Intelligent Systems

Assignment 2

Bernardo Morais Chagas (103639)

September 30, 2025

Repository link: **GitHub Repository**

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
```

Importation of Dataset 1

In [2]:
```
# CHOOSE DATASET

# Binary classification dataset
diabetes = datasets.load_diabetes(as_frame=True)

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

X = diabetes.data.values
y = diabetes.target.values

print("Shape:", X.shape)

print(diabetes.data.head(), "\n \n")# first rows of features
print(diabetes.target.head()) # first rows of target
```

```
Shape: (442, 10)
        age       sex       bmi        bp        s1        s2        s3  \
0  0.038076  0.050680  0.061696  0.021872 -0.044223 -0.034821 -0.043401
1 -0.001882 -0.044642 -0.051474 -0.026328 -0.008449 -0.019163  0.074412
2  0.085299  0.050680  0.044451 -0.005670 -0.045599 -0.034194 -0.032356
3 -0.089063 -0.044642 -0.011595 -0.036656  0.012191  0.024991 -0.036038
4  0.005383 -0.044642 -0.036385  0.021872  0.003935  0.015596  0.008142

         s4        s5        s6
0 -0.002592  0.019907 -0.017646
1 -0.039493 -0.068332 -0.092204
2 -0.002592  0.002861 -0.025930
3  0.034309  0.022688 -0.009362
4 -0.002592 -0.031988 -0.046641


0    151.0
1     75.0
2    141.0
3    206.0
4    135.0
Name: target, dtype: float64
```

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)
```

In order to be able to compare the results, the number of clusters and value of m used, was the same as the one in the previous assigment (n_clusters = 4; m=1.1)

In [5]:
```
 # Number of clusters
n_clusters = 4
m=1.1
```

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

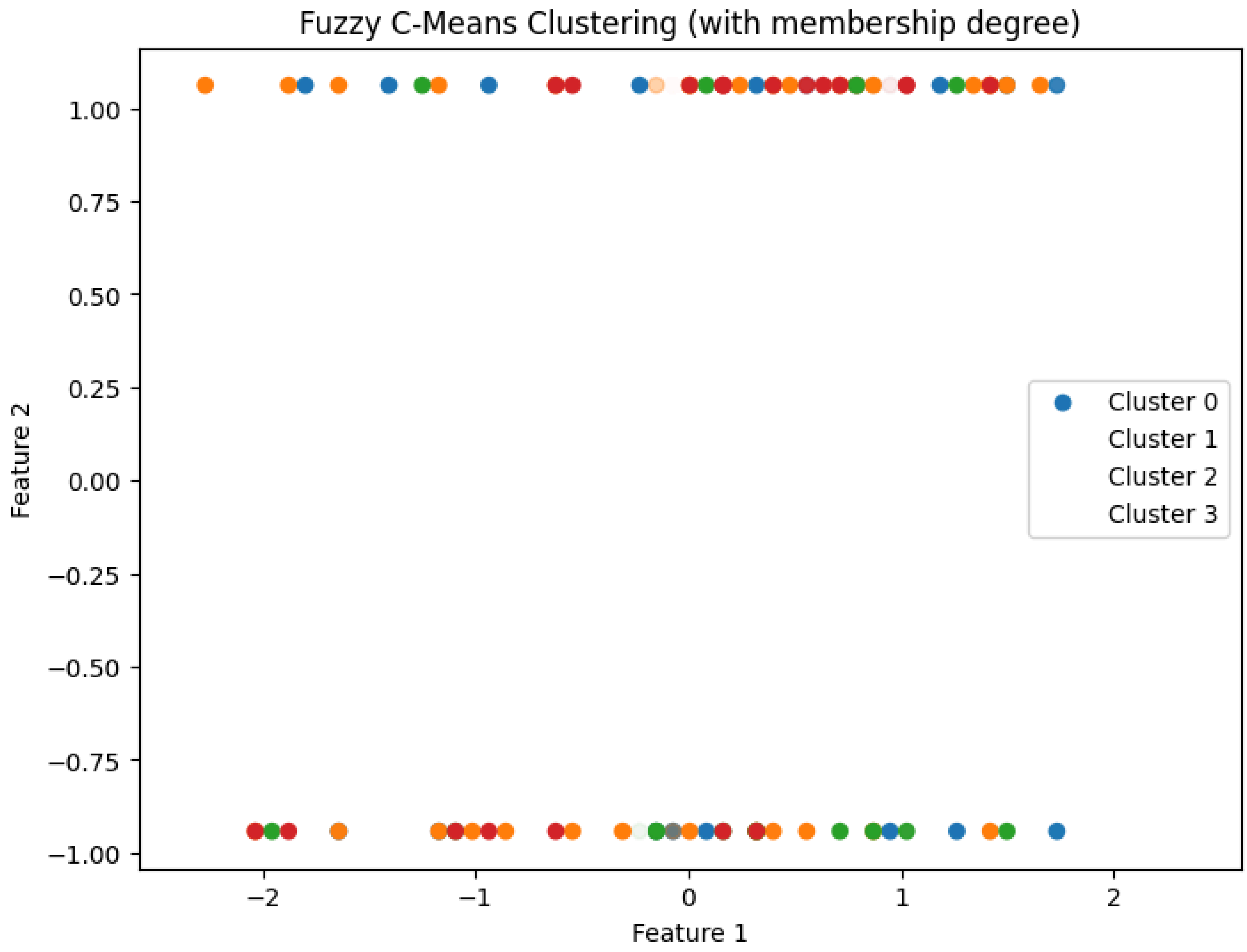
Out[6]: (4, 11)

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)
```

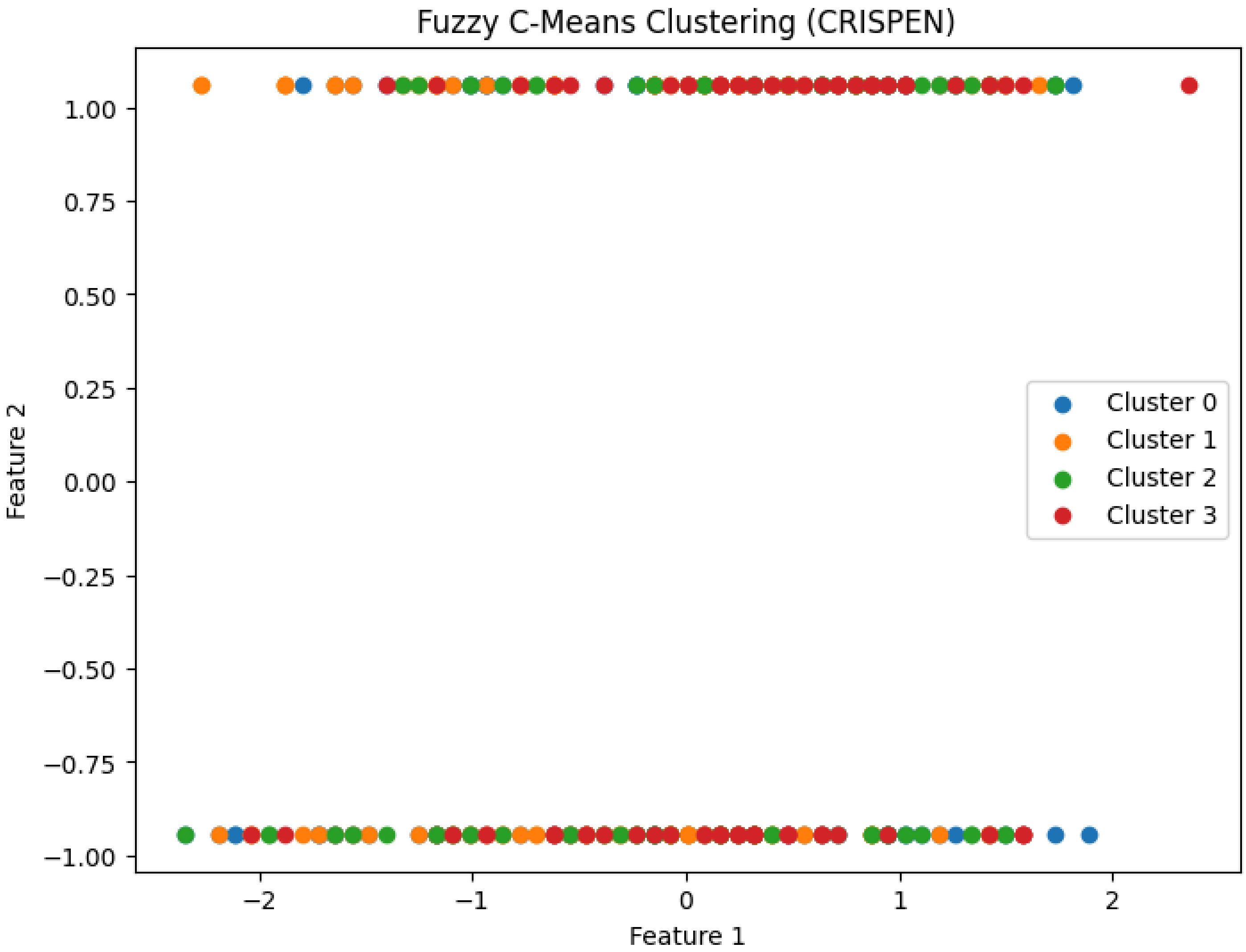
In [8]:
```
# Hard clustering from fuzzy membership
cluster_labels = np.argmax(u, axis=0)
print("Fuzzy partition coefficient (FPC):", fpc)

# 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, 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): 0.982771992053368
```

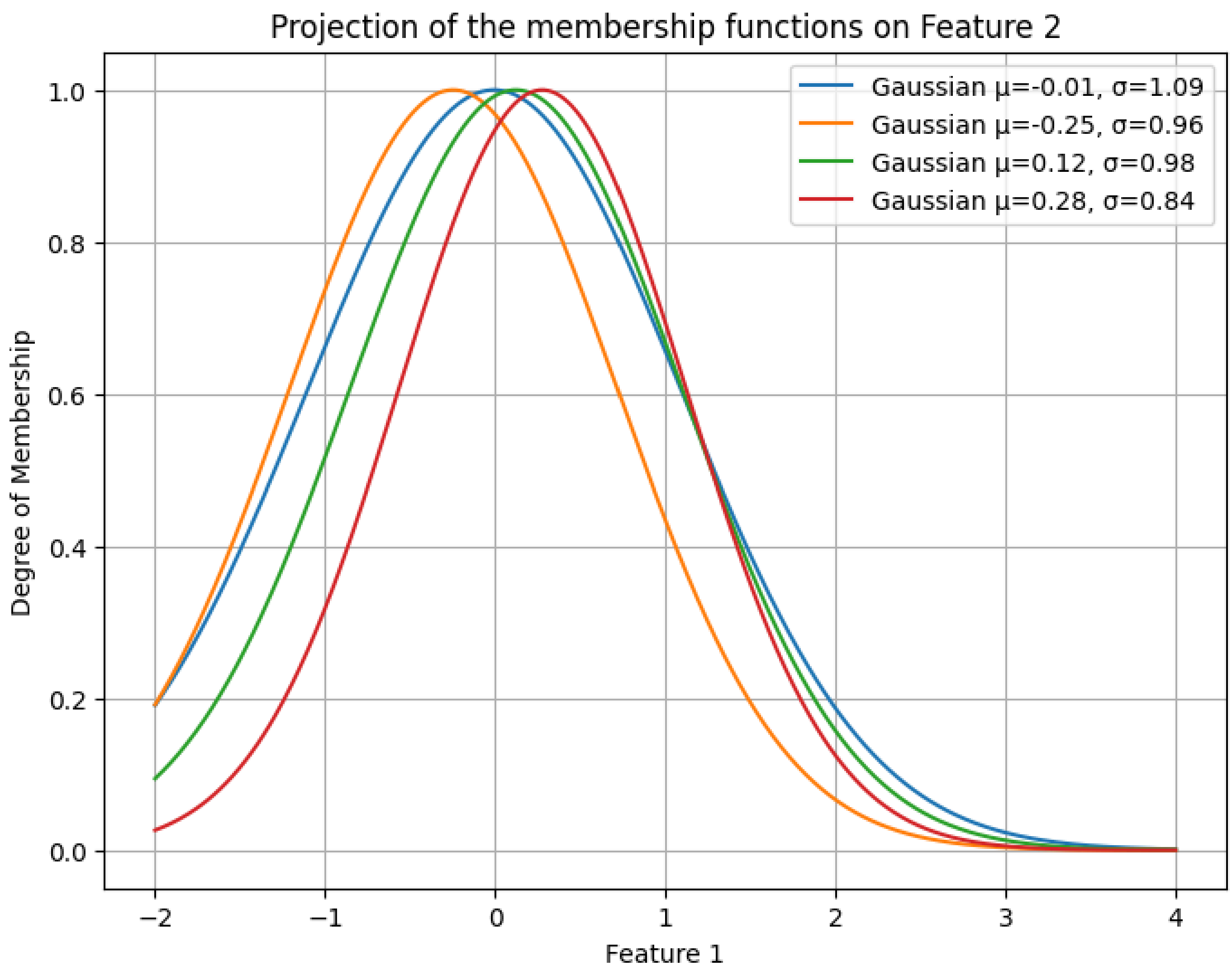


In [9]:
```
# Plot first two features with cluster assignments
plt.figure(figsize=(8,6))
for j in range(n_clusters):
    plt.scatter(
        Xexp[cluster_labels == j, 0],
        Xexp[cluster_labels == j, 1],
        label=f'Cluster {j}'
    )

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



In [10]:
```
# 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 [11]:
```
# --------------------------
# 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
```

```python
        # 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]:

```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 [14]:
```
# ---------------------------
# Hybrid Training (Classic ANFIS)
# ---------------------------
def train_hybrid_anfis(model, X, y, max_iters=10, gd_epochs=20, lr=1e-3): #10, 20, 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 [15]:
```
# ---------------------------
# Alternative Hybrid Training (LS+ gradient descent on all)
# ---------------------------
def train_hybrid(model, X, y, epochs=100, lr=1e-5): #def; 100, 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))
train_hybrid_anfis(model, Xtr, ytr.reshape(-1,1), max_iters=10, gd_epochs=14, lr=1e-4) #10 20 3
```

```
tensor(2382.7402, grad_fn=<MseLossBackward0>)
tensor(2382.0920, grad_fn=<MseLossBackward0>)
tensor(2381.4475, grad_fn=<MseLossBackward0>)
tensor(2380.8083, grad_fn=<MseLossBackward0>)
tensor(2380.1721, grad_fn=<MseLossBackward0>)
tensor(2379.5393, grad_fn=<MseLossBackward0>)
tensor(2378.9099, grad_fn=<MseLossBackward0>)
tensor(2378.2839, grad_fn=<MseLossBackward0>)
tensor(2377.6609, grad_fn=<MseLossBackward0>)
tensor(2377.0415, grad_fn=<MseLossBackward0>)
tensor(2376.4268, grad_fn=<MseLossBackward0>)
tensor(2375.8213, grad_fn=<MseLossBackward0>)
tensor(2375.2188, grad_fn=<MseLossBackward0>)
tensor(2374.6155, grad_fn=<MseLossBackward0>)
tensor(2373.7542, grad_fn=<MseLossBackward0>)
tensor(2373.1123, grad_fn=<MseLossBackward0>)
tensor(2372.4719, grad_fn=<MseLossBackward0>)
tensor(2371.8337, grad_fn=<MseLossBackward0>)
tensor(2371.1987, grad_fn=<MseLossBackward0>)
tensor(2370.5691, grad_fn=<MseLossBackward0>)
tensor(2369.9434, grad_fn=<MseLossBackward0>)
tensor(2369.3379, grad_fn=<MseLossBackward0>)
tensor(2368.7354, grad_fn=<MseLossBackward0>)
tensor(2368.1357, grad_fn=<MseLossBackward0>)
tensor(2367.5376, grad_fn=<MseLossBackward0>)
tensor(2366.9377, grad_fn=<MseLossBackward0>)
tensor(2366.3398, grad_fn=<MseLossBackward0>)
tensor(2365.7446, grad_fn=<MseLossBackward0>)
tensor(2364.9121, grad_fn=<MseLossBackward0>)
tensor(2364.2815, grad_fn=<MseLossBackward0>)
tensor(2363.6543, grad_fn=<MseLossBackward0>)
tensor(2363.0310, grad_fn=<MseLossBackward0>)
tensor(2362.4106, grad_fn=<MseLossBackward0>)
tensor(2361.7935, grad_fn=<MseLossBackward0>)
tensor(2361.1787, grad_fn=<MseLossBackward0>)
tensor(2360.5669, grad_fn=<MseLossBackward0>)
tensor(2359.9563, grad_fn=<MseLossBackward0>)
tensor(2359.3535, grad_fn=<MseLossBackward0>)
tensor(2358.7549, grad_fn=<MseLossBackward0>)
tensor(2358.1594, grad_fn=<MseLossBackward0>)
tensor(2357.5669, grad_fn=<MseLossBackward0>)
```

```
tensor(2356.9775, grad_fn=<MseLossBackward0>)
tensor(2356.1841, grad_fn=<MseLossBackward0>)
tensor(2355.5654, grad_fn=<MseLossBackward0>)
tensor(2354.9495, grad_fn=<MseLossBackward0>)
tensor(2354.3408, grad_fn=<MseLossBackward0>)
tensor(2353.7346, grad_fn=<MseLossBackward0>)
tensor(2353.1316, grad_fn=<MseLossBackward0>)
tensor(2352.5300, grad_fn=<MseLossBackward0>)
tensor(2351.9309, grad_fn=<MseLossBackward0>)
tensor(2351.3394, grad_fn=<MseLossBackward0>)
tensor(2350.7532, grad_fn=<MseLossBackward0>)
tensor(2350.1694, grad_fn=<MseLossBackward0>)
tensor(2349.5889, grad_fn=<MseLossBackward0>)
tensor(2349.0115, grad_fn=<MseLossBackward0>)
tensor(2348.4370, grad_fn=<MseLossBackward0>)
tensor(2347.6797, grad_fn=<MseLossBackward0>)
tensor(2347.0759, grad_fn=<MseLossBackward0>)
tensor(2346.4822, grad_fn=<MseLossBackward0>)
tensor(2345.8916, grad_fn=<MseLossBackward0>)
tensor(2345.3071, grad_fn=<MseLossBackward0>)
tensor(2344.7175, grad_fn=<MseLossBackward0>)
tensor(2344.1318, grad_fn=<MseLossBackward0>)
tensor(2343.5520, grad_fn=<MseLossBackward0>)
tensor(2342.9741, grad_fn=<MseLossBackward0>)
tensor(2342.3982, grad_fn=<MseLossBackward0>)
tensor(2341.8259, grad_fn=<MseLossBackward0>)
tensor(2341.2549, grad_fn=<MseLossBackward0>)
tensor(2340.6863, grad_fn=<MseLossBackward0>)
tensor(2340.1182, grad_fn=<MseLossBackward0>)
tensor(2339.3713, grad_fn=<MseLossBackward0>)
tensor(2338.7761, grad_fn=<MseLossBackward0>)
tensor(2338.1802, grad_fn=<MseLossBackward0>)
tensor(2337.5908, grad_fn=<MseLossBackward0>)
tensor(2337.0042, grad_fn=<MseLossBackward0>)
tensor(2336.4197, grad_fn=<MseLossBackward0>)
tensor(2335.8406, grad_fn=<MseLossBackward0>)
tensor(2335.2729, grad_fn=<MseLossBackward0>)
tensor(2334.7048, grad_fn=<MseLossBackward0>)
tensor(2334.1355, grad_fn=<MseLossBackward0>)
tensor(2333.5708, grad_fn=<MseLossBackward0>)
tensor(2333.0073, grad_fn=<MseLossBackward0>)
```

```
tensor(2332.4473, grad_fn=<MseLossBackward0>)
tensor(2331.8894, grad_fn=<MseLossBackward0>)
tensor(2331.1370, grad_fn=<MseLossBackward0>)
tensor(2330.5532, grad_fn=<MseLossBackward0>)
tensor(2329.9731, grad_fn=<MseLossBackward0>)
tensor(2329.3979, grad_fn=<MseLossBackward0>)
tensor(2328.8264, grad_fn=<MseLossBackward0>)
tensor(2328.2559, grad_fn=<MseLossBackward0>)
tensor(2327.6873, grad_fn=<MseLossBackward0>)
tensor(2327.1230, grad_fn=<MseLossBackward0>)
tensor(2326.5615, grad_fn=<MseLossBackward0>)
tensor(2326.0034, grad_fn=<MseLossBackward0>)
tensor(2325.4482, grad_fn=<MseLossBackward0>)
tensor(2324.8960, grad_fn=<MseLossBackward0>)
tensor(2324.3457, grad_fn=<MseLossBackward0>)
tensor(2323.8000, grad_fn=<MseLossBackward0>)
tensor(2323.0796, grad_fn=<MseLossBackward0>)
tensor(2322.5195, grad_fn=<MseLossBackward0>)
tensor(2321.9587, grad_fn=<MseLossBackward0>)
tensor(2321.3984, grad_fn=<MseLossBackward0>)
tensor(2320.8430, grad_fn=<MseLossBackward0>)
tensor(2320.2932, grad_fn=<MseLossBackward0>)
tensor(2319.7451, grad_fn=<MseLossBackward0>)
tensor(2319.2026, grad_fn=<MseLossBackward0>)
tensor(2318.6626, grad_fn=<MseLossBackward0>)
tensor(2318.1272, grad_fn=<MseLossBackward0>)
tensor(2317.5950, grad_fn=<MseLossBackward0>)
tensor(2317.0642, grad_fn=<MseLossBackward0>)
tensor(2316.5383, grad_fn=<MseLossBackward0>)
tensor(2316.0144, grad_fn=<MseLossBackward0>)
tensor(2315.3186, grad_fn=<MseLossBackward0>)
tensor(2314.7651, grad_fn=<MseLossBackward0>)
tensor(2314.2065, grad_fn=<MseLossBackward0>)
tensor(2313.6548, grad_fn=<MseLossBackward0>)
tensor(2313.1030, grad_fn=<MseLossBackward0>)
tensor(2312.5525, grad_fn=<MseLossBackward0>)
tensor(2312.0039, grad_fn=<MseLossBackward0>)
tensor(2311.4597, grad_fn=<MseLossBackward0>)
tensor(2310.9177, grad_fn=<MseLossBackward0>)
tensor(2310.3774, grad_fn=<MseLossBackward0>)
tensor(2309.8406, grad_fn=<MseLossBackward0>)
```

```
tensor(2309.3066, grad_fn=<MseLossBackward0>)
tensor(2308.7769, grad_fn=<MseLossBackward0>)
tensor(2308.2498, grad_fn=<MseLossBackward0>)
tensor(2307.5374, grad_fn=<MseLossBackward0>)
tensor(2306.9841, grad_fn=<MseLossBackward0>)
tensor(2306.4329, grad_fn=<MseLossBackward0>)
tensor(2305.8892, grad_fn=<MseLossBackward0>)
tensor(2305.3472, grad_fn=<MseLossBackward0>)
tensor(2304.8159, grad_fn=<MseLossBackward0>)
tensor(2304.2869, grad_fn=<MseLossBackward0>)
tensor(2303.7603, grad_fn=<MseLossBackward0>)
tensor(2303.2358, grad_fn=<MseLossBackward0>)
tensor(2302.7139, grad_fn=<MseLossBackward0>)
tensor(2302.1946, grad_fn=<MseLossBackward0>)
tensor(2301.6787, grad_fn=<MseLossBackward0>)
tensor(2301.1660, grad_fn=<MseLossBackward0>)
tensor(2300.6550, grad_fn=<MseLossBackward0>)
```

In [18]:

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

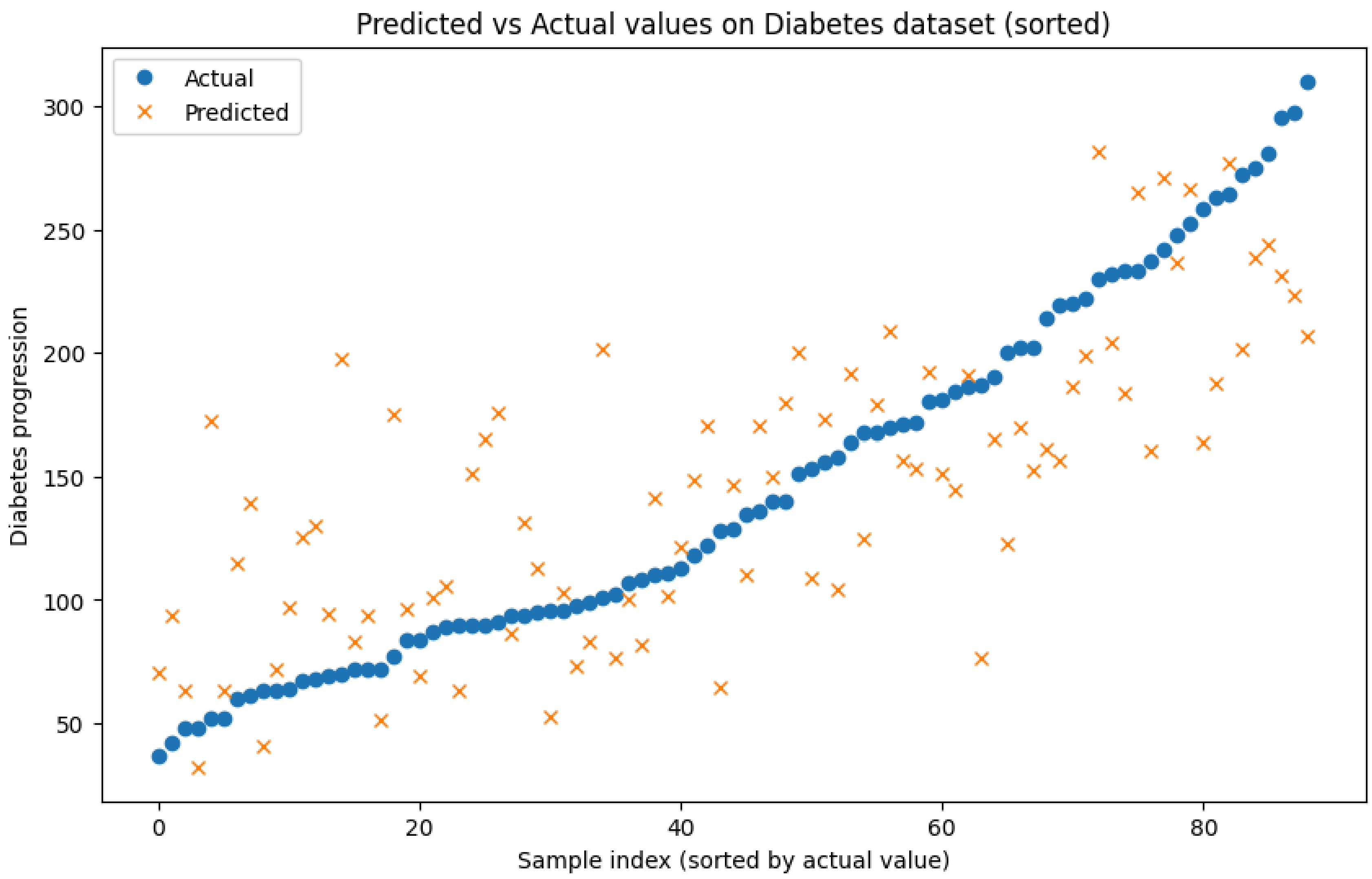
```
MSE:2391.4755859375
```

In comparison with TSK model previous used, it was possible to slightly improve the results, going from a MSE of 2476.79 to 2391.48. The change was not significative. Given the range of the target values, the MSE obtain represents a large value, not giving confidence for a certain predicted value. When visualizing the error (as ploted in the chart bellow) it is possible to notice that, bigger the value of the targer, beter it predicts it. It is possible to verify that the predicted values tend to map the trend (if the real value is bigger, the predicted one tend to be bigger as well). The problem is that it seems to be affected by a large "noise", responsible for the large MSE value obtained.

In [19]:

```
# Plot predictions vs actual
# Converter y_pred para numpy e flatten
y_pred_np = y_pred.detach().numpy().flatten()
yte_np = yte.detach().numpy()

# Obter índices que ordenam yte
sort_idx = np.argsort(yte_np)
```

```
# Ordenar yte e y_pred segundo esses índices
yte_sorted = yte_np[sort_idx]
y_pred_sorted = y_pred_np[sort_idx]

# Plot
plt.figure(figsize=(10,6))
plt.plot(range(len(yte_sorted)), yte_sorted, label="Actual", marker="o", linestyle='')
plt.plot(range(len(y_pred_sorted)), y_pred_sorted, label="Predicted", marker="x", linestyle='')

plt.xlabel("Sample index (sorted by actual value)")
plt.ylabel("Diabetes progression")
plt.title("Predicted vs Actual values on Diabetes dataset (sorted)")
plt.legend()
plt.show()
```


Predicted vs Actual values on Diabetes dataset (sorted)
Actual
Predicted
Diabetes progression
Sample index (sorted by actual value)
300
250
200
150
100
50
0
20
40
60
80

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 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 [2]:
```
# CHOOSE DATASET

# Binary classification dataset
diabetes = datasets.load_diabetes(as_frame=True)

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

X = diabetes.data.values
y = diabetes.target.values

print("Shape:", X.shape)

print(diabetes.data.head(), "\n \n")# first rows of features
print(diabetes.target.head()) # first rows of target
```

```
Shape: (442, 10)
        age       sex       bmi        bp        s1        s2        s3  \
0  0.038076  0.050680  0.061696  0.021872 -0.044223 -0.034821 -0.043401
1 -0.001882 -0.044642 -0.051474 -0.026328 -0.008449 -0.019163  0.074412
2  0.085299  0.050680  0.044451 -0.005670 -0.045599 -0.034194 -0.032356
3 -0.089063 -0.044642 -0.011595 -0.036656  0.012191  0.024991 -0.036038
4  0.005383 -0.044642 -0.036385  0.021872  0.003935  0.015596  0.008142

         s4        s5        s6
0 -0.002592  0.019907 -0.017646
1 -0.039493 -0.068332 -0.092204
2 -0.002592  0.002861 -0.025930
3  0.034309  0.022688 -0.009362
4 -0.002592 -0.031988 -0.046641


0    151.0
1     75.0
2    141.0
3    206.0
4    135.0
Name: target, dtype: float64
```

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)
```

A fixed seed was added to this code to ensure the reproducibility of the analysis. This allowed for manual tuning of the hyperparameters to achieve better model training.

In [5]:

```
import random

seed = 42
torch.manual_seed(seed)
```

```
np.random.seed(seed)
random.seed(seed)

# Para GPU
torch.cuda.manual_seed(seed)
torch.cuda.manual_seed_all(seed)

# Tornar CUDA determinístico
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

In [6]:
```
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
```

In [7]:
```
num_epochs=190
lr=0.01
dropout=0.25
batch_size=128
```

This model was trained on the GPU and then transferred to the CPU for use with NumPy. Given the low number of parameters (i.e., the model's low complexity), the time required on the CPU was similar to that on the GPU.

In [8]:
```
# Model, Loss, Optimizer
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
#device = "cpu"  # force to use CPU
print(device)
```

```
cuda
```

In [9]:
```
Xtr = torch.tensor(Xtr, dtype=torch.float32).to(device)
ytr = torch.tensor(ytr, dtype=torch.float32).to(device)
Xte = torch.tensor(Xte, dtype=torch.float32).to(device)
yte = torch.tensor(yte, dtype=torch.float32).to("cpu")

# 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 [10]:
```
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)
```

The model was implemented as a fully connected neural network (MLP) with four hidden layers of 64 neurons each, using ReLU activation functions. The network takes the input features of the dataset and outputs a single value for regression (Diabetes Progression).

In [11]:
```
# Training Loop
import time
start_time = time.time()
for epoch in range(num_epochs):
```

```
    model.train()
    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()
        optimizer.step()

        epoch_loss += loss.item()

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

end_time = time.time()
print(f"Training time: {end_time - start_time:.2f} seconds")
```

```
Epoch [1/190], Loss: 29648.7161
Epoch [2/190], Loss: 27738.2480
Epoch [3/190], Loss: 19526.0052
Epoch [4/190], Loss: 10916.4779
Epoch [5/190], Loss: 7908.8825
Epoch [6/190], Loss: 6315.4227
Epoch [7/190], Loss: 6881.3011
Epoch [8/190], Loss: 5394.8262
Epoch [9/190], Loss: 5833.4985
Epoch [10/190], Loss: 4486.8579
Epoch [11/190], Loss: 4801.0705
Epoch [12/190], Loss: 4380.2632
Epoch [13/190], Loss: 4705.3464
Epoch [14/190], Loss: 4574.8371
Epoch [15/190], Loss: 4115.9676
Epoch [16/190], Loss: 4263.3736
Epoch [17/190], Loss: 4220.0199
Epoch [18/190], Loss: 3679.3091
Epoch [19/190], Loss: 3745.2568
Epoch [20/190], Loss: 3896.5340
Epoch [21/190], Loss: 4014.5900
Epoch [22/190], Loss: 3893.0127
Epoch [23/190], Loss: 3573.5227
Epoch [24/190], Loss: 3891.4432
Epoch [25/190], Loss: 3775.5807
Epoch [26/190], Loss: 3753.8115
Epoch [27/190], Loss: 3697.5588
Epoch [28/190], Loss: 3638.7832
Epoch [29/190], Loss: 4002.4415
Epoch [30/190], Loss: 3921.0071
Epoch [31/190], Loss: 3542.5956
Epoch [32/190], Loss: 3535.1024
Epoch [33/190], Loss: 3598.4771
Epoch [34/190], Loss: 3983.6483
Epoch [35/190], Loss: 3522.1357
Epoch [36/190], Loss: 3819.0868
Epoch [37/190], Loss: 3664.6637
Epoch [38/190], Loss: 3815.6850
Epoch [39/190], Loss: 3901.3689
Epoch [40/190], Loss: 3601.0779
Epoch [41/190], Loss: 3759.6592
```

```
Epoch [42/190], Loss: 3391.2233
Epoch [43/190], Loss: 3242.0986
Epoch [44/190], Loss: 3577.6099
Epoch [45/190], Loss: 3480.0977
Epoch [46/190], Loss: 3555.8261
Epoch [47/190], Loss: 3476.3621
Epoch [48/190], Loss: 3268.5854
Epoch [49/190], Loss: 3618.4499
Epoch [50/190], Loss: 3584.3310
Epoch [51/190], Loss: 3675.5367
Epoch [52/190], Loss: 3722.2685
Epoch [53/190], Loss: 3227.3150
Epoch [54/190], Loss: 2974.1118
Epoch [55/190], Loss: 3562.6582
Epoch [56/190], Loss: 3198.2245
Epoch [57/190], Loss: 3238.8750
Epoch [58/190], Loss: 3376.6782
Epoch [59/190], Loss: 3346.6515
Epoch [60/190], Loss: 3468.1538
Epoch [61/190], Loss: 3152.5792
Epoch [62/190], Loss: 3484.5428
Epoch [63/190], Loss: 3397.3236
Epoch [64/190], Loss: 3454.3319
Epoch [65/190], Loss: 3955.6242
Epoch [66/190], Loss: 3623.3734
Epoch [67/190], Loss: 3310.0218
Epoch [68/190], Loss: 3456.5499
Epoch [69/190], Loss: 3696.4705
Epoch [70/190], Loss: 3757.5662
Epoch [71/190], Loss: 3274.0444
Epoch [72/190], Loss: 3489.4048
Epoch [73/190], Loss: 3018.0824
Epoch [74/190], Loss: 3315.5653
Epoch [75/190], Loss: 3203.5889
Epoch [76/190], Loss: 3338.0278
Epoch [77/190], Loss: 3350.0305
Epoch [78/190], Loss: 3378.0791
Epoch [79/190], Loss: 3423.2143
Epoch [80/190], Loss: 3527.6632
Epoch [81/190], Loss: 3268.2509
Epoch [82/190], Loss: 3227.4768
```

```
Epoch [83/190], Loss: 3589.3934
Epoch [84/190], Loss: 3299.7594
Epoch [85/190], Loss: 3385.5459
Epoch [86/190], Loss: 3498.3327
Epoch [87/190], Loss: 3477.1771
Epoch [88/190], Loss: 3174.3293
Epoch [89/190], Loss: 3714.6216
Epoch [90/190], Loss: 3343.3033
Epoch [91/190], Loss: 3414.0710
Epoch [92/190], Loss: 3248.5864
Epoch [93/190], Loss: 3488.2712
Epoch [94/190], Loss: 3373.2797
Epoch [95/190], Loss: 3372.9124
Epoch [96/190], Loss: 3554.2501
Epoch [97/190], Loss: 3368.5915
Epoch [98/190], Loss: 3215.7687
Epoch [99/190], Loss: 3260.8147
Epoch [100/190], Loss: 3192.8174
Epoch [101/190], Loss: 3423.2056
Epoch [102/190], Loss: 2991.1534
Epoch [103/190], Loss: 3152.7769
Epoch [104/190], Loss: 3303.9213
Epoch [105/190], Loss: 3282.2486
Epoch [106/190], Loss: 3331.6805
Epoch [107/190], Loss: 3279.8656
Epoch [108/190], Loss: 3262.2849
Epoch [109/190], Loss: 3125.2458
Epoch [110/190], Loss: 3407.5113
Epoch [111/190], Loss: 3498.1632
Epoch [112/190], Loss: 3553.5989
Epoch [113/190], Loss: 3089.8211
Epoch [114/190], Loss: 3075.9001
Epoch [115/190], Loss: 3539.4534
Epoch [116/190], Loss: 2961.9173
Epoch [117/190], Loss: 3655.1819
Epoch [118/190], Loss: 3049.0596
Epoch [119/190], Loss: 3615.5492
Epoch [120/190], Loss: 3256.5053
Epoch [121/190], Loss: 3308.7235
Epoch [122/190], Loss: 3240.1512
Epoch [123/190], Loss: 3193.3521
```

```
Epoch [124/190], Loss: 3216.8752
Epoch [125/190], Loss: 3073.2205
Epoch [126/190], Loss: 3247.6636
Epoch [127/190], Loss: 3242.5971
Epoch [128/190], Loss: 3164.6580
Epoch [129/190], Loss: 3109.4061
Epoch [130/190], Loss: 3200.9823
Epoch [131/190], Loss: 2903.9763
Epoch [132/190], Loss: 3002.0817
Epoch [133/190], Loss: 3106.7635
Epoch [134/190], Loss: 3022.0481
Epoch [135/190], Loss: 2955.9606
Epoch [136/190], Loss: 2726.1379
Epoch [137/190], Loss: 2718.2030
Epoch [138/190], Loss: 2962.2109
Epoch [139/190], Loss: 2859.1110
Epoch [140/190], Loss: 2965.7663
Epoch [141/190], Loss: 2804.8240
Epoch [142/190], Loss: 3106.8247
Epoch [143/190], Loss: 2979.2811
Epoch [144/190], Loss: 3120.4154
Epoch [145/190], Loss: 3010.3355
Epoch [146/190], Loss: 2924.7756
Epoch [147/190], Loss: 3123.8313
Epoch [148/190], Loss: 3109.5750
Epoch [149/190], Loss: 3098.7642
Epoch [150/190], Loss: 2857.3700
Epoch [151/190], Loss: 2996.3384
Epoch [152/190], Loss: 3129.6170
Epoch [153/190], Loss: 2759.2476
Epoch [154/190], Loss: 3000.5859
Epoch [155/190], Loss: 2925.3764
Epoch [156/190], Loss: 2861.1976
Epoch [157/190], Loss: 2986.6821
Epoch [158/190], Loss: 2743.7799
Epoch [159/190], Loss: 3126.5764
Epoch [160/190], Loss: 2914.5562
Epoch [161/190], Loss: 2880.0532
Epoch [162/190], Loss: 2857.0591
Epoch [163/190], Loss: 2817.7529
Epoch [164/190], Loss: 2701.1991
```

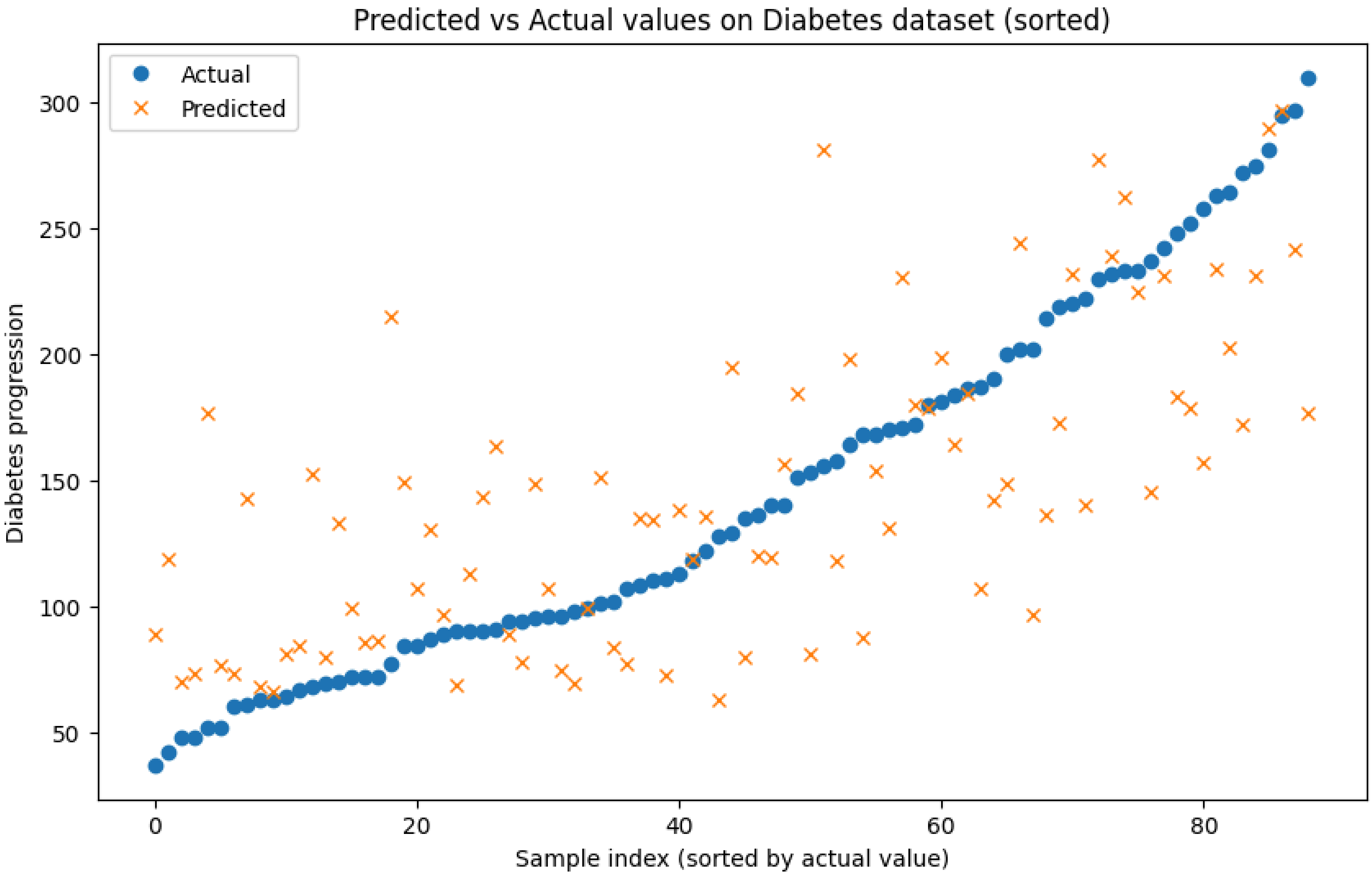
```
Epoch [165/190], Loss: 3057.9538
Epoch [166/190], Loss: 2827.3732
Epoch [167/190], Loss: 2802.8832
Epoch [168/190], Loss: 3131.7845
Epoch [169/190], Loss: 3075.6080
Epoch [170/190], Loss: 2931.3674
Epoch [171/190], Loss: 2838.9221
Epoch [172/190], Loss: 3041.7693
Epoch [173/190], Loss: 3094.6465
Epoch [174/190], Loss: 3209.8215
Epoch [175/190], Loss: 3203.1895
Epoch [176/190], Loss: 3175.8590
Epoch [177/190], Loss: 2773.8187
Epoch [178/190], Loss: 3101.5061
Epoch [179/190], Loss: 3159.9993
Epoch [180/190], Loss: 2911.1960
Epoch [181/190], Loss: 2810.7509
Epoch [182/190], Loss: 2745.5351
Epoch [183/190], Loss: 2873.0893
Epoch [184/190], Loss: 3086.1829
Epoch [185/190], Loss: 3048.7624
Epoch [186/190], Loss: 2893.8034
Epoch [187/190], Loss: 2915.0261
Epoch [188/190], Loss: 2572.4602
Epoch [189/190], Loss: 2886.5369
Epoch [190/190], Loss: 2821.7037
Training time: 1.99 seconds
```

In [12]:
```
y_pred=model(Xte).cpu()  # só nesta altura volta ao CPU
#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
```

```
MSE:2814.833251953125
```

After tuning the hyperparameters, the best configuration was: num_epochs = 200, lr = 0.02, dropout = 0.2, and batch_size = 128.A dropout with a probability of 0.2 was applied after each layer to mitigate overfitting.It order to validate the model the MSE was computed. The model was evaluated using the Mean Squared Error (MSE), which resulted in a high value of 2815. This indicates that, despite the training and parameter optimization, the current approach is insufficient to accurately predict Diabetes Progression. The high error may be attributed to the limited size of the dataset, the low number of features. In comparison with previous approaches, such as ANFIS or TSK models, this approach exhibited the worst performance.

In [13]:
```
# Plot predictions vs actual
# Converter y_pred para numpy e flatten
y_pred_np = y_pred.detach().numpy().flatten()
yte_np = yte.detach().numpy()

# Obter índices que ordenam yte
sort_idx = np.argsort(yte_np)

# Ordenar yte e y_pred segundo esses índices
yte_sorted = yte_np[sort_idx]
y_pred_sorted = y_pred_np[sort_idx]

# Plot
plt.figure(figsize=(10,6))
plt.plot(range(len(yte_sorted)), yte_sorted, label="Actual", marker="o", linestyle='')
plt.plot(range(len(y_pred_sorted)), y_pred_sorted, label="Predicted", marker="x", linestyle='')

plt.xlabel("Sample index (sorted by actual value)")
plt.ylabel("Diabetes progression")
plt.title("Predicted vs Actual values on Diabetes dataset (sorted)")
plt.legend()
plt.show()
```


Predicted vs Actual values on Diabetes dataset (sorted)
Actual
Predicted
Diabetes progression
300
250
200
150
100
50
0
20
40
60
80
Sample index (sorted by actual value)

In [96]:
```
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error,accuracy_score,classification_report, confusion_matrix

import skfuzzy as fuzz
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
import pandas
```

Importation of Dataset 2

In [97]:
```
# CHOOSE DATASET

# Binary classification dataset
diabetes =fetch_openml("diabetes", version = 1, as_frame=True)

X = diabetes.data.values
y = diabetes.target.values

y = np.where(y == "tested_positive", 1, 0)

print("Shape:", X.shape)

print(diabetes.data.head(), "\n \n")# first rows of features
print(diabetes.target.head()) # first rows of target
```

```
Shape: (768, 8)
   preg  plas  pres  skin  insu  mass   pedi  age
0     6   148    72    35     0  33.6  0.627   50
1     1    85    66    29     0  26.6  0.351   31
2     8   183    64     0     0  23.3  0.672   32
3     1    89    66    23    94  28.1  0.167   21
4     0   137    40    35   168  43.1  2.288   33


0    tested_positive
1    tested_negative
2    tested_positive
3    tested_negative
4    tested_positive
Name: class, dtype: category
Categories (2, object): ['tested_negative', 'tested_positive']
```

In [98]:
```
#train test spliting
test_size=0.2
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=test_size, random_state=42)
```

In [99]:
```
# Standardize features
scaler=StandardScaler()
Xtr= scaler.fit_transform(Xtr)
Xte= scaler.transform(Xte)
```

In order to be able to compare the results, the number of clusters and value of m used, was the same as the one in the previous assigment (n_clusters = 4; m=1.1)

In [100…
```
 # Number of clusters
n_clusters = 4
m=1.1

# 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
```

```python
# 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 [101…

```python
centers.shape
```

Out[101…

```
(4, 9)
```

In [102…

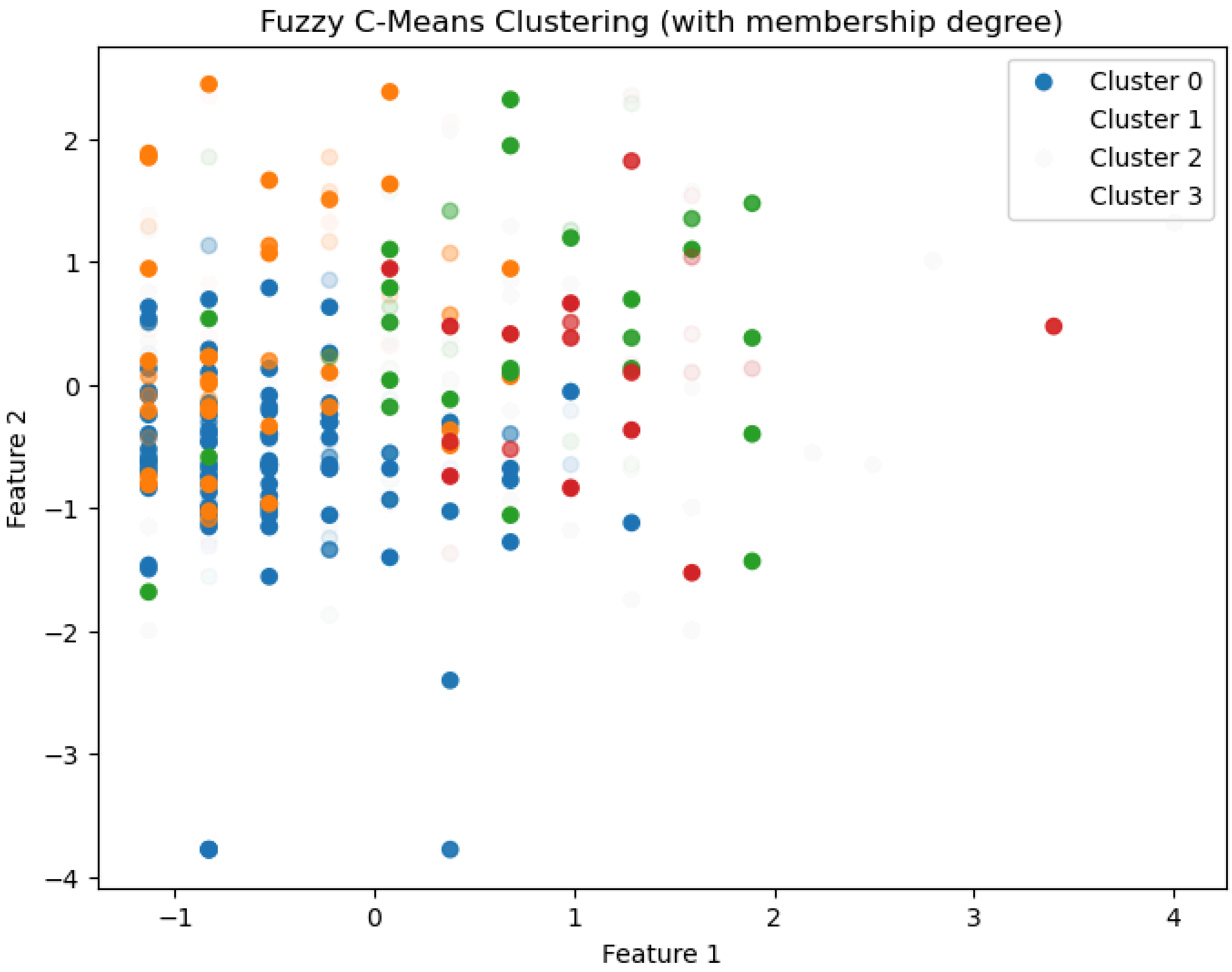
```python
# Compute sigma (spread) for each cluster
sigmas = []
for j in range(n_clusters):
    # membership weights for cluster j, raised to m
    u_j = u[j, :] ** m
    # weighted variance for each feature
    var_j = np.average((Xexp - centers[j])**2, axis=0, weights=u_j)
    sigma_j = np.sqrt(var_j)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

In [103…

```python
# 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()
```

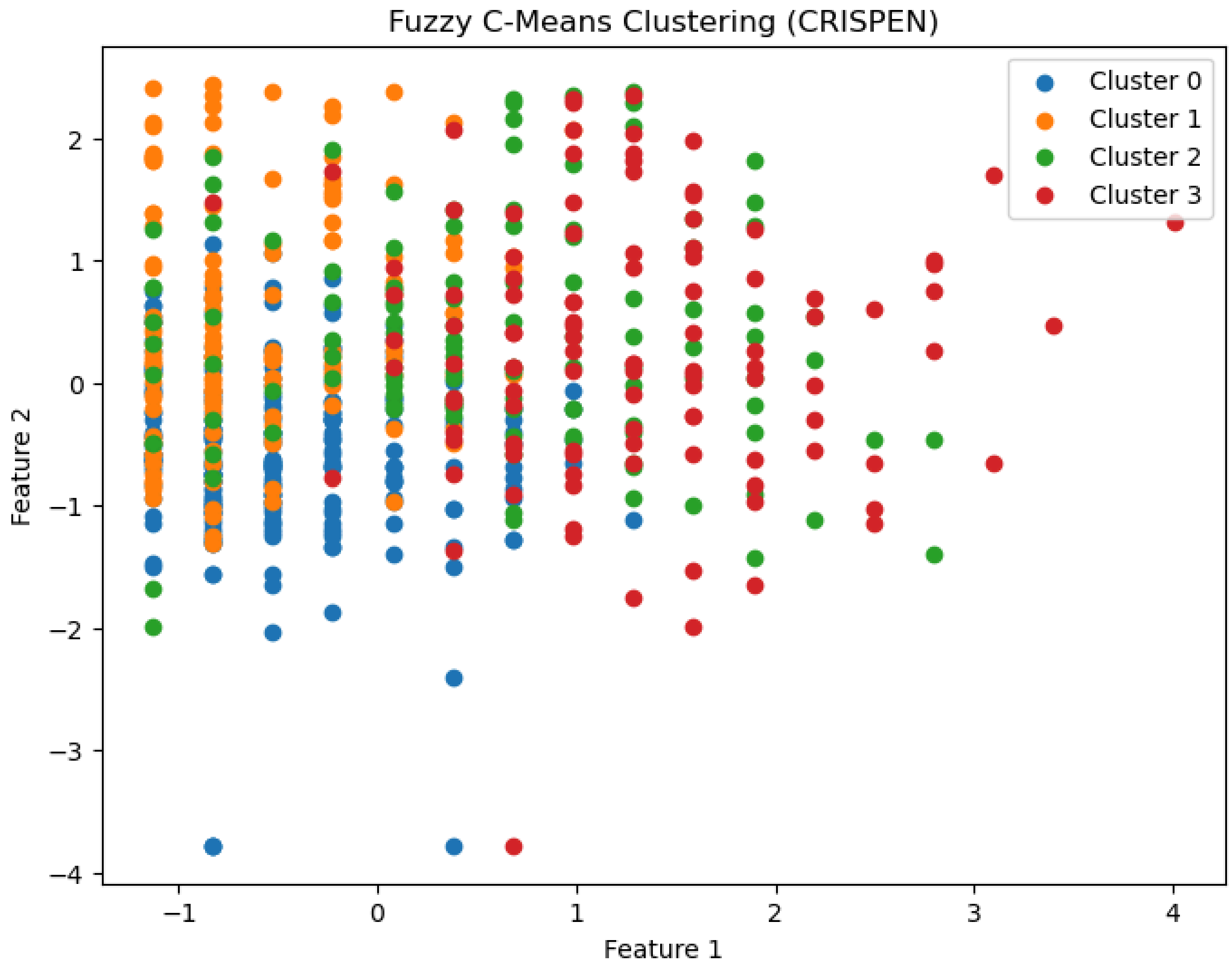
```
Fuzzy partition coefficient (FPC): 0.9091786112603655
```



In [104…

```
# Plot first two features with cluster assignments
plt.figure(figsize=(8,6))
for j in range(n_clusters):
    plt.scatter(
```

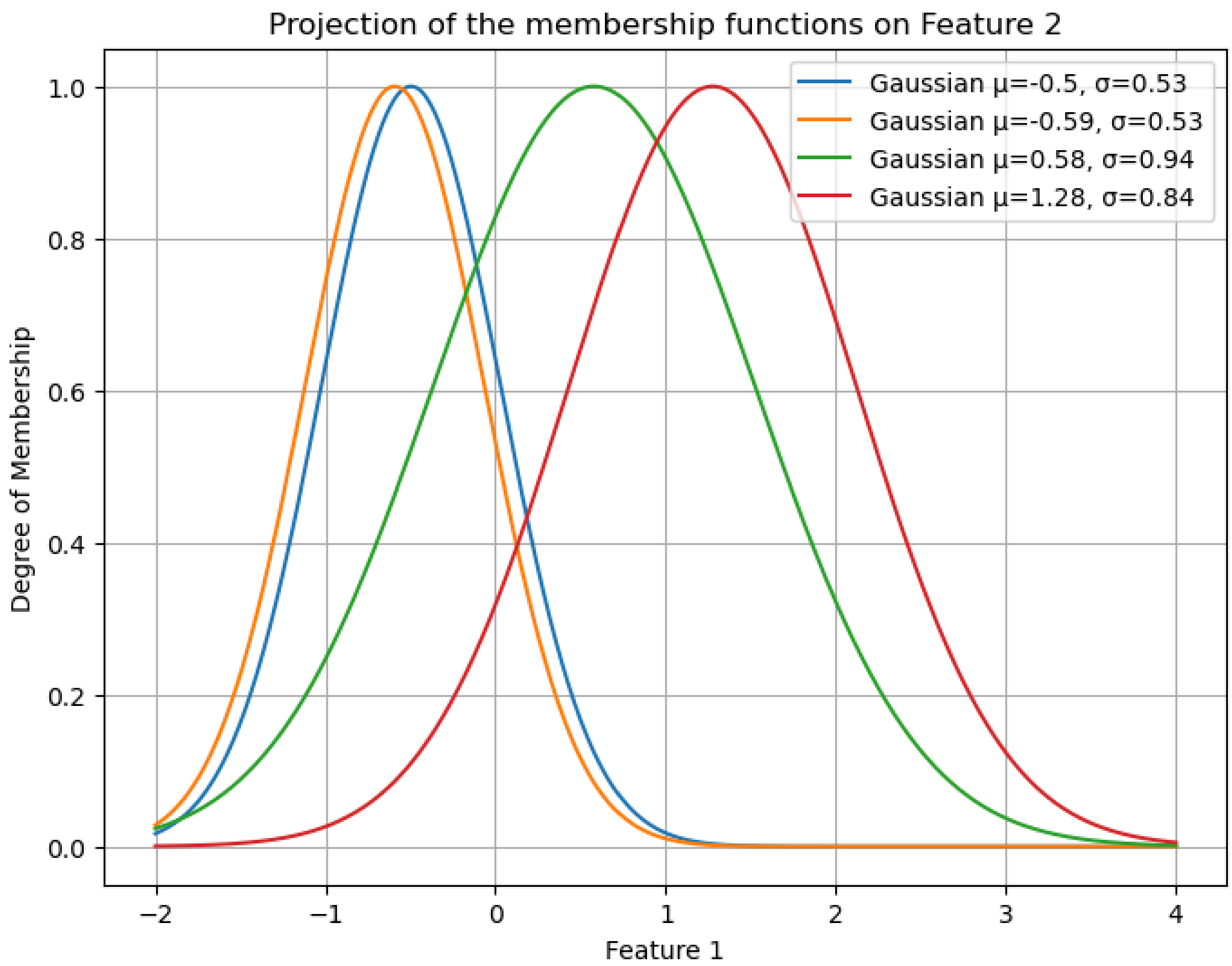
```
        Xexp[cluster_labels == j, 0],
        Xexp[cluster_labels == j, 1],
        label=f'Cluster {j}'
    )

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



In [105…

```
# 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 [106…

```
# ---------------------------
# 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 [107…

```
# ---------------------------
# 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 [108…

```
# ---------------------------
# 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 [109…

```
# ---------------------------
# Hybrid Training (Classic ANFIS)
# ---------------------------
def train_hybrid_anfis(model, X, y, max_iters=10, gd_epochs=20, lr=1e-3): #10, 20, 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 [110…

```
# ---------------------------
# Alternative Hybrid Training (LS+ gradient descent on all)
# ---------------------------
def train_hybrid(model, X, y, epochs=100, lr=1e-5): #def; 100, 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 [111…

```
# 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 [112…

```
# Training with LS:
#train_ls(model, Xtr, ytr.reshape(-1,1))
train_hybrid_anfis(model, Xtr, ytr.reshape(-1,1), max_iters=13, gd_epochs=11, lr=1e-4) #10 20 3
```

In [113…

```
thr = 0.6 #threshold to tune

y_pred, _, _=model(Xte)
y_true = yte.detach().numpy()
y_pred_bin = (y_pred.detach().numpy() > thr).astype(int)  # binary predictions

# Accuracy
acc = accuracy_score(y_true, y_pred_bin)
print(f'ACC:{accuracy_score(y_true,y_pred_bin)}') #classification

cm = confusion_matrix(y_true, y_pred_bin)
print("Confusion Matrix:")
print(cm)

#print(f'ACC:{mean_squared_error(yte.detach().numpy(),y_pred.detach().numpy())}') #regression
```

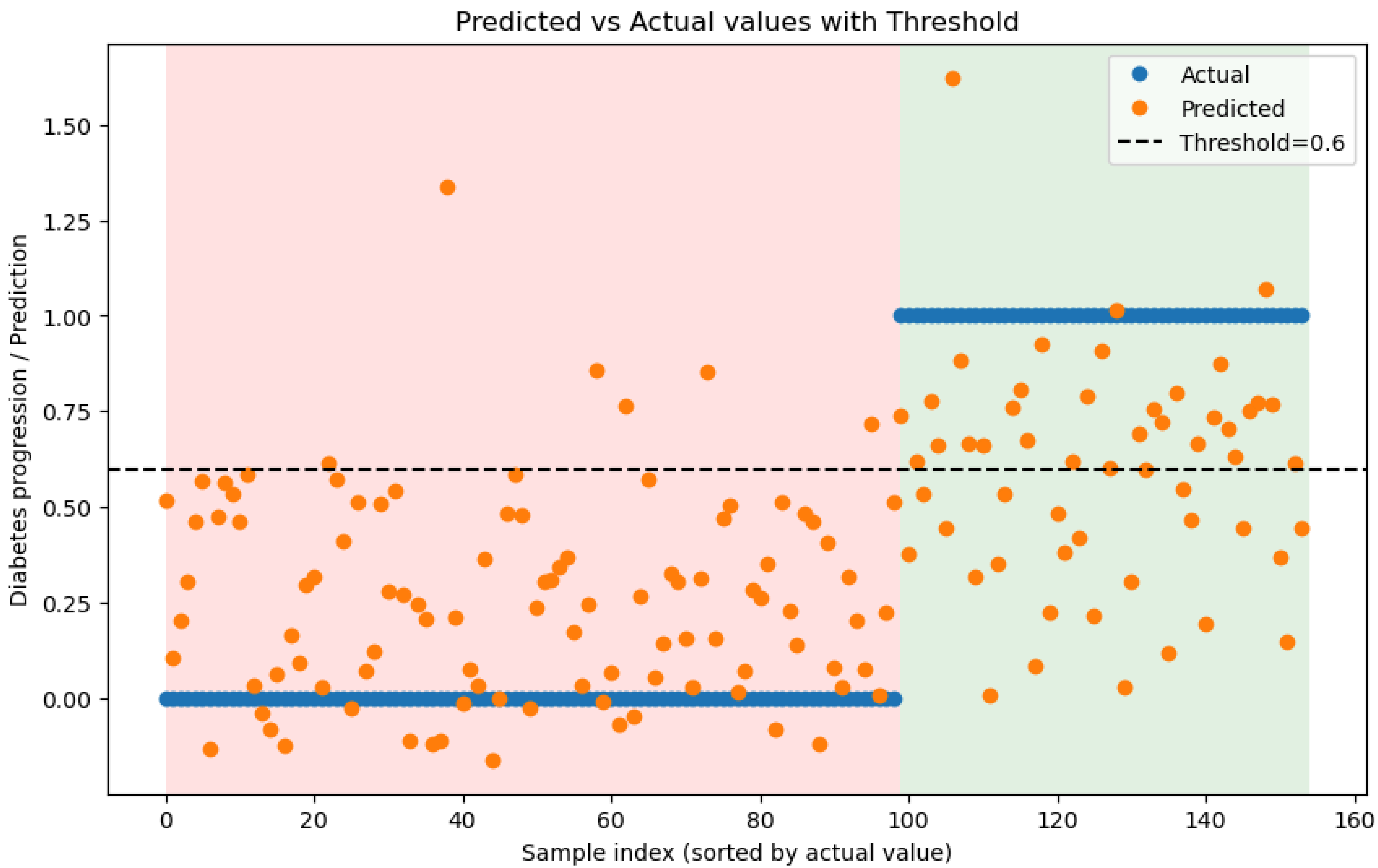
```
ACC:0.8116883116883117
Confusion Matrix:
[[93  6]
 [23 32]]
```

After manual tunning, the best model was achieved after changing the hyperparameters to: max_iters=13, gd_epochs=11, lr=1e-4. In comparison with TSK model previous used, it was possible to slightly improve the results, going from a accuracy score of 79.87% to 81.17%. The change was not significative, beeing able to only reduce 2 FP values (from 8 to 6). When visualizing the error (as ploted in the chart bellow) it is possible to notice that the model mutch better classifies for negative examples, represent by a clear separation in the predicted target distribution. To predict the positive exemples to model have more dificulty, presenting a recall score of 58.19% (TP/(TP+FN)).

In [114…

```
# Convert y_pred to numpy and flatten
y_pred_np = y_pred.detach().numpy().flatten()
```

```
y_pred_bin_np = y_pred_bin
yte_np = yte.detach().numpy()

# Get indices that sort yte
sort_idx = np.argsort(yte_np)

# Sort yte and y_pred
yte_sorted = yte_np[sort_idx]
y_pred_sorted = y_pred_np[sort_idx]

# Determine the index where actual target switches from 0 to 1
frontier_idx = np.argmax(yte_sorted == 1)  # first occurrence of 1

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

# Shade negative region (yte = 0)
plt.axvspan(0, frontier_idx, facecolor='red', alpha=0.1)

# Shade positive region (yte = 1)
plt.axvspan(frontier_idx, len(yte_sorted), facecolor='green', alpha=0.1)

# Actual and predicted points
plt.plot(range(len(yte_sorted)), yte_sorted, label="Actual", marker="o", linestyle='')
plt.plot(range(len(y_pred_sorted)), y_pred_sorted, label="Predicted", marker="o", linestyle='')

# Threshold line
plt.axhline(y=thr, color='k', linestyle='--', label=f'Threshold={thr}')

plt.xlabel("Sample index (sorted by actual value)")
plt.ylabel("Diabetes progression / Prediction")
plt.title("Predicted vs Actual values with Threshold")
plt.legend()
plt.show()
```


Predicted vs Actual values with Threshold
Actual
Predicted
Threshold=0.6
Diabetes progression / Prediction
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
20
40
60
80
100
120
140
160
Sample index (sorted by actual value)

In [9]:
```
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error,accuracy_score,classification_report, confusion_matrix
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
```

Importation of Dataset 2

In [10]:
```
# CHOOSE DATASET

# Binary classification dataset
diabetes =fetch_openml("diabetes", version = 1, as_frame=True)

X = diabetes.data.values
y = diabetes.target.values

y = np.where(y == "tested_positive", 1, 0)

print("Shape:", X.shape)

print(diabetes.data.head(), "\n \n")# first rows of features
print(diabetes.target.head()) # first rows of target
```

```
Shape: (768, 8)
   preg  plas  pres  skin  insu  mass   pedi  age
0     6   148    72    35     0  33.6  0.627   50
1     1    85    66    29     0  26.6  0.351   31
2     8   183    64     0     0  23.3  0.672   32
3     1    89    66    23    94  28.1  0.167   21
4     0   137    40    35   168  43.1  2.288   33
```

```
0    tested_positive
1    tested_negative
2    tested_positive
3    tested_negative
4    tested_positive
Name: class, dtype: category
Categories (2, object): ['tested_negative', 'tested_positive']
```

In [11]:
```
#train test spliting
test_size=0.2
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=test_size, random_state=42)
```

In [12]:
```
# Standardize features
scaler=StandardScaler()
Xtr= scaler.fit_transform(Xtr)
Xte= scaler.transform(Xte)
```

A fixed seed was added to this code to ensure the reproducibility of the analysis. This allowed for manual tuning of the hyperparameters to achieve better model training.

In [13]:
```
import random

seed = 42
torch.manual_seed(seed)
np.random.seed(seed)
random.seed(seed)

# Para GPU
torch.cuda.manual_seed(seed)
torch.cuda.manual_seed_all(seed)
```

```
# Tornar CUDA determinístico
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

For this model, the number of neurons per layer was increased from 64 to 100. This change was made after observing that a higher number of neurons per layer could lead to improved model performance.

In [14]:
```
class MLP(nn.Module):
    def __init__(self, input_size, output_size=1, dropout_prob=0.5):
        super(MLP, self).__init__()

        a = 100

        self.fc1 = nn.Linear(input_size, a)
        self.fc2 = nn.Linear(a, a)
        self.fc3 = nn.Linear(a, a)
        self.fc4 = nn.Linear(a, a)
        self.out = nn.Linear(a, 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
```

In [15]:
```
num_epochs=150
lr=0.0001
dropout=0.1
batch_size=128
```

This model was trained on the GPU and then transferred to the CPU for use with NumPy. Given the low number of parameters (i.e., the model's low complexity), the time required on the CPU was similar to that on the GPU.

In [16]:
```
# Model, Loss, Optimizer
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
#device = "cpu"  # force to use CPU
print(device)
```

```
cuda
```

In [17]:
```
Xtr = torch.tensor(Xtr, dtype=torch.float32).to(device)
ytr = torch.tensor(ytr, dtype=torch.float32).to(device)
Xte = torch.tensor(Xte, dtype=torch.float32).to(device)
yte = torch.tensor(yte, dtype=torch.float32).to("cpu")

# 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 [18]:
```
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)
```

The model was implemented as a fully connected neural network (MLP) with four hidden layers of 64 neurons each, using ReLU activation functions. The network takes the input features of the dataset and outputs a single value for regression (Diabetes Progression).

In [19]:
```
# Training Loop
import time
start_time = time.time()
for epoch in range(num_epochs):
```

```
    model.train()
    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()
        optimizer.step()

        epoch_loss += loss.item()

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

end_time = time.time()
print(f"Training time: {end_time - start_time:.2f} seconds")
```

```
Epoch [1/150], Loss: 0.3047
Epoch [2/150], Loss: 0.2965
Epoch [3/150], Loss: 0.2853
Epoch [4/150], Loss: 0.2748
Epoch [5/150], Loss: 0.2666
Epoch [6/150], Loss: 0.2612
Epoch [7/150], Loss: 0.2516
Epoch [8/150], Loss: 0.2430
Epoch [9/150], Loss: 0.2342
Epoch [10/150], Loss: 0.2262
Epoch [11/150], Loss: 0.2204
Epoch [12/150], Loss: 0.2118
Epoch [13/150], Loss: 0.2021
Epoch [14/150], Loss: 0.2008
Epoch [15/150], Loss: 0.1928
Epoch [16/150], Loss: 0.1910
Epoch [17/150], Loss: 0.1854
Epoch [18/150], Loss: 0.1833
Epoch [19/150], Loss: 0.1756
Epoch [20/150], Loss: 0.1753
Epoch [21/150], Loss: 0.1742
Epoch [22/150], Loss: 0.1699
Epoch [23/150], Loss: 0.1679
Epoch [24/150], Loss: 0.1621
Epoch [25/150], Loss: 0.1665
Epoch [26/150], Loss: 0.1668
Epoch [27/150], Loss: 0.1589
Epoch [28/150], Loss: 0.1595
Epoch [29/150], Loss: 0.1589
Epoch [30/150], Loss: 0.1579
Epoch [31/150], Loss: 0.1586
Epoch [32/150], Loss: 0.1594
Epoch [33/150], Loss: 0.1581
Epoch [34/150], Loss: 0.1524
Epoch [35/150], Loss: 0.1543
Epoch [36/150], Loss: 0.1496
Epoch [37/150], Loss: 0.1515
Epoch [38/150], Loss: 0.1545
Epoch [39/150], Loss: 0.1566
Epoch [40/150], Loss: 0.1530
Epoch [41/150], Loss: 0.1583
```

```
Epoch [42/150], Loss: 0.1496
Epoch [43/150], Loss: 0.1494
Epoch [44/150], Loss: 0.1520
Epoch [45/150], Loss: 0.1494
Epoch [46/150], Loss: 0.1473
Epoch [47/150], Loss: 0.1484
Epoch [48/150], Loss: 0.1517
Epoch [49/150], Loss: 0.1460
Epoch [50/150], Loss: 0.1500
Epoch [51/150], Loss: 0.1483
Epoch [52/150], Loss: 0.1478
Epoch [53/150], Loss: 0.1515
Epoch [54/150], Loss: 0.1521
Epoch [55/150], Loss: 0.1468
Epoch [56/150], Loss: 0.1484
Epoch [57/150], Loss: 0.1493
Epoch [58/150], Loss: 0.1454
Epoch [59/150], Loss: 0.1473
Epoch [60/150], Loss: 0.1443
Epoch [61/150], Loss: 0.1475
Epoch [62/150], Loss: 0.1423
Epoch [63/150], Loss: 0.1428
Epoch [64/150], Loss: 0.1438
Epoch [65/150], Loss: 0.1434
Epoch [66/150], Loss: 0.1442
Epoch [67/150], Loss: 0.1461
Epoch [68/150], Loss: 0.1430
Epoch [69/150], Loss: 0.1445
Epoch [70/150], Loss: 0.1458
Epoch [71/150], Loss: 0.1419
Epoch [72/150], Loss: 0.1426
Epoch [73/150], Loss: 0.1438
Epoch [74/150], Loss: 0.1436
Epoch [75/150], Loss: 0.1452
Epoch [76/150], Loss: 0.1464
Epoch [77/150], Loss: 0.1443
Epoch [78/150], Loss: 0.1388
Epoch [79/150], Loss: 0.1399
Epoch [80/150], Loss: 0.1416
Epoch [81/150], Loss: 0.1383
Epoch [82/150], Loss: 0.1432
```

```
Epoch [83/150], Loss: 0.1387
Epoch [84/150], Loss: 0.1417
Epoch [85/150], Loss: 0.1406
Epoch [86/150], Loss: 0.1373
Epoch [87/150], Loss: 0.1430
Epoch [88/150], Loss: 0.1435
Epoch [89/150], Loss: 0.1341
Epoch [90/150], Loss: 0.1422
Epoch [91/150], Loss: 0.1343
Epoch [92/150], Loss: 0.1393
Epoch [93/150], Loss: 0.1401
Epoch [94/150], Loss: 0.1424
Epoch [95/150], Loss: 0.1357
Epoch [96/150], Loss: 0.1378
Epoch [97/150], Loss: 0.1424
Epoch [98/150], Loss: 0.1363
Epoch [99/150], Loss: 0.1360
Epoch [100/150], Loss: 0.1367
Epoch [101/150], Loss: 0.1409
Epoch [102/150], Loss: 0.1403
Epoch [103/150], Loss: 0.1374
Epoch [104/150], Loss: 0.1308
Epoch [105/150], Loss: 0.1308
Epoch [106/150], Loss: 0.1387
Epoch [107/150], Loss: 0.1351
Epoch [108/150], Loss: 0.1383
Epoch [109/150], Loss: 0.1396
Epoch [110/150], Loss: 0.1402
Epoch [111/150], Loss: 0.1359
Epoch [112/150], Loss: 0.1393
Epoch [113/150], Loss: 0.1298
Epoch [114/150], Loss: 0.1328
Epoch [115/150], Loss: 0.1302
Epoch [116/150], Loss: 0.1371
Epoch [117/150], Loss: 0.1325
Epoch [118/150], Loss: 0.1379
Epoch [119/150], Loss: 0.1357
Epoch [120/150], Loss: 0.1342
Epoch [121/150], Loss: 0.1400
Epoch [122/150], Loss: 0.1363
Epoch [123/150], Loss: 0.1359
```

```
Epoch [124/150], Loss: 0.1322
Epoch [125/150], Loss: 0.1306
Epoch [126/150], Loss: 0.1338
Epoch [127/150], Loss: 0.1299
Epoch [128/150], Loss: 0.1254
Epoch [129/150], Loss: 0.1297
Epoch [130/150], Loss: 0.1311
Epoch [131/150], Loss: 0.1315
Epoch [132/150], Loss: 0.1311
Epoch [133/150], Loss: 0.1289
Epoch [134/150], Loss: 0.1358
Epoch [135/150], Loss: 0.1307
Epoch [136/150], Loss: 0.1301
Epoch [137/150], Loss: 0.1312
Epoch [138/150], Loss: 0.1297
Epoch [139/150], Loss: 0.1352
Epoch [140/150], Loss: 0.1322
Epoch [141/150], Loss: 0.1337
Epoch [142/150], Loss: 0.1260
Epoch [143/150], Loss: 0.1281
Epoch [144/150], Loss: 0.1324
Epoch [145/150], Loss: 0.1312
Epoch [146/150], Loss: 0.1259
Epoch [147/150], Loss: 0.1283
Epoch [148/150], Loss: 0.1314
Epoch [149/150], Loss: 0.1234
Epoch [150/150], Loss: 0.1310
Training time: 2.52 seconds
```

In [20]:

```
thr = 0.6 #threshold to tune

y_pred=model(Xte).cpu()
y_true = yte.detach().numpy()
y_pred_bin = (y_pred.detach().numpy() > thr).astype(int)  # binary predictions

# Accuracy
acc = accuracy_score(y_true, y_pred_bin)
print(f'ACC:{accuracy_score(y_true,y_pred_bin)}') #classification

cm = confusion_matrix(y_true, y_pred_bin)
print("Confusion Matrix:")
```

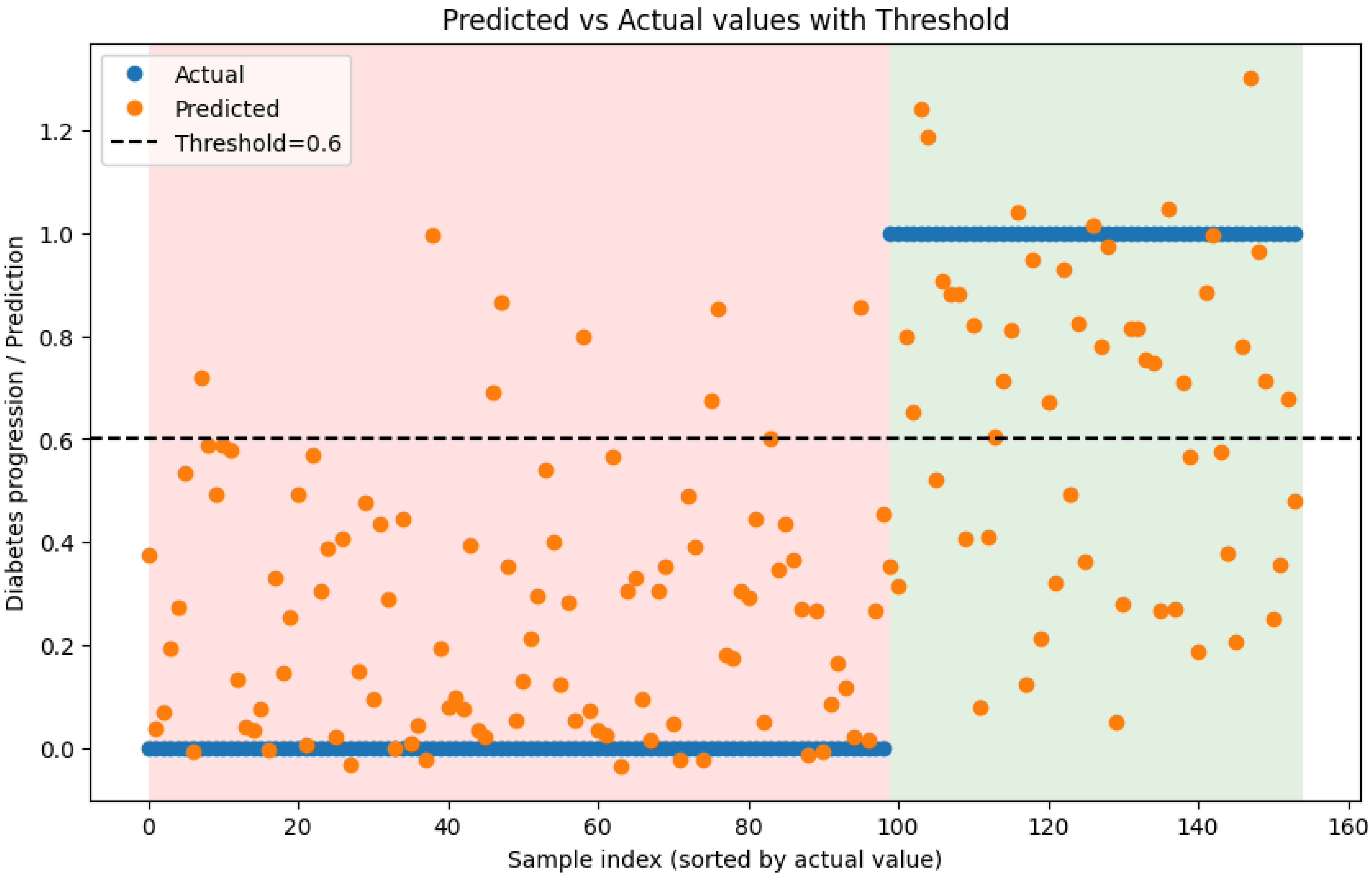
```
print(cm)

#print(f'ACC:{mean_squared_error(yte.detach().numpy(),y_pred.detach().numpy())}') #regression
```

```
ACC:0.7922077922077922
Confusion Matrix:
[[90  9]
 [23 32]]
```

After manually tuning the hyperparameters, the best configuration obtained was: num_epochs = 150, lr = 0.0001, dropout = 0.1, and batch_size = 128. Dropout with a probability of 0.1 was applied after each layer to mitigate overfitting. To validate the model, both accuracy and the confusion matrix were computed. An accuracy of 79.22% was achieved, which is comparable to other models. It was observed that the model performs better when predicting negative cases compared to positive cases. This behavior is likely due to the dataset being unbalanced, highlighting the importance of data quality for achieving reliable model performance. One possible approach to address this issue is to oversample the positive class so that both classes have similar representation during training.

In [21]:
```
# Convert y_pred to numpy and flatten
y_pred_np = y_pred.detach().numpy().flatten()
y_pred_bin_np = y_pred_bin
yte_np = yte.detach().numpy()

# Get indices that sort yte
sort_idx = np.argsort(yte_np)

# Sort yte and y_pred
yte_sorted = yte_np[sort_idx]
y_pred_sorted = y_pred_np[sort_idx]

# Determine the index where actual target switches from 0 to 1
frontier_idx = np.argmax(yte_sorted == 1)  # first occurrence of 1

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

# Shade negative region (yte = 0)
plt.axvspan(0, frontier_idx, facecolor='red', alpha=0.1)

# Shade positive region (yte = 1)
plt.axvspan(frontier_idx, len(yte_sorted), facecolor='green', alpha=0.1)
```

```
# Actual and predicted points
plt.plot(range(len(yte_sorted)), yte_sorted, label="Actual", marker="o", linestyle='')
plt.plot(range(len(y_pred_sorted)), y_pred_sorted, label="Predicted", marker="o", linestyle='')

# Threshold line
plt.axhline(y=thr, color='k', linestyle='--', label=f'Threshold={thr}')

plt.xlabel("Sample index (sorted by actual value)")
plt.ylabel("Diabetes progression / Prediction")
plt.title("Predicted vs Actual values with Threshold")
plt.legend()
plt.show()
```


Predicted vs Actual values with Threshold
Actual
Predicted
Threshold=0.6
Diabetes progression / Prediction
Sample index (sorted by actual value)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
20
40
60
80
100
120
140
160

```
In [ ]:
```