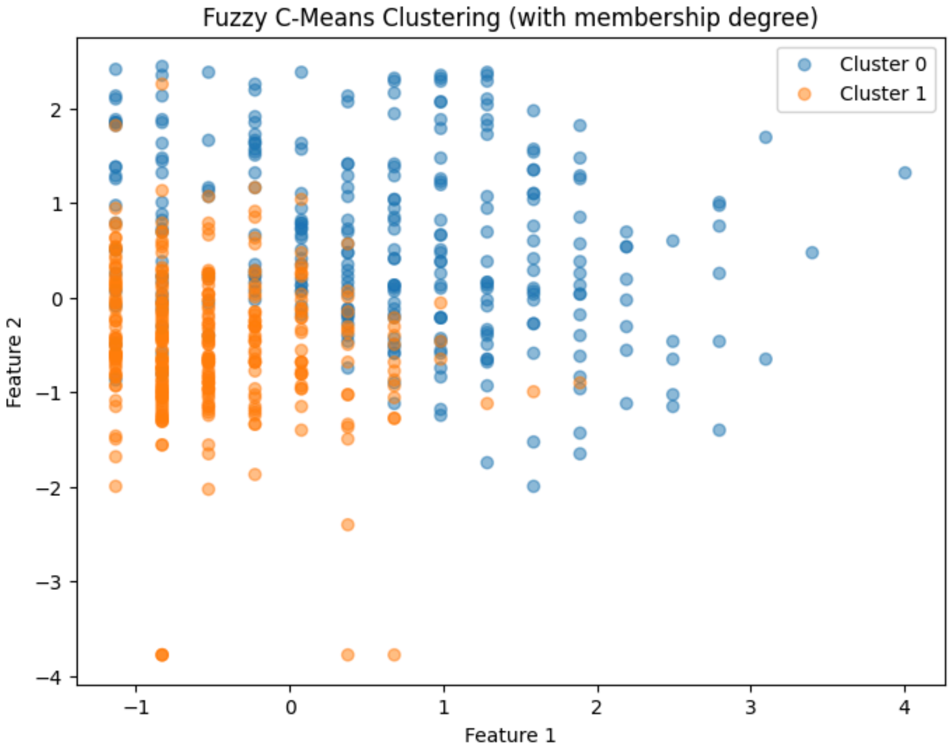
In [312…

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

```python
# 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, 1],             # Feature 2
        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 2")
plt.legend()
plt.show()
```

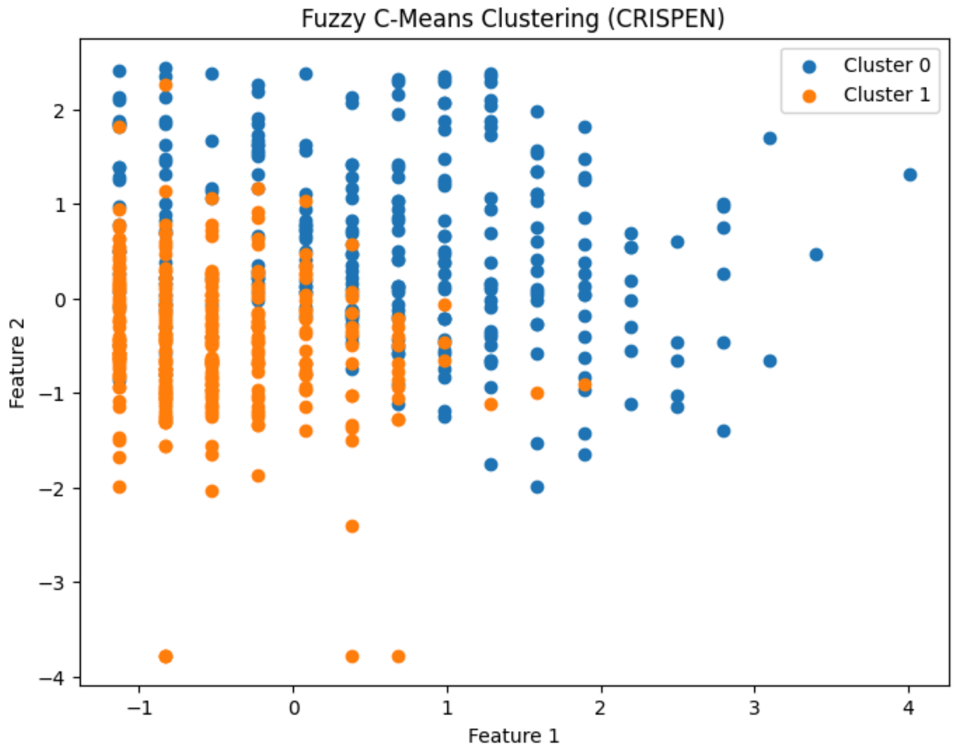
```
Fuzzy partition coefficient (FPC): 0.5000002846707329
```



In [312…

```python
# 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, 1],
        label=f'Cluster {j}'
    )

plt.title("Fuzzy C-Means Clustering (CRISPEN)")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.show()
```



In [312…

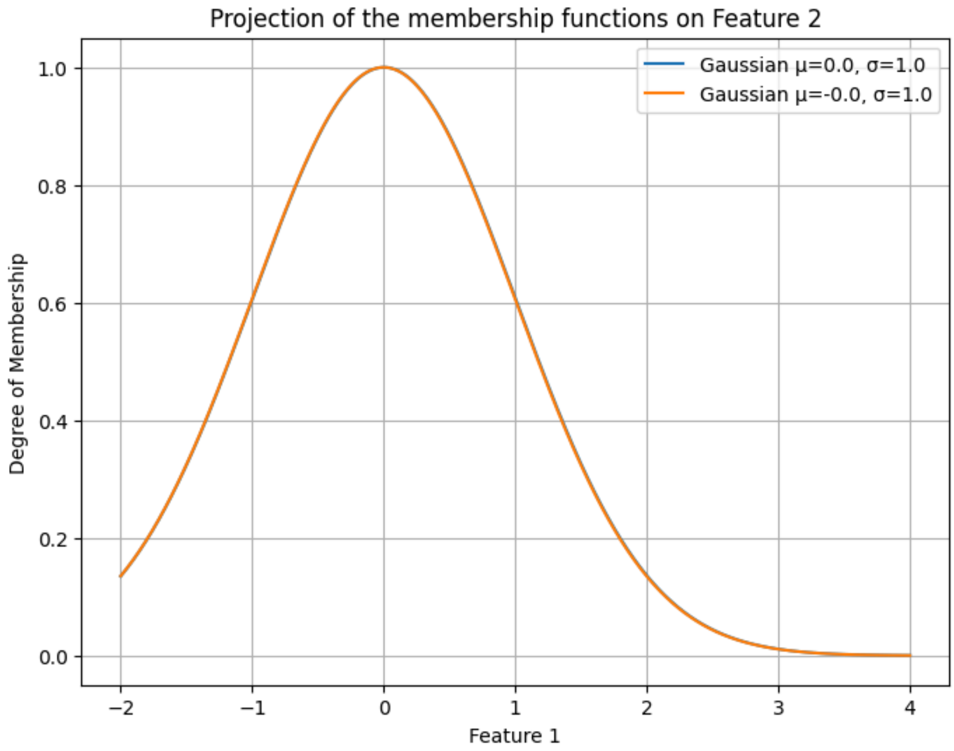
```
# Gaussian formula
def gaussian(x, mu, sigma):
    return np.exp(-0.5 * ((x - mu)/sigma)**2)

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

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

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

plt.title("Projection of the membership functions on Feature 2")
plt.xlabel("Feature 1")
plt.ylabel("Degree of Membership")
plt.legend()
plt.grid(True)
plt.show()
```



In [312…

```
# --------------------------
# 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 [312…
```
# --------------------------
# 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 [312…
```
# --------------------------
# 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 [312…
```
# --------------------------
# Hybrid Training (Classic ANFIS)
# --------------------------
def train_hybrid_anfis(model, X, y, max_iters=10, gd_epochs=20, lr=1e-3):
    train_ls(model, X, y)
    for _ in range(max_iters):
        # Step A: GD on antecedents (freeze consequents)
        model.consequents.requires_grad = False
        train_gd(model, X, y, epochs=gd_epochs, lr=lr)

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

        # Re-enable antecedents
        model.mfs.requires_grad = True
```

In [313…
```
# --------------------------
# Alternative Hybrid Training (LS+ gradient descent on all)
# --------------------------
def train_hybrid(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 [313…
```
# 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)
```

Hyperparameters

- `max_iters` – Number of hybrid training cycles, alternating between gradient descent and least squares. (Default value = 10)
- `gd_epochs` – Number of epochs for the gradient descent step in each cycle. (Default value = 20)

- `lr` – Learning rate. Controls the size of parameter updates during gradient descent. (Default value = 0.001)
  - Higher `lr` → faster convergence but risk of instability or finding incorrect local minima.
  - Lower `lr` → more stable but slower training.

After tuning the model, the following values were chosen for the different datasets:
Regression: `max_iters` =7, `gd_epochs` =10, `lr` =0.001
Classification: `max_iters` =8, `gd_epochs` =15, `lr` =0.001

Important to note is that these values are not necessarily fully optimized. Training the model over and over with the same hyperparameters can give very different performance.

In [313…

```
# Training with LS:
train_hybrid_anfis(model, Xtr, ytr.reshape(-1,1), max_iters=8, gd_epochs=15, lr=0.001)
```

```
tensor(0.1524, grad_fn=<MseLossBackward0>)
tensor(0.1808, grad_fn=<MseLossBackward0>)
tensor(0.1535, grad_fn=<MseLossBackward0>)
tensor(0.1577, grad_fn=<MseLossBackward0>)
tensor(0.1660, grad_fn=<MseLossBackward0>)
tensor(0.1614, grad_fn=<MseLossBackward0>)
tensor(0.1530, grad_fn=<MseLossBackward0>)
tensor(0.1506, grad_fn=<MseLossBackward0>)
tensor(0.1550, grad_fn=<MseLossBackward0>)
tensor(0.1585, grad_fn=<MseLossBackward0>)
tensor(0.1564, grad_fn=<MseLossBackward0>)
tensor(0.1517, grad_fn=<MseLossBackward0>)
tensor(0.1497, grad_fn=<MseLossBackward0>)
tensor(0.1516, grad_fn=<MseLossBackward0>)
tensor(0.1542, grad_fn=<MseLossBackward0>)
tensor(0.1497, grad_fn=<MseLossBackward0>)
tensor(0.1751, grad_fn=<MseLossBackward0>)
tensor(0.1507, grad_fn=<MseLossBackward0>)
tensor(0.1539, grad_fn=<MseLossBackward0>)
tensor(0.1628, grad_fn=<MseLossBackward0>)
tensor(0.1578, grad_fn=<MseLossBackward0>)
tensor(0.1499, grad_fn=<MseLossBackward0>)
tensor(0.1490, grad_fn=<MseLossBackward0>)
tensor(0.1535, grad_fn=<MseLossBackward0>)
tensor(0.1556, grad_fn=<MseLossBackward0>)
tensor(0.1529, grad_fn=<MseLossBackward0>)
tensor(0.1493, grad_fn=<MseLossBackward0>)
tensor(0.1485, grad_fn=<MseLossBackward0>)
tensor(0.1504, grad_fn=<MseLossBackward0>)
tensor(0.1519, grad_fn=<MseLossBackward0>)
tensor(0.1503, grad_fn=<MseLossBackward0>)
tensor(0.1602, grad_fn=<MseLossBackward0>)
tensor(0.1500, grad_fn=<MseLossBackward0>)
tensor(0.1495, grad_fn=<MseLossBackward0>)
tensor(0.1539, grad_fn=<MseLossBackward0>)
tensor(0.1520, grad_fn=<MseLossBackward0>)
tensor(0.1484, grad_fn=<MseLossBackward0>)
tensor(0.1484, grad_fn=<MseLossBackward0>)
tensor(0.1505, grad_fn=<MseLossBackward0>)
tensor(0.1508, grad_fn=<MseLossBackward0>)
tensor(0.1491, grad_fn=<MseLossBackward0>)
tensor(0.1479, grad_fn=<MseLossBackward0>)
tensor(0.1484, grad_fn=<MseLossBackward0>)
tensor(0.1493, grad_fn=<MseLossBackward0>)
tensor(0.1491, grad_fn=<MseLossBackward0>)
tensor(0.1473, grad_fn=<MseLossBackward0>)
tensor(0.1680, grad_fn=<MseLossBackward0>)
tensor(0.1473, grad_fn=<MseLossBackward0>)
tensor(0.1500, grad_fn=<MseLossBackward0>)
tensor(0.1576, grad_fn=<MseLossBackward0>)
tensor(0.1533, grad_fn=<MseLossBackward0>)
tensor(0.1466, grad_fn=<MseLossBackward0>)
tensor(0.1457, grad_fn=<MseLossBackward0>)
tensor(0.1494, grad_fn=<MseLossBackward0>)
tensor(0.1514, grad_fn=<MseLossBackward0>)
tensor(0.1496, grad_fn=<MseLossBackward0>)
tensor(0.1464, grad_fn=<MseLossBackward0>)
tensor(0.1452, grad_fn=<MseLossBackward0>)
tensor(0.1466, grad_fn=<MseLossBackward0>)
tensor(0.1482, grad_fn=<MseLossBackward0>)
tensor(0.1458, grad_fn=<MseLossBackward0>)
tensor(0.1565, grad_fn=<MseLossBackward0>)
tensor(0.1459, grad_fn=<MseLossBackward0>)
tensor(0.1463, grad_fn=<MseLossBackward0>)
tensor(0.1502, grad_fn=<MseLossBackward0>)
tensor(0.1484, grad_fn=<MseLossBackward0>)
tensor(0.1449, grad_fn=<MseLossBackward0>)
tensor(0.1440, grad_fn=<MseLossBackward0>)
tensor(0.1456, grad_fn=<MseLossBackward0>)
tensor(0.1466, grad_fn=<MseLossBackward0>)
tensor(0.1456, grad_fn=<MseLossBackward0>)
tensor(0.1440, grad_fn=<MseLossBackward0>)
tensor(0.1435, grad_fn=<MseLossBackward0>)
tensor(0.1441, grad_fn=<MseLossBackward0>)
tensor(0.1445, grad_fn=<MseLossBackward0>)
tensor(0.1425, grad_fn=<MseLossBackward0>)
tensor(0.1569, grad_fn=<MseLossBackward0>)
tensor(0.1428, grad_fn=<MseLossBackward0>)
tensor(0.1443, grad_fn=<MseLossBackward0>)
tensor(0.1494, grad_fn=<MseLossBackward0>)
tensor(0.1467, grad_fn=<MseLossBackward0>)
tensor(0.1421, grad_fn=<MseLossBackward0>)
tensor(0.1410, grad_fn=<MseLossBackward0>)
tensor(0.1434, grad_fn=<MseLossBackward0>)
tensor(0.1447, grad_fn=<MseLossBackward0>)
tensor(0.1435, grad_fn=<MseLossBackward0>)
tensor(0.1414, grad_fn=<MseLossBackward0>)
tensor(0.1406, grad_fn=<MseLossBackward0>)
```

```
tensor(0.1413, grad_fn=<MseLossBackward0>)
tensor(0.1421, grad_fn=<MseLossBackward0>)
tensor(0.1411, grad_fn=<MseLossBackward0>)
tensor(0.1559, grad_fn=<MseLossBackward0>)
tensor(0.1414, grad_fn=<MseLossBackward0>)
tensor(0.1431, grad_fn=<MseLossBackward0>)
tensor(0.1483, grad_fn=<MseLossBackward0>)
tensor(0.1450, grad_fn=<MseLossBackward0>)
tensor(0.1404, grad_fn=<MseLossBackward0>)
tensor(0.1406, grad_fn=<MseLossBackward0>)
tensor(0.1435, grad_fn=<MseLossBackward0>)
tensor(0.1440, grad_fn=<MseLossBackward0>)
tensor(0.1418, grad_fn=<MseLossBackward0>)
tensor(0.1400, grad_fn=<MseLossBackward0>)
tensor(0.1403, grad_fn=<MseLossBackward0>)
tensor(0.1417, grad_fn=<MseLossBackward0>)
tensor(0.1421, grad_fn=<MseLossBackward0>)
tensor(0.1407, grad_fn=<MseLossBackward0>)
tensor(0.1536, grad_fn=<MseLossBackward0>)
tensor(0.1412, grad_fn=<MseLossBackward0>)
tensor(0.1424, grad_fn=<MseLossBackward0>)
tensor(0.1468, grad_fn=<MseLossBackward0>)
tensor(0.1441, grad_fn=<MseLossBackward0>)
tensor(0.1402, grad_fn=<MseLossBackward0>)
tensor(0.1403, grad_fn=<MseLossBackward0>)
tensor(0.1428, grad_fn=<MseLossBackward0>)
tensor(0.1433, grad_fn=<MseLossBackward0>)
tensor(0.1414, grad_fn=<MseLossBackward0>)
tensor(0.1397, grad_fn=<MseLossBackward0>)
tensor(0.1401, grad_fn=<MseLossBackward0>)
tensor(0.1414, grad_fn=<MseLossBackward0>)
tensor(0.1417, grad_fn=<MseLossBackward0>)
```

We print `mean_squared_error` and `accuracy_score` as indications of the performance of the model for regression and classification respectively.

In [313…

```
y_pred, _, _=model(Xte)
# Performance metric for regression
# print(f'MSE:{mean_squared_error(yte.detach().numpy(),y_pred.detach().numpy())}')

# Performance metric for classification
print(f'ACC:{accuracy_score(yte.detach().numpy(),y_pred.detach().numpy()>0.5)}')
```

```
ACC:0.7857142857142857
```

Neural Network for Dataset 1 (Regression)

In [200…

```
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 matplotlib.pyplot as plt
import torch.nn.functional as F
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
import pandas
```

In [200…

```
# CHOOSE DATASET

# 1. Regression
diabetes = datasets.load_diabetes(as_frame=True)
X = diabetes.data.values
y = diabetes.target.values

# 2. Classification
# diabetes = datasets.fetch_openml(name="diabetes", version=1, as_frame=True)
# X = diabetes.data.values
# y = diabetes.target.astype(str).map({'tested_positive': 1, 'tested_negative': 0}).values
```

In [200…

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

In [200…

```
# Standardize features
scaler=StandardScaler()
Xtr= scaler.fit_transform(Xtr)
Xte= scaler.transform(Xte)
```

NN Architecture

The architecture can be tuned by changing the number of layers, layer size and regularization (dropout). Dropout prevents overfitting by randomly "dropping out" (setting to zero) a fraction of the neurons during training, which forces the network to learn more robust and generalized features. Regularization is determined later together with other hyperparameters.

It can be very hard to determine an optimal architecture but I ended up using the following:
Regression: 3 hidden layers with 64 neurons each.
Classification: 4 hidden layers with 64 neurons in the first three and 32 neurons in the last.

In [200…

```
class MLP(nn.Module):
    def __init__(self, input_size, output_size=1, dropout_prob=0.5):
        super(MLP, self).__init__()

        self.fc1 = nn.Linear(input_size, 64)
        # self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, 64)
        self.fc4 = nn.Linear(64, 64)
        self.out = nn.Linear(64, output_size)

        self.dropout = nn.Dropout(p=dropout_prob)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.dropout(x)

        # x = F.relu(self.fc2(x))
        # x = self.dropout(x)

        x = F.relu(self.fc3(x))
        x = self.dropout(x)

        x = F.relu(self.fc4(x))
        x = self.dropout(x)

        x = self.out(x)
        return x
```

Hyperparameters

- `num_epochs` – Number of training passes over the entire dataset. Don't want too many epochs to avoid overfitting to noise.
- `lr` – Learning rate. Step size for updating weights during training. Controls how fast the model learns.

- `dropout` – Fraction of neurons randomly dropped during training to reduce overfitting. In this task I'm using 10% but for toy datasets, higher fraction could be used.
- `batch_size` – Number of samples processed before updating the model. If too much RAM is being used, this value could for example be dropped to 32.

After tuning the model, the following values were chosen for the different datasets:
Regression: `num_epochs` =60, `lr` =0.001, `dropout` =0.1, `batch_size` =64
Classification: `num_epochs` =75, `lr` =0.001, `dropout` =0.1, `batch_size` =64

Important to note is that these values are not necessarily fully optimized. Training the model over and over with the same hyperparameters can give very different performance.

In [200…
```
num_epochs=60
lr=0.001
dropout=0.1
batch_size=64
```

In [200…
```
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)

# Wrap Xtr and ytr into a dataset
train_dataset = TensorDataset(Xtr, ytr)

# Create DataLoader
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
```

In [200…
```
# Model, Loss, Optimizer
# device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# Ignoring this line since I'm not using cuda

model = MLP(input_size=Xtr.shape[1], dropout_prob=dropout)#.to(device)
# criterion = nn.BCEWithLogitsLoss()  # for binary classification
criterion = nn.MSELoss() # for regression
optimizer = optim.Adam(model.parameters(), lr=lr) #can use different optimizer such as AdamW but not necessary
```

In [201…
```
# Training loop
for epoch in range(num_epochs):
    model.train() #train or evolve
    epoch_loss = 0.0

    for batch_x, batch_y in train_dataloader:
        batch_x = batch_x#.to(device)
        batch_y = batch_y#.to(device)

        logits = model(batch_x)
        loss = criterion(logits, batch_y.view(-1, 1))

        optimizer.zero_grad()
        loss.backward() #directly related to the forward function defined above
        optimizer.step()

        epoch_loss += loss.item()

    avg_loss = epoch_loss / len(train_dataloader)
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {avg_loss:.4f}")
```

```
Epoch [1/60], Loss: 29575.9437
Epoch [2/60], Loss: 30131.5693
Epoch [3/60], Loss: 29604.6950
Epoch [4/60], Loss: 29306.4294
Epoch [5/60], Loss: 28574.7874
Epoch [6/60], Loss: 28728.4570
Epoch [7/60], Loss: 27837.8532
Epoch [8/60], Loss: 27302.4528
Epoch [9/60], Loss: 26152.4359
Epoch [10/60], Loss: 23954.2630
Epoch [11/60], Loss: 21583.5163
Epoch [12/60], Loss: 18228.2782
Epoch [13/60], Loss: 14263.4688
Epoch [14/60], Loss: 10740.4442
Epoch [15/60], Loss: 8089.3803
Epoch [16/60], Loss: 6009.1539
Epoch [17/60], Loss: 5237.9879
Epoch [18/60], Loss: 5130.6556
Epoch [19/60], Loss: 4511.1023
Epoch [20/60], Loss: 4366.9094
Epoch [21/60], Loss: 3985.4761
Epoch [22/60], Loss: 4150.6009
Epoch [23/60], Loss: 3794.2279
Epoch [24/60], Loss: 4077.5017
Epoch [25/60], Loss: 3815.5411
Epoch [26/60], Loss: 3694.2213
Epoch [27/60], Loss: 3857.4960
Epoch [28/60], Loss: 3602.5615
Epoch [29/60], Loss: 3520.0992
Epoch [30/60], Loss: 3633.3056
Epoch [31/60], Loss: 3381.4645
Epoch [32/60], Loss: 3406.5404
Epoch [33/60], Loss: 3482.0994
Epoch [34/60], Loss: 3377.6129
Epoch [35/60], Loss: 3291.2661
Epoch [36/60], Loss: 3390.1335
Epoch [37/60], Loss: 3315.6458
Epoch [38/60], Loss: 3360.6391
Epoch [39/60], Loss: 3155.7889
Epoch [40/60], Loss: 3310.3440
Epoch [41/60], Loss: 3240.7974
Epoch [42/60], Loss: 3283.4766
Epoch [43/60], Loss: 3320.3667
Epoch [44/60], Loss: 3248.5180
Epoch [45/60], Loss: 3157.4210
Epoch [46/60], Loss: 3276.6381
Epoch [47/60], Loss: 3230.1724
Epoch [48/60], Loss: 3087.7275
Epoch [49/60], Loss: 2932.4183
Epoch [50/60], Loss: 3325.6077
Epoch [51/60], Loss: 3025.9305
Epoch [52/60], Loss: 3018.6002
Epoch [53/60], Loss: 3288.2196
Epoch [54/60], Loss: 3188.2162
Epoch [55/60], Loss: 3165.9945
Epoch [56/60], Loss: 3100.1974
Epoch [57/60], Loss: 3166.8178
Epoch [58/60], Loss: 3157.9259
Epoch [59/60], Loss: 3119.9365
Epoch [60/60], Loss: 3030.1863
```

We print `mean_squared_error` and `accuracy_score` as indications of the performance of the model for regression and classification respectively.

In [201…

```
y_pred=model(Xte)
# Performance metric for regression
print(f'MSE:{mean_squared_error(yte.detach().numpy(),y_pred.detach().numpy())}')

# Performance metric for classification
# print(f'ACC:{accuracy_score(yte.detach().numpy(),y_pred.detach().numpy()>0.5)}')
```

```
MSE:2840.69287109375
```

Neural Network for Dataset 2 (Classification)

In [234…

```
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 matplotlib.pyplot as plt
import torch.nn.functional as F
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
import pandas
```

In [234…

```
# CHOOSE DATASET

# 1. Regression
# diabetes = datasets.load_diabetes(as_frame=True)
# X = diabetes.data.values
# y = diabetes.target.values

# 2. Classification
diabetes = datasets.fetch_openml(name="diabetes", version=1, as_frame=True)
X = diabetes.data.values
y = diabetes.target.astype(str).map({'tested_positive': 1, 'tested_negative': 0}).values
```

In [234…

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

In [234…

```
# Standardize features
scaler=StandardScaler()
Xtr= scaler.fit_transform(Xtr)
Xte= scaler.transform(Xte)
```

NN Architecture

The architecture can be tuned by changing the number of layers, layer size and regularization (dropout). Dropout prevents overfitting by randomly "dropping out" (setting to zero) a fraction of the neurons during training, which forces the network to learn more robust and generalized features. Regularization is determined later together with other hyperparameters.

It can be very hard to determine an optimal architecture but I ended up using the following:
Regression: 3 hidden layers with 64 neurons each.
Classification: 4 hidden layers with 64 neurons in the first three and 32 neurons in the last.

In [234…

```
class MLP(nn.Module):
    def __init__(self, input_size, output_size=1, dropout_prob=0.5):
        super(MLP, self).__init__()

        self.fc1 = nn.Linear(input_size, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, 64)
        self.fc4 = nn.Linear(64, 32)
        self.out = nn.Linear(32, output_size)

        self.dropout = nn.Dropout(p=dropout_prob)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.dropout(x)

        x = F.relu(self.fc2(x))
        x = self.dropout(x)

        x = F.relu(self.fc3(x))
        x = self.dropout(x)

        x = F.relu(self.fc4(x))
        x = self.dropout(x)

        x = self.out(x)
        return x
```

Hyperparameters

- `num_epochs` – Number of training passes over the entire dataset. Don't want too many epochs to avoid overfitting to noise.
- `lr` – Learning rate. Step size for updating weights during training. Controls how fast the model learns.

- `dropout` – Fraction of neurons randomly dropped during training to reduce overfitting. In this task I'm using 10% but for toy datasets, higher fraction could be used.
- `batch_size` – Number of samples processed before updating the model. If too much RAM is being used, this value could for example be dropped to 32.

After tuning the model, the following values were chosen for the different datasets:
Regression: `num_epochs` =60, `lr` =0.001, `dropout` =0.1, `batch_size` =64
Classification: `num_epochs` =75, `lr` =0.001, `dropout` =0.1, `batch_size` =64

Important to note is that these values are not necessarily fully optimized. Training the model over and over with the same hyperparameters can give very different performance.

In [234...

```
num_epochs=75
lr=0.001
dropout=0.1
batch_size=64
```

In [234...

```
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)

# Wrap Xtr and ytr into a dataset
train_dataset = TensorDataset(Xtr, ytr)

# Create DataLoader
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
```

In [234...

```
# Model, Loss, Optimizer
# device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# Ignoring this line since I'm not using cuda

model = MLP(input_size=Xtr.shape[1], dropout_prob=dropout)#.to(device)
criterion = nn.BCEWithLogitsLoss()  # for binary classification
# criterion = nn.MSELoss() # for regression
optimizer = optim.Adam(model.parameters(), lr=lr) #can use different optimizer such as AdamW but not necessary
```

In [235...

```
# Training loop
for epoch in range(num_epochs):
    model.train() #train or evolve
    epoch_loss = 0.0

    for batch_x, batch_y in train_dataloader:
        batch_x = batch_x#.to(device)
        batch_y = batch_y#.to(device)

        logits = model(batch_x)
        loss = criterion(logits, batch_y.view(-1, 1))

        optimizer.zero_grad()
        loss.backward() #directly related to the forward function defined above
        optimizer.step()

        epoch_loss += loss.item()

    avg_loss = epoch_loss / len(train_dataloader)
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {avg_loss:.4f}")
```

```
Epoch [1/75], Loss: 0.6835
Epoch [2/75], Loss: 0.6692
Epoch [3/75], Loss: 0.6394
Epoch [4/75], Loss: 0.5882
Epoch [5/75], Loss: 0.5221
Epoch [6/75], Loss: 0.4957
Epoch [7/75], Loss: 0.4754
Epoch [8/75], Loss: 0.4677
Epoch [9/75], Loss: 0.4749
Epoch [10/75], Loss: 0.4667
Epoch [11/75], Loss: 0.4353
Epoch [12/75], Loss: 0.4642
Epoch [13/75], Loss: 0.4428
Epoch [14/75], Loss: 0.4389
Epoch [15/75], Loss: 0.4491
Epoch [16/75], Loss: 0.4447
Epoch [17/75], Loss: 0.4344
Epoch [18/75], Loss: 0.4383
Epoch [19/75], Loss: 0.4351
Epoch [20/75], Loss: 0.4278
Epoch [21/75], Loss: 0.4274
Epoch [22/75], Loss: 0.4197
Epoch [23/75], Loss: 0.4188
Epoch [24/75], Loss: 0.4240
Epoch [25/75], Loss: 0.4137
Epoch [26/75], Loss: 0.4202
Epoch [27/75], Loss: 0.4215
Epoch [28/75], Loss: 0.4141
Epoch [29/75], Loss: 0.4112
Epoch [30/75], Loss: 0.4078
Epoch [31/75], Loss: 0.3989
Epoch [32/75], Loss: 0.4067
Epoch [33/75], Loss: 0.4010
Epoch [34/75], Loss: 0.3892
Epoch [35/75], Loss: 0.4026
Epoch [36/75], Loss: 0.3921
Epoch [37/75], Loss: 0.3977
Epoch [38/75], Loss: 0.4073
Epoch [39/75], Loss: 0.3947
Epoch [40/75], Loss: 0.3839
Epoch [41/75], Loss: 0.3896
Epoch [42/75], Loss: 0.3842
Epoch [43/75], Loss: 0.3847
Epoch [44/75], Loss: 0.3763
Epoch [45/75], Loss: 0.3743
Epoch [46/75], Loss: 0.3690
Epoch [47/75], Loss: 0.3823
Epoch [48/75], Loss: 0.3699
Epoch [49/75], Loss: 0.3629
Epoch [50/75], Loss: 0.3632
Epoch [51/75], Loss: 0.3817
Epoch [52/75], Loss: 0.3681
Epoch [53/75], Loss: 0.3643
Epoch [54/75], Loss: 0.3665
Epoch [55/75], Loss: 0.3735
Epoch [56/75], Loss: 0.3527
Epoch [57/75], Loss: 0.3486
Epoch [58/75], Loss: 0.3389
Epoch [59/75], Loss: 0.3513
Epoch [60/75], Loss: 0.3435
Epoch [61/75], Loss: 0.3503
Epoch [62/75], Loss: 0.3508
Epoch [63/75], Loss: 0.3630
Epoch [64/75], Loss: 0.3472
Epoch [65/75], Loss: 0.3382
Epoch [66/75], Loss: 0.3241
Epoch [67/75], Loss: 0.3291
Epoch [68/75], Loss: 0.3372
Epoch [69/75], Loss: 0.3528
Epoch [70/75], Loss: 0.3269
Epoch [71/75], Loss: 0.3217
Epoch [72/75], Loss: 0.3323
Epoch [73/75], Loss: 0.3386
Epoch [74/75], Loss: 0.3416
Epoch [75/75], Loss: 0.3270
```

We print `mean_squared_error` and `accuracy_score` as indications of the performance of the model for regression and classification respectively.

In [235…

```
y_pred=model(Xte)
# Performance metric for regression
# print(f'MSE:{mean_squared_error(yte.detach().numpy(),y_pred.detach().numpy())}')

# Performance metric for classification
print(f'ACC:{accuracy_score(yte.detach().numpy(),y_pred.detach().numpy()>0.5)}')
```

```
ACC:0.7792207792207793
```

Discussion

For `Dataset 1 (regression)`, the following MSE scores were obtained using the different models:

- **LS**: 2545.29
- **ANFIS**: 2491.41
- **NN**: 2818.01

We can see that the Hybrid ANFIS model achieved the lowest MSE, slightly outperforming the least squares model from assignment 1. This is because ANFIS combines least squares with gradient descent to adjust both the antecedent (fuzzy membership) and consequent parameters. The Neural Network model performed worst, likely due to insufficient tuning, leading to underfitting or poor generalization. For this dataset, a simpler model with fuzzy structure may be more effective.

For `Dataset 2 (classification)`, the following accuracy scores were obtained using the different models:

- **LS**: 0.7532
- **ANFIS**: 0.7857
- **NN**: 0.7792

All models performed reasonably well, with ANFIS achieving the highest accuracy, suggesting that the fuzzy clustering and rule-based structure captured class boundaries effectively. The neural network performed slightly worse than ANFIS but better than LS. In order to outperform ANFIS, it would probably require more careful tuning of architecture and hyperparameters.