Intelligent Systems

Mechanical Engineering

Class Assignment 2

Author:

Diogo João Mendes Pereira (ISTID 87172)
diogojmpereira@tecnico.ulisboa.pt

2025/2026 – 1º Semester, P1

Contents

1 Dataset 1: Diabetes Dataset (Regression)

1.1 ANFIS

Continuing on assignment 1's work, the model was trained with ANFIS method. The training parameters were tuned with a grid search method culminating in the best values of:

n_iterations: 7 gd_epochs: 10 lr: 0.001

With these parameters the resulting model had an MSE of 2467.228515625.

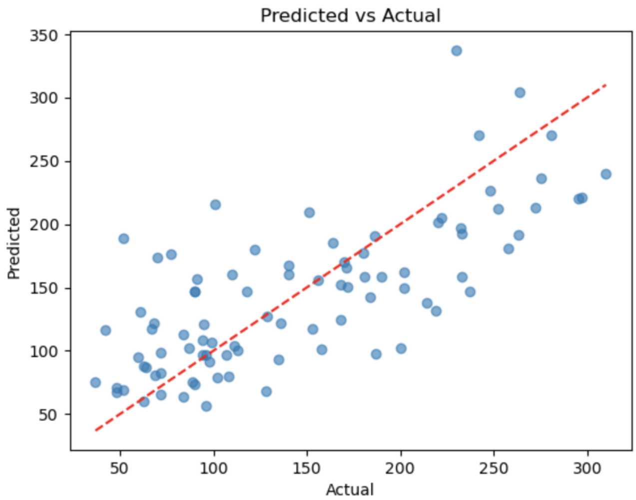


Figure 1: Confusion Matrix

1.2 MLP

For the MLP, the parameters where manually tuned:

num_epochs=100 lr=0.0003 dropout=0.2 batch_size=64

Testing different neural network layer configurations wasn't done.

The resulting model had a MSE of 4207.64013671875.

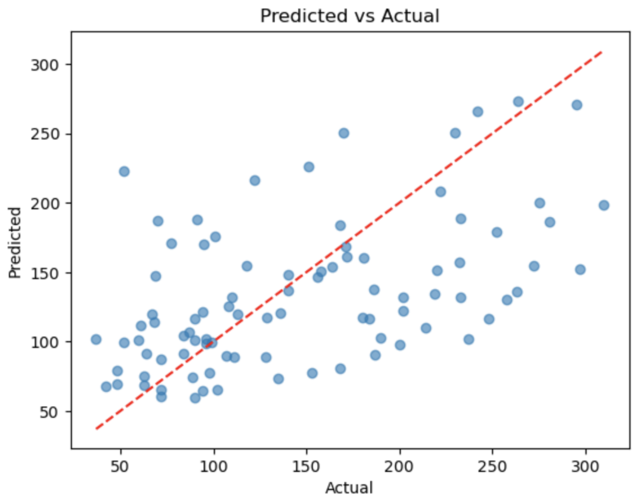


Figure 2: Confusion Matrix

2 Dataset 1: Diabetes Dataset (Regression)

2.1 ANFIS

Continuing on assignment 1's work, the model was trained with ANFIS method. The training parameters were tuned with a grid search method culminating in the best values of:

n_iterations: 3 gd_epochs: 10 lr: 0.01

With these parameters the resulting model had an ACC of 0.8206896551724138.

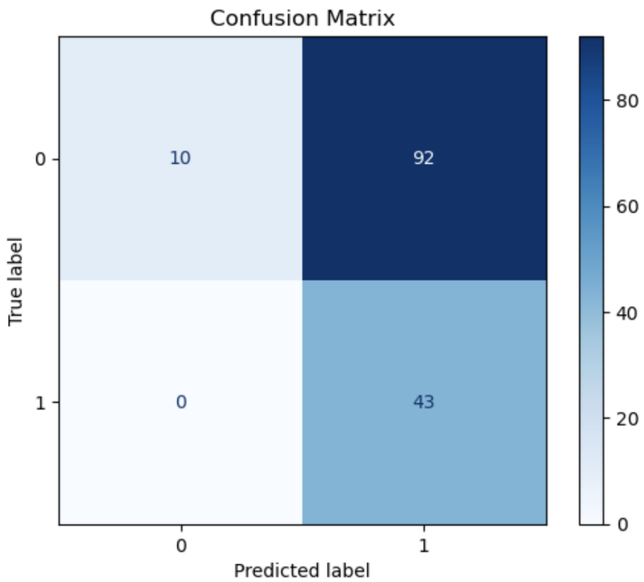


Figure 3: Confusion Matrix

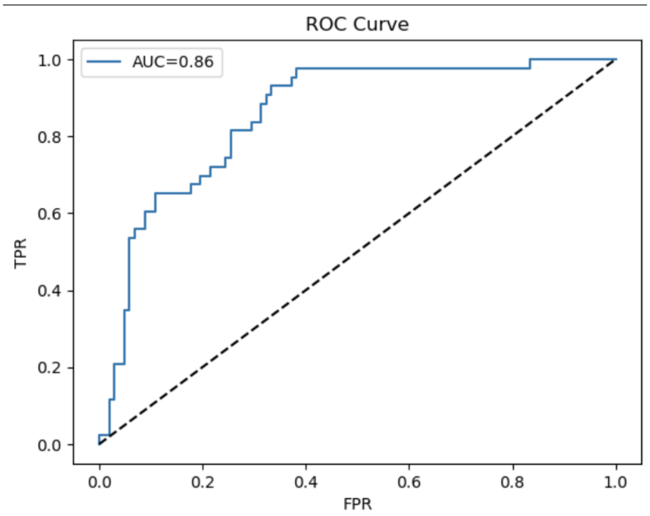


Figure 4: ROC

2.2 MLP

For the MLP, the parameters where manually tuned:
num_epochs=100 lr=0.0003 dropout=0.2 batch_size=64
Testing different neural network layer configurations wasn't done.
The resulting model had an ACC of 0.7931034482758621.

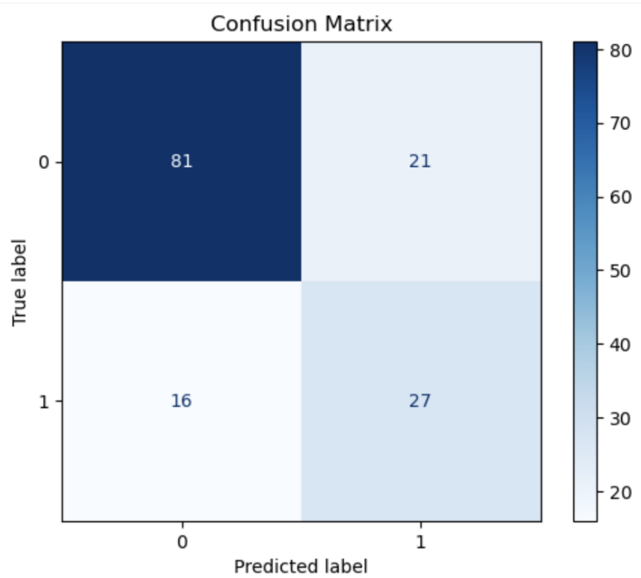


Figure 5: Confusion Matrix

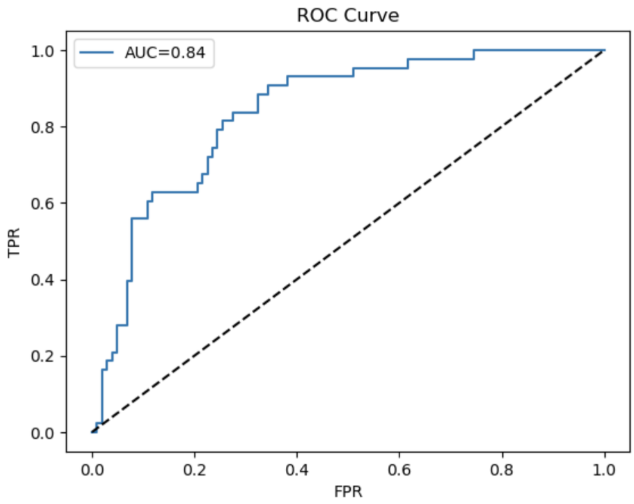


Figure 6: ROC

3 GitHub Repo

github.com/Pereira98/SI_Individual_87172.git

[REG] TSK_pytorch

October 3, 2025

[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

import copy
import itertools
```

[2]:
```
# CHOOSE DATASET

# Regression dataset
data = datasets.load_diabetes(as_frame=True)

X = data.data.values
y = data.target.values
X.shape
```

[2]: (442, 10)

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

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

[5]:
```
# Number of clusters
n_clusters = 2
m=1.5

# 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,
)
```

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

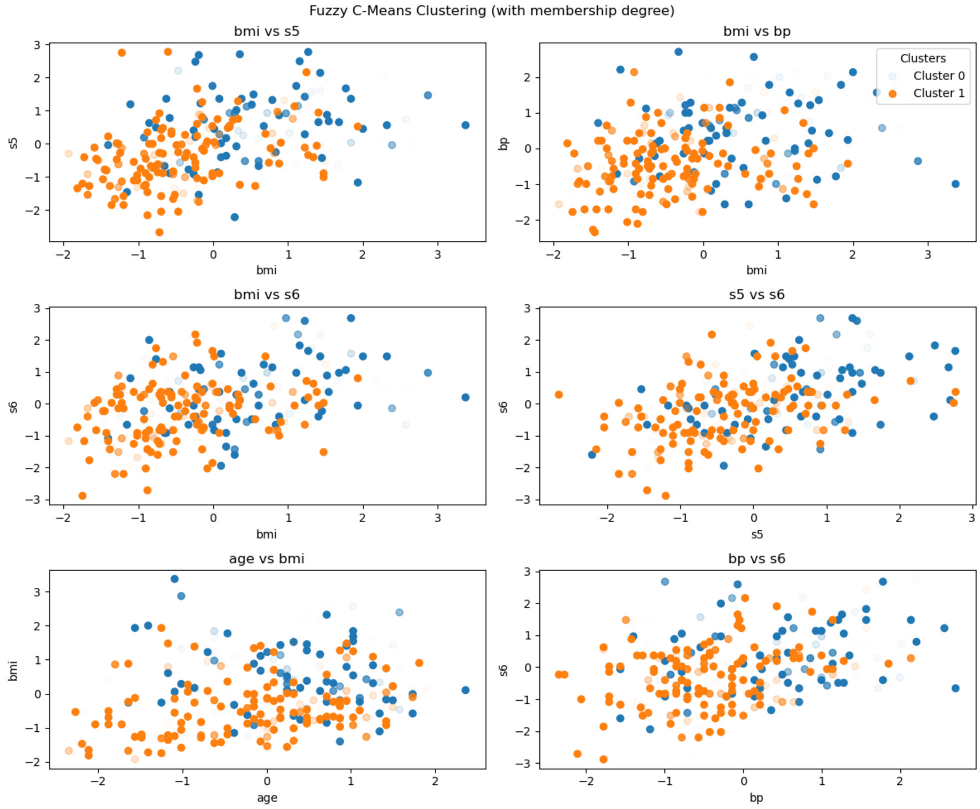
[6]: (2, 11)

[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)
```

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

# Diabetes feature names (sklearn's load_diabetes ordering)
feature_names = [
    "age",  # 0
    "sex",  # 1
    "bmi",  # 2
    "bp",   # 3
    "s1",   # 4 (tc)
    "s2",   # 5 (ldl)
    "s3",   # 6 (hdl)
    "s4",   # 7 (tch)
    "s5",   # 8 (ltg)
    "s6"    # 9 (glu)
```

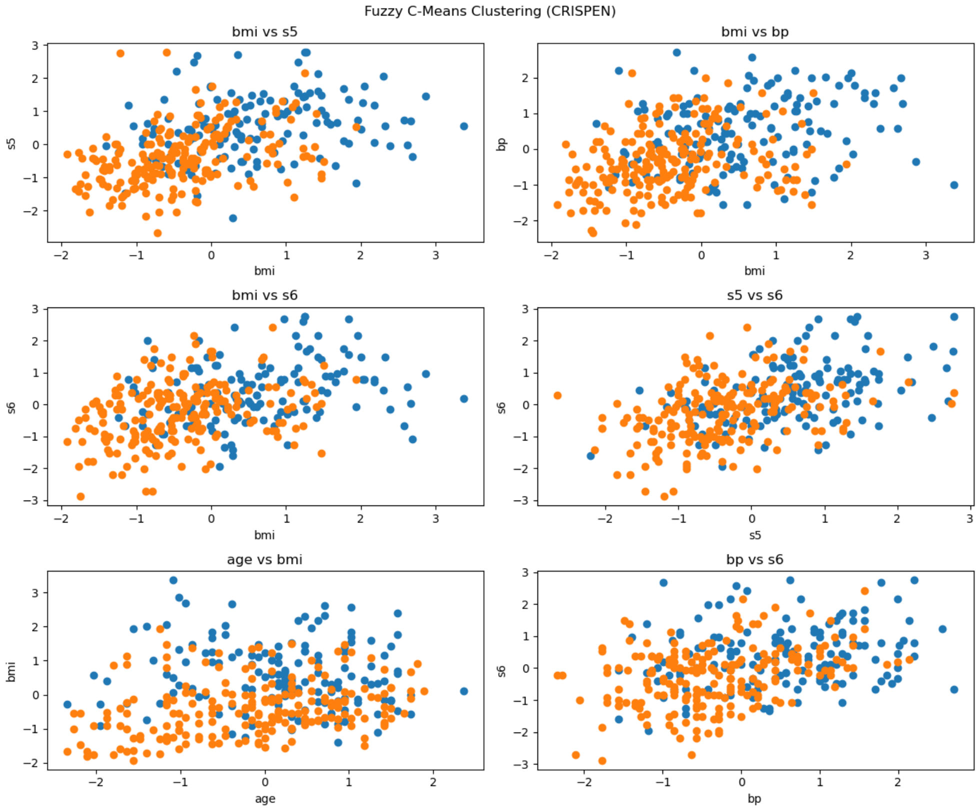
```python
]

# Choose 6 feature pairs (indices in Xexp)
pairs = [
    (2, 8),  # bmi vs s5 (both strong predictors)
    (2, 3),  # bmi vs bp
    (2, 9),  # bmi vs glu
    (8, 9),  # s5 vs glu
    (0, 2),  # age vs bmi
    (3, 9)   # bp vs glu
]

fig, axes = plt.subplots(3, 2, figsize=(12, 10))

for ax, (i, j) in zip(axes.ravel(), pairs):
    # Plot 2 features with fuzzy membership
    for k in range(n_clusters):
        ax.scatter(
            Xexp[cluster_labels == k, i],   # Feature i
            Xexp[cluster_labels == k, j],   # Feature j
            alpha=u[k, :],                  # transparency ~ membership degree
            label=f'Cluster {k}'
        )
    ax.set_xlabel(feature_names[i])
    ax.set_ylabel(feature_names[j])
    ax.set_title(f"{feature_names[i]} vs {feature_names[j]}")

fig.suptitle("Fuzzy C-Means Clustering (with membership degree)")
axes[0, 1].legend(title="Clusters", loc="upper right")
plt.tight_layout()
plt.show()

fig.savefig("../Plots/Fuzzy C-Means Clustering (with membership degree) REG.
 ↪pdf", format="pdf", bbox_inches="tight")
```

```
Fuzzy partition coefficient (FPC): 0.9397236627124719
```



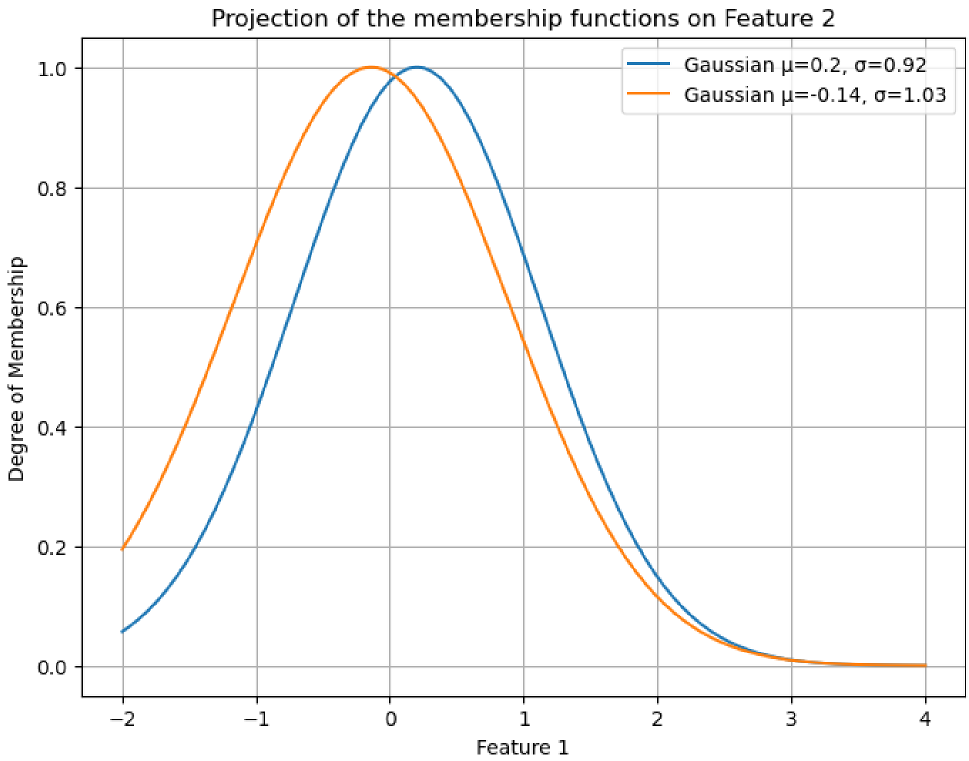
[9]:
```
fig, axes = plt.subplots(3, 2, figsize=(12, 10))

for ax, (i, j) in zip(axes.ravel(), pairs):
    # Plot 2 features with fuzzy membership
    for k in range(n_clusters):
        ax.scatter(
            Xexp[cluster_labels == k, i],
            Xexp[cluster_labels == k, j],
            label=f'Cluster {k}'
        )

    ax.set_xlabel(feature_names[i])
    ax.set_ylabel(feature_names[j])
    ax.set_title(f"{feature_names[i]} vs {feature_names[j]}")

fig.suptitle("Fuzzy C-Means Clustering (CRISPEN)")
fig.tight_layout()
fig.show()
```

```
fig.savefig("../Plots/Fuzzy C-Means Clustering (CRISPEN) REG.pdf",␣
↪format="pdf", bbox_inches="tight")
```



[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  ={np.
↪round(centers[j,feature],2)},  ={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()
```



[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 is the same as the max on dist)

        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)
```

```python
        # 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
```

[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)
```

[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()
```

[14]:
```
# ---------------------------
# 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
```

[15]:
```
# ---------------------------
# 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)
```

[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)
```

[17]:
```
param_grid = {
    "max_iters": [3, 5, 7],
    "gd_epochs": [10, 20, 50],
    "lr": [1e-2, 1e-3, 3e-4]
}

results = []
```

```python
for max_iters, gd_epochs, lr in itertools.product(
    param_grid["max_iters"],
    param_grid["gd_epochs"],
    param_grid["lr"]
):
    m = copy.deepcopy(model)
    train_hybrid_anfis(m, Xtr, ytr.reshape(-1,1),
                       max_iters=max_iters,
                       gd_epochs=gd_epochs,
                       lr=lr)

    # forward pass
    y_pred, _, _ = m(Xte)

    # compute accuracy
    rse = mean_squared_error(yte.detach().numpy(),y_pred.detach().numpy())

    # save result as dict
    results.append({
        "max_iters": max_iters,
        "gd_epochs": gd_epochs,
        "lr": lr,
        "rse": rse
    })

# Convert to DataFrame
df = pandas.DataFrame(results)

# Sort by accuracy (descending)
df = df.sort_values(by="rse", ascending=True).reset_index(drop=True)

print(df)
```

```
    max_iters  gd_epochs      lr          rse
0           7         10  0.0010  2467.229248
1           5         50  0.0003  2467.694092
2           3         20  0.0010  2471.744873
3           5         10  0.0010  2475.592773
4           7         20  0.0003  2482.729980
5           3         50  0.0003  2482.941162
6           3         10  0.0010  2491.065674
7           5         20  0.0010  2492.151367
8           5         20  0.0003  2492.248535
9           7         10  0.0003  2499.436035
10          3         20  0.0003  2504.704102
11          5         10  0.0003  2509.327881
12          7         50  0.0003  2510.137207
13          3         10  0.0003  2518.572754
```

14	7	20	0.0010	2573.720459
15	3	50	0.0010	2592.591309
16	3	10	0.0100	2693.129150
17	5	50	0.0010	2698.234619
18	5	10	0.0100	2734.992920
19	7	50	0.0010	2741.527344
20	3	20	0.0100	2760.484131
21	3	50	0.0100	2773.041260
22	7	10	0.0100	2788.303955
23	5	50	0.0100	2814.766602
24	7	50	0.0100	2845.574707
25	5	20	0.0100	2856.106445
26	7	20	0.0100	2857.052002

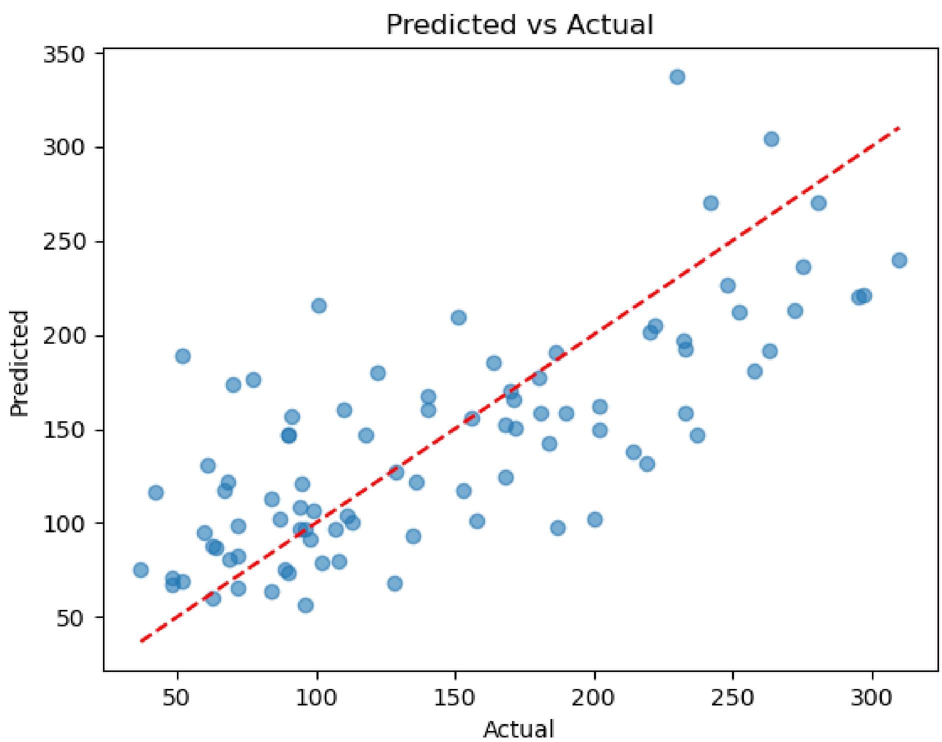
[18]:
```
# Training with LS:
model_ls = copy.deepcopy(model)
train_ls(model_ls, Xtr, ytr.reshape(-1,1))

# Training with GD:
#model_gd = copy.deepcopy(model)
#train_gd(model_gd, Xtr, ytr.reshape(-1,1), epochs=100, lr=1e-3)

# Training with Hybrid (Classic ANFIS):
model_h = copy.deepcopy(model)
train_hybrid_anfis(model_h, Xtr, ytr.reshape(-1,1), max_iters=7, gd_epochs=10,␣
↪lr=1e-3)

# Training with Alternative Hybrid (LS + GD):
#model_ah = copy.deepcopy(model)
#train_hybrid(model_ah, Xtr, ytr.reshape(-1,1), epochs=100, lr=1e-4)
```

[19]:
```
y_pred, _, _=model_ls(Xte)
#performance metric for regression
print(f'LS MSE:{mean_squared_error(yte.detach().numpy(),y_pred.detach().
↪numpy())}') #regression
#y_pred, _, _=model_gd(Xte)
#performance metric for regression
#print(f'GD MSE:{mean_squared_error(yte.detach().numpy(),y_pred.detach().
↪numpy())}') #regression
y_pred, _, _=model_h(Xte)
#performance metric for regression
print(f'H  MSE:{mean_squared_error(yte.detach().numpy(),y_pred.detach().
↪numpy())}') #regression
#y_pred, _, _=model_ah(Xte)
#performance metric for regression
```

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

```
LS MSE:2534.1474609375
H  MSE:2467.228515625
```

[20]:
```
plt.scatter(yte, y_pred.detach().numpy(), alpha=0.6)
plt.plot([yte.min(), yte.max()], [yte.min(), yte.max()], 'r--')
plt.xlabel("Actual")
plt.ylabel("Predicted")
plt.title("Predicted vs Actual")
plt.show()
plt.savefig("../Plots/PredictedActualANFIS.pdf", format="pdf",␣
 ↪bbox_inches="tight")
```




```
<Figure size 640x480 with 0 Axes>
```

[ ]:

[REG] simple_mlp_pytorch

October 3, 2025

[1]:
```python
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
```

[2]:
```python
# CHOOSE DATASET

# Regression dataset
data = datasets.load_diabetes(as_frame=True)

X = data.data.values
y = data.target.values
X.shape
```

[2]:
```
(442, 10)
```

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

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

[5]:
```python
class MLP(nn.Module):
    def __init__(self, input_size, output_size=1, dropout_prob=0.5):
```

```python
        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
```

[6]:
```python
num_epochs=100
lr=0.0003
dropout=0.2
batch_size=64
```

[7]:
```python
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)
```

[8]:
```python
# Model, Loss, Optimizer
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = MLP(input_size=Xtr.shape[1], dropout_prob=dropout).to(device)
```

```
criterion = nn.MSELoss() #for regression
optimizer = optim.Adam(model.parameters(), lr=lr)
```

[9]:
```
# Training loop
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}")
```

```
Epoch [1/100], Loss: 29963.8867
Epoch [2/100], Loss: 29899.7767
Epoch [3/100], Loss: 29866.7109
Epoch [4/100], Loss: 29854.5889
Epoch [5/100], Loss: 28869.4707
Epoch [6/100], Loss: 29600.7695
Epoch [7/100], Loss: 29236.1309
Epoch [8/100], Loss: 29280.0104
Epoch [9/100], Loss: 29287.0186
Epoch [10/100], Loss: 29358.9215
Epoch [11/100], Loss: 29781.8675
Epoch [12/100], Loss: 28431.5807
Epoch [13/100], Loss: 28741.9020
Epoch [14/100], Loss: 28719.7448
Epoch [15/100], Loss: 27394.0889
Epoch [16/100], Loss: 26717.4395
Epoch [17/100], Loss: 25787.5550
Epoch [18/100], Loss: 23656.4255
Epoch [19/100], Loss: 22760.5400
Epoch [20/100], Loss: 20806.1188
Epoch [21/100], Loss: 18125.7946
Epoch [22/100], Loss: 15652.5057
Epoch [23/100], Loss: 12824.7303
Epoch [24/100], Loss: 10519.0179
```

```
Epoch [25/100], Loss: 8550.2349
Epoch [26/100], Loss: 7798.2155
Epoch [27/100], Loss: 6271.0618
Epoch [28/100], Loss: 5640.2359
Epoch [29/100], Loss: 5980.7294
Epoch [30/100], Loss: 5330.4427
Epoch [31/100], Loss: 5788.4426
Epoch [32/100], Loss: 5583.0360
Epoch [33/100], Loss: 5159.8354
Epoch [34/100], Loss: 5094.7147
Epoch [35/100], Loss: 4921.0780
Epoch [36/100], Loss: 5162.0347
Epoch [37/100], Loss: 4948.4863
Epoch [38/100], Loss: 4348.9304
Epoch [39/100], Loss: 4583.8053
Epoch [40/100], Loss: 4667.6394
Epoch [41/100], Loss: 4131.9939
Epoch [42/100], Loss: 4062.7534
Epoch [43/100], Loss: 4765.8565
Epoch [44/100], Loss: 4044.8011
Epoch [45/100], Loss: 4279.3233
Epoch [46/100], Loss: 4078.8448
Epoch [47/100], Loss: 4385.0946
Epoch [48/100], Loss: 4196.5803
Epoch [49/100], Loss: 4120.9447
Epoch [50/100], Loss: 4283.8970
Epoch [51/100], Loss: 4110.0351
Epoch [52/100], Loss: 4619.4892
Epoch [53/100], Loss: 3971.0571
Epoch [54/100], Loss: 4147.9974
Epoch [55/100], Loss: 4453.0304
Epoch [56/100], Loss: 4425.9319
Epoch [57/100], Loss: 4184.6817
Epoch [58/100], Loss: 3871.1070
Epoch [59/100], Loss: 3760.2430
Epoch [60/100], Loss: 4075.9898
Epoch [61/100], Loss: 3560.8109
Epoch [62/100], Loss: 4148.9313
Epoch [63/100], Loss: 3939.6185
Epoch [64/100], Loss: 3934.4459
Epoch [65/100], Loss: 3814.6288
Epoch [66/100], Loss: 3879.1348
Epoch [67/100], Loss: 4088.5636
Epoch [68/100], Loss: 3920.8263
Epoch [69/100], Loss: 4078.7714
Epoch [70/100], Loss: 3840.1358
Epoch [71/100], Loss: 3947.9802
Epoch [72/100], Loss: 4014.3708
```

```
Epoch [73/100], Loss: 4327.6577
Epoch [74/100], Loss: 4017.3526
Epoch [75/100], Loss: 3856.8468
Epoch [76/100], Loss: 4354.9016
Epoch [77/100], Loss: 3879.2114
Epoch [78/100], Loss: 3884.2566
Epoch [79/100], Loss: 3813.7034
Epoch [80/100], Loss: 3949.5700
Epoch [81/100], Loss: 4042.1320
Epoch [82/100], Loss: 3726.5214
Epoch [83/100], Loss: 3672.1545
Epoch [84/100], Loss: 3564.3408
Epoch [85/100], Loss: 3697.4874
Epoch [86/100], Loss: 3968.1486
Epoch [87/100], Loss: 3982.2527
Epoch [88/100], Loss: 3832.3629
Epoch [89/100], Loss: 3587.4059
Epoch [90/100], Loss: 3599.8098
Epoch [91/100], Loss: 3808.5240
Epoch [92/100], Loss: 3651.7393
Epoch [93/100], Loss: 3530.6734
Epoch [94/100], Loss: 3784.5076
Epoch [95/100], Loss: 3566.1061
Epoch [96/100], Loss: 3871.3206
Epoch [97/100], Loss: 3977.1104
Epoch [98/100], Loss: 3779.8511
Epoch [99/100], Loss: 3656.1186
Epoch [100/100], Loss: 3481.5117
```

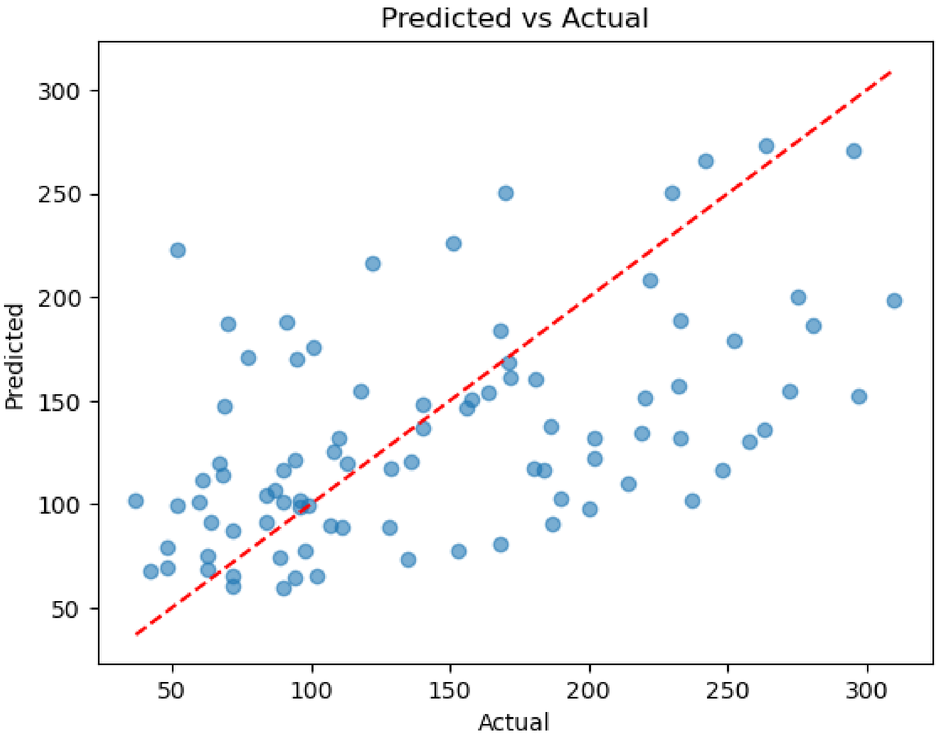
[10]:

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

```
MSE:4207.64013671875
```

[11]:

```python
plt.scatter(yte, y_pred.detach().numpy(), alpha=0.6)
plt.plot([yte.min(), yte.max()], [yte.min(), yte.max()], 'r--')
plt.xlabel("Actual")
plt.ylabel("Predicted")
plt.title("Predicted vs Actual")
plt.show()
plt.savefig("../Plots/PredictedActualMLP.pdf", format="pdf",␣
 ↪bbox_inches="tight")
```



<Figure size 640x480 with 0 Axes>

[ ]:

[CLS] TSK_pytorch

October 3, 2025

[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,confusion_matrix,ConfusionMatrixDis
import skfuzzy as fuzz
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
import pandas

import copy
import itertools
```

[2]:
```
# CHOOSE DATASET

# Binary classification dataset
data = datasets.fetch_openml(name="diabetes",version=1, as_frame=True)

X = data.data.values
y = data.target.values
X.shape

# Keep as DataFrame for named-column ops
df = data.data.copy()
y = np.where(data.target.values == "tested_positive", 1, 0).astype(np.float32)

# indices of features with invalid zeros
invalid_idx = [1, 2, 3, 4, 5, 7]

# count zeros per feature
zero_counts = (X[:, invalid_idx] == 0).sum(axis=0)
rows_with_zero = (X[:, invalid_idx] == 0).any(axis=1).sum()

print("Zeros per feature:\n", zero_counts)
```

```
print(f"Rows with  1 zero: {rows_with_zero} / {len(df)}")

(len(X), zero_counts, rows_with_zero)

# Drop columns 3 and 4 (0-based indexing)
X = np.delete(X, [3, 4], axis=1)

# Keep only rows where Glucose, BloodPressure, BMI are non-zero
mask = (X[:, [1, 2, 3, 4]] != 0).all(axis=1)
X = X[mask]
y = y[mask]
```

```
Zeros per feature:
 [  5  35 227 374  11   0]
Rows with  1 zero: 376 / 768
```

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

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

[5]:
```
 # Number of clusters
n_clusters = 2
m=1.5

# 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,
)
```

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

[6]: (2, 7)

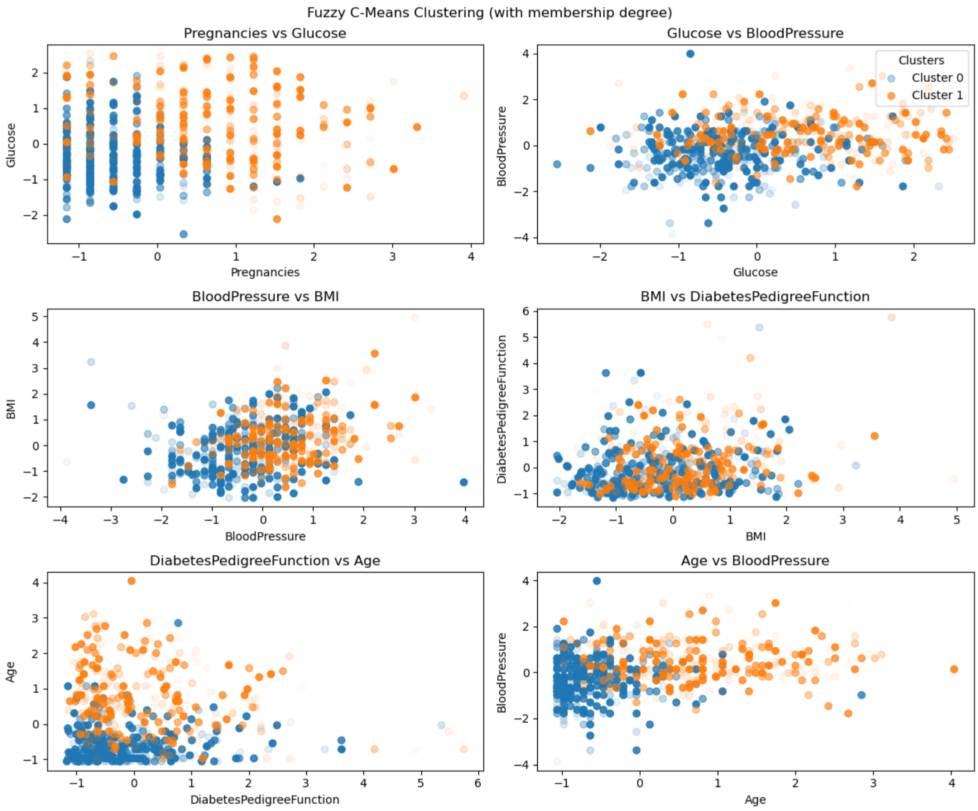
[7]:
```
# Compute sigma (spread) for each cluster
sigmas = []
```

```python
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)
```

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

feature_names = [
    "Pregnancies", "Glucose", "BloodPressure",
    "BMI", "DiabetesPedigreeFunction", "Age"
]

# Choose 4 feature pairs (indices in Xexp)
pairs = [(0, 1),  # Pregnancies vs Glucose
         (1, 2),  # Glucose vs BloodPressure
         (2, 3),  # BloodPressure vs BMI
         (3, 4),  # BMI vs Pedigree
         (4, 5),  # Pedigree vs Age
         (5, 2)]  # Age vs Glucose

fig, axes = plt.subplots(3, 2, figsize=(12, 10))

for ax, (i, j) in zip(axes.ravel(), pairs):
    # Plot 2 features with fuzzy membership
    for k in range(n_clusters):
        ax.scatter(
            Xexp[cluster_labels == k, i],             # Feature 1
            Xexp[cluster_labels == k, j],             # Feature 2
            alpha=u[k, :],           # transparency ~ membership
            label=f'Cluster {k}'
        )

    ax.set_xlabel(feature_names[i])
    ax.set_ylabel(feature_names[j])
    ax.set_title(f"{feature_names[i]} vs {feature_names[j]}")

fig.suptitle("Fuzzy C-Means Clustering (with membership degree)")
axes[0,1].legend(title="Clusters", loc="upper right")
plt.tight_layout()
plt.show()
```

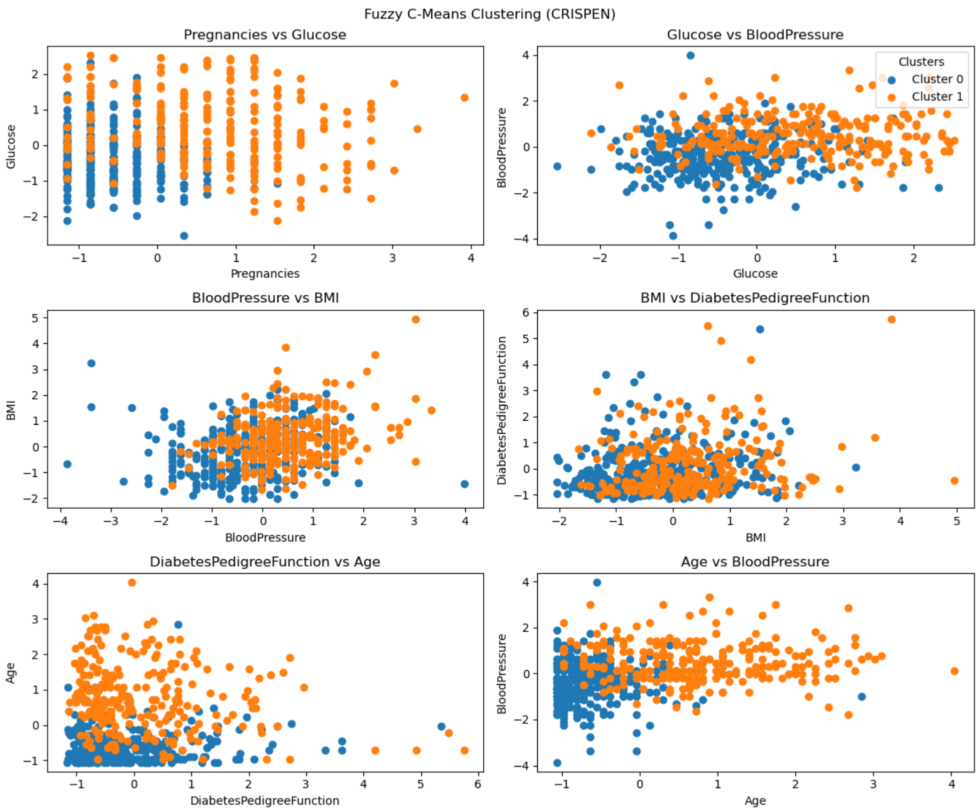
```
fig.savefig("../Plots/Fuzzy C-Means Clustering (with membership degree) CLF.
↪pdf", format="pdf", bbox_inches="tight")
```

```
Fuzzy partition coefficient (FPC): 0.7231777903765536
```



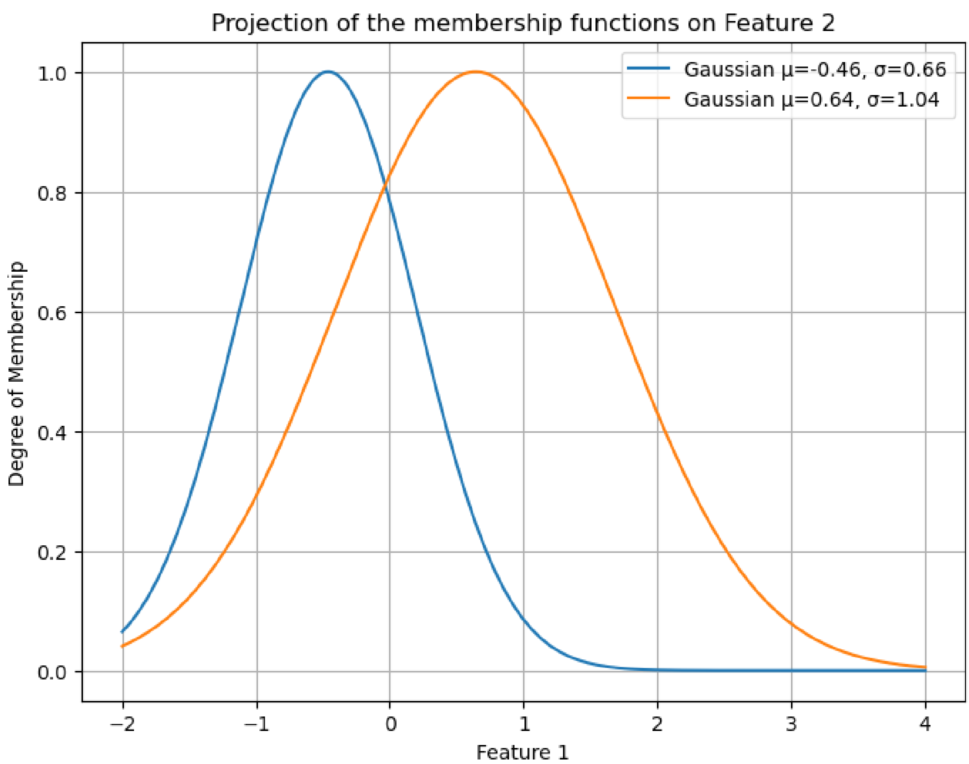

```
[9]: fig, axes = plt.subplots(3, 2, figsize=(12, 10))

     for ax, (i, j) in zip(axes.ravel(), pairs):
         # Plot 2 features with fuzzy membership
         for k in range(n_clusters):
             ax.scatter(
                 Xexp[cluster_labels == k, i],
                 Xexp[cluster_labels == k, j],
                 label=f'Cluster {k}'
             )

         ax.set_xlabel(feature_names[i])
         ax.set_ylabel(feature_names[j])
```

```
    ax.set_title(f"{feature_names[i]} vs {feature_names[j]}")

fig.suptitle("Fuzzy C-Means Clustering (CRISPEN)")
axes[0,1].legend(title="Clusters", loc="upper right")
fig.tight_layout()
fig.show()

fig.savefig("../Plots/Fuzzy C-Means Clustering (CRISPEN) CLF.pdf",␣
 ↪format="pdf", bbox_inches="tight")
```



[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  ={np.
 ↪round(centers[j,feature],2)},  ={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()
```



[11]:
```
# ---------------------------
# Gaussian Membership Function
# ---------------------------
```

```python
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 is the same as the max on dist)

        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
```

[12]:
```
# ---------------------------
# 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)
```

[13]:
```
# ---------------------------
# Gradient Descent Training
# ---------------------------
def train_gd(model, X, y, epochs=100, lr=1e-3):
    optimizer = optim.Adam(model.parameters(), lr=lr)
```

```
    criterion = nn.MSELoss()
    for _ in range(epochs):
        optimizer.zero_grad()
        y_pred, _, _ = model(X)
        loss = criterion(y_pred, y)
        #print(loss)
        loss.backward()
        optimizer.step()
```

[14]:
```
# ---------------------------
# 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
```

[15]:
```
# ---------------------------
# 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)
```

[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)
```

[17]:
```
param_grid = {
    "max_iters": [3, 5, 7],
```

```
    "gd_epochs": [10, 20, 50],
    "lr": [1e-2, 1e-3, 3e-4]
}

results = []

for max_iters, gd_epochs, lr in itertools.product(
    param_grid["max_iters"],
    param_grid["gd_epochs"],
    param_grid["lr"]
):
    m = copy.deepcopy(model)
    train_hybrid_anfis(m, Xtr, ytr.reshape(-1,1),
                       max_iters=max_iters,
                       gd_epochs=gd_epochs,
                       lr=lr)

    # forward pass
    y_pred, _, _ = m(Xte)

    # compute accuracy
    acc = accuracy_score(
        yte.detach().cpu().numpy(),
        (y_pred.detach().cpu().numpy() > 0.5)
    )

    # save result as dict
    results.append({
        "max_iters": max_iters,
        "gd_epochs": gd_epochs,
        "lr": lr,
        "accuracy": acc
    })

# Convert to DataFrame
df = pandas.DataFrame(results)

# Sort by accuracy (descending)
df = df.sort_values(by="accuracy", ascending=False).reset_index(drop=True)

print(df)
```

```
   max_iters  gd_epochs      lr  accuracy
0          3         10  0.0100  0.820690
1          5         50  0.0010  0.820690
2          3         50  0.0100  0.806897
3          7         20  0.0100  0.806897
4          7         50  0.0010  0.806897
```

5	5	50	0.0100	0.806897
6	3	50	0.0003	0.800000
7	3	10	0.0003	0.800000
8	3	20	0.0003	0.800000
9	5	10	0.0100	0.800000
10	3	10	0.0010	0.800000
11	7	10	0.0003	0.800000
12	7	20	0.0003	0.800000
13	5	10	0.0003	0.800000
14	5	10	0.0010	0.800000
15	5	20	0.0003	0.800000
16	5	20	0.0100	0.800000
17	3	20	0.0010	0.793103
18	3	20	0.0100	0.793103
19	7	10	0.0100	0.793103
20	3	50	0.0010	0.786207
21	5	50	0.0003	0.786207
22	5	20	0.0010	0.786207
23	7	20	0.0010	0.786207
24	7	10	0.0010	0.786207
25	7	50	0.0100	0.786207
26	7	50	0.0003	0.786207

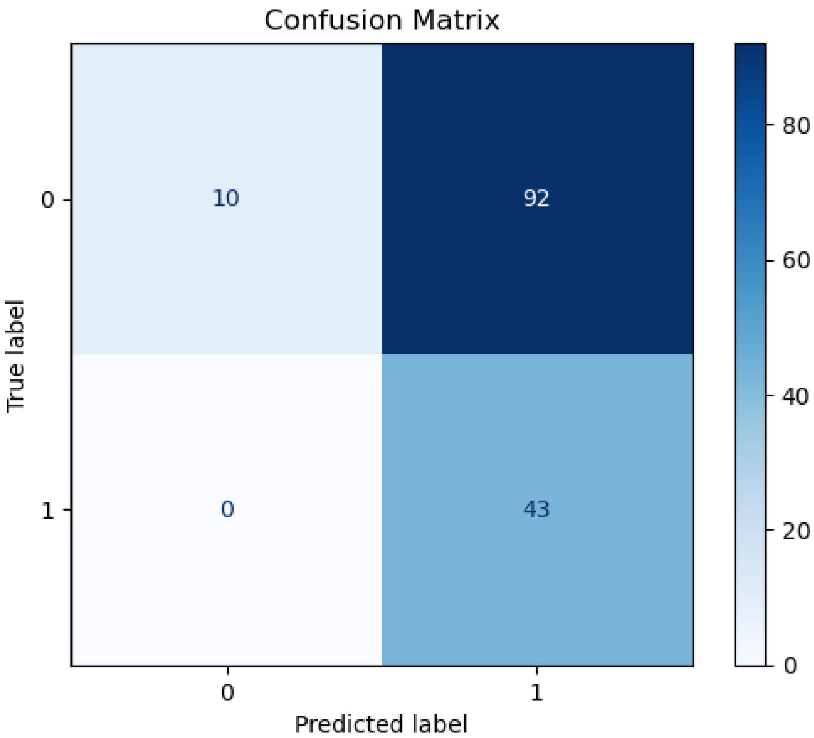
[18]:
```
# Training with LS:
model_ls = copy.deepcopy(model)
train_ls(model_ls, Xtr, ytr.reshape(-1,1))

# Training with GD:
model_gd = copy.deepcopy(model)
train_gd(model_gd, Xtr, ytr.reshape(-1,1), epochs=100, lr=1e-3)

# Training with Hybrid (Classic ANFIS):
model_h = copy.deepcopy(model)
train_hybrid_anfis(model_h, Xtr, ytr.reshape(-1,1), max_iters=3, gd_epochs=10,␣
 ↪lr=1e-2)

# Training with Alternative Hybrid (LS + GD):
model_ah = copy.deepcopy(model)
train_hybrid(model_ah, Xtr, ytr.reshape(-1,1), epochs=100, lr=1e-4)
```

[19]:
```
y_pred, _, _=model_ls(Xte)
#performance metric for classification
print(f'LS ACC:{accuracy_score(yte.detach().numpy(),y_pred.detach().numpy()>0.
 ↪5)}') #classification
#y_pred, _, _=model_gd(Xte)
#performance metric for classification
```

```
#print(f'GD ACC:{accuracy_score(yte.detach().numpy(),y_pred.detach().numpy()>0.
↪5)}') #classification
y_pred, _, _=model_h(Xte)
#performance metric for classification
print(f'H  ACC:{accuracy_score(yte.detach().numpy(),y_pred.detach().numpy()>0.
↪5)}') #classification
#y_pred, _, _=model_ah(Xte)
#performance metric for classification
#print(f'AH ACC:{accuracy_score(yte.detach().numpy(),y_pred.detach().numpy()>0.
↪5)}') #classification
```

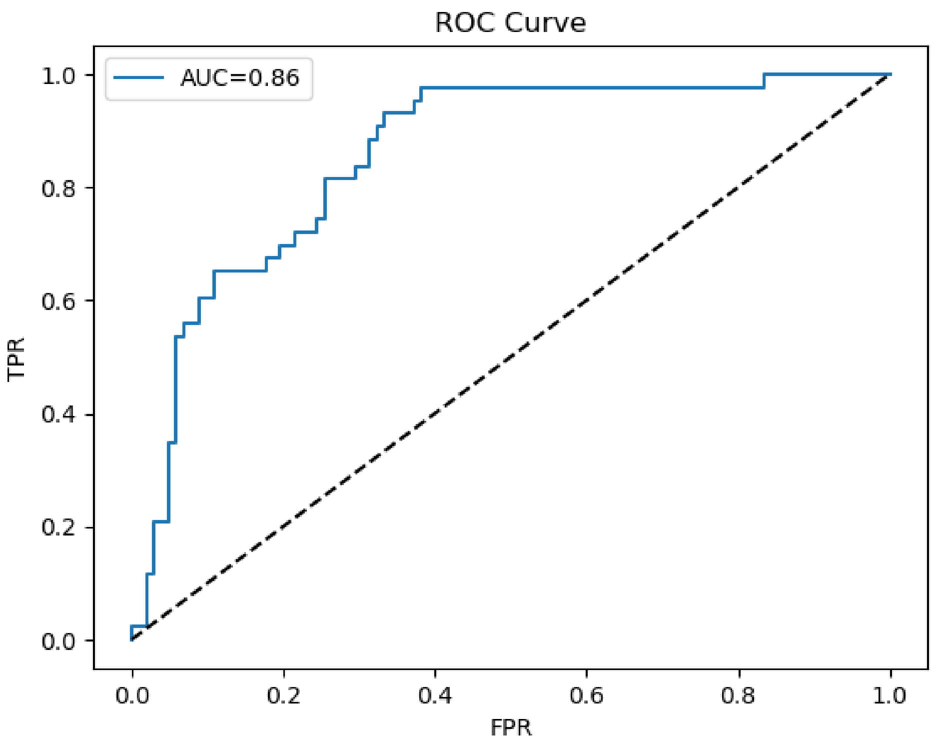
```
LS ACC:0.8
H  ACC:0.8206896551724138
```

[22]:
```
# ---- forward pass on the test set ----
model_h.eval()
with torch.no_grad():
    Xte_t = Xte.float()
    logits, *_ = model_h(Xte_t)
    y_proba = torch.sigmoid(logits).cpu().numpy().reshape(-1)  # probabilities␣
↪in [0,1]

# ---- threshold at 0.5 to get class labels ----
y_pred = (y_proba > 0.5).astype(int)


# ---- confusion matrix ----
cm = confusion_matrix(yte, y_pred)
ConfusionMatrixDisplay(cm).plot(cmap="Blues")
plt.title("Confusion Matrix")
plt.show()
plt.savefig("../Plots/ConfusionMatrixANFIS.pdf", format="pdf",␣
↪bbox_inches="tight")
```



<Figure size 640x480 with 0 Axes>

[21]:
```
fpr, tpr, _ = roc_curve(yte, y_proba)
roc_auc = auc(fpr, tpr)
plt.plot(fpr, tpr, label=f"AUC={roc_auc:.2f}")
plt.plot([0,1],[0,1],'k--')
plt.xlabel("FPR"); plt.ylabel("TPR"); plt.title("ROC Curve"); plt.legend(); plt.
    ↪show()
plt.savefig("../Plots/RoCANFIS.pdf", format="pdf", bbox_inches="tight")
```




```
<Figure size 640x480 with 0 Axes>
```

```
[ ]:
```

[CLS] simple_mlp_pytorch

October 3, 2025

[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,confusion_matrix,ConfusionMatrixDis
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
```

[2]:
```
# CHOOSE DATASET

# Binary classification dataset
data = datasets.fetch_openml(name="diabetes",version=1, as_frame=True)

X = data.data.values
y = data.target.values
X.shape

# Keep as DataFrame for named-column ops
df = data.data.copy()
y = np.where(data.target.values == "tested_positive", 1, 0).astype(np.float32)

# indices of features with invalid zeros
invalid_idx = [1, 2, 3, 4, 5, 7]

# count zeros per feature
zero_counts = (X[:, invalid_idx] == 0).sum(axis=0)
rows_with_zero = (X[:, invalid_idx] == 0).any(axis=1).sum()

print("Zeros per feature:\n", zero_counts)
print(f"Rows with  1 zero: {rows_with_zero} / {len(df)}")
```

```python
(len(X), zero_counts, rows_with_zero)

# Drop columns 3 and 4 (0-based indexing)
X = np.delete(X, [3, 4], axis=1)

# Keep only rows where Glucose, BloodPressure, BMI are non-zero
mask = (X[:, [1, 2, 3, 4]] != 0).all(axis=1)
X = X[mask]
y = y[mask]
```

```
Zeros per feature:
 [  5  35 227 374  11   0]
Rows with  1 zero: 376 / 768
```

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

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

[5]:
```python
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)
```

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

[6]:
```python
num_epochs=100
lr=0.0003
dropout=0.2
batch_size=64
```

[7]:
```python
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)
```

[8]:
```python
# Model, Loss, Optimizer
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = MLP(input_size=Xtr.shape[1], dropout_prob=dropout).to(device)
criterion = nn.BCEWithLogitsLoss()  # for binary classification
optimizer = optim.Adam(model.parameters(), lr=lr)
```

[9]:
```python
# Training loop
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)
```

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

```
Epoch [1/100], Loss: 0.6730
Epoch [2/100], Loss: 0.6669
Epoch [3/100], Loss: 0.6624
Epoch [4/100], Loss: 0.6630
Epoch [5/100], Loss: 0.6381
Epoch [6/100], Loss: 0.6343
Epoch [7/100], Loss: 0.6055
Epoch [8/100], Loss: 0.6017
Epoch [9/100], Loss: 0.5675
Epoch [10/100], Loss: 0.5774
Epoch [11/100], Loss: 0.5246
Epoch [12/100], Loss: 0.5239
Epoch [13/100], Loss: 0.5293
Epoch [14/100], Loss: 0.4761
Epoch [15/100], Loss: 0.4889
Epoch [16/100], Loss: 0.5030
Epoch [17/100], Loss: 0.4604
Epoch [18/100], Loss: 0.4668
Epoch [19/100], Loss: 0.4959
Epoch [20/100], Loss: 0.4499
Epoch [21/100], Loss: 0.4668
Epoch [22/100], Loss: 0.4624
Epoch [23/100], Loss: 0.4320
Epoch [24/100], Loss: 0.4519
Epoch [25/100], Loss: 0.4800
Epoch [26/100], Loss: 0.4419
Epoch [27/100], Loss: 0.5479
Epoch [28/100], Loss: 0.5238
Epoch [29/100], Loss: 0.4859
Epoch [30/100], Loss: 0.4435
Epoch [31/100], Loss: 0.4712
Epoch [32/100], Loss: 0.4592
Epoch [33/100], Loss: 0.4637
Epoch [34/100], Loss: 0.5141
Epoch [35/100], Loss: 0.4478
Epoch [36/100], Loss: 0.5066
Epoch [37/100], Loss: 0.4727
Epoch [38/100], Loss: 0.4530
Epoch [39/100], Loss: 0.4711
Epoch [40/100], Loss: 0.4277
Epoch [41/100], Loss: 0.4638
Epoch [42/100], Loss: 0.4734
Epoch [43/100], Loss: 0.4410
Epoch [44/100], Loss: 0.4750
Epoch [45/100], Loss: 0.4241
Epoch [46/100], Loss: 0.4414
```

```
Epoch [47/100], Loss: 0.5735
Epoch [48/100], Loss: 0.4204
Epoch [49/100], Loss: 0.4649
Epoch [50/100], Loss: 0.4405
Epoch [51/100], Loss: 0.4783
Epoch [52/100], Loss: 0.4769
Epoch [53/100], Loss: 0.4284
Epoch [54/100], Loss: 0.4111
Epoch [55/100], Loss: 0.4272
Epoch [56/100], Loss: 0.4863
Epoch [57/100], Loss: 0.4392
Epoch [58/100], Loss: 0.4137
Epoch [59/100], Loss: 0.4195
Epoch [60/100], Loss: 0.4857
Epoch [61/100], Loss: 0.4733
Epoch [62/100], Loss: 0.4258
Epoch [63/100], Loss: 0.4185
Epoch [64/100], Loss: 0.4048
Epoch [65/100], Loss: 0.4563
Epoch [66/100], Loss: 0.4183
Epoch [67/100], Loss: 0.4831
Epoch [68/100], Loss: 0.5334
Epoch [69/100], Loss: 0.4311
Epoch [70/100], Loss: 0.4908
Epoch [71/100], Loss: 0.4396
Epoch [72/100], Loss: 0.4244
Epoch [73/100], Loss: 0.4514
Epoch [74/100], Loss: 0.4238
Epoch [75/100], Loss: 0.4764
Epoch [76/100], Loss: 0.4454
Epoch [77/100], Loss: 0.4340
Epoch [78/100], Loss: 0.4671
Epoch [79/100], Loss: 0.4570
Epoch [80/100], Loss: 0.4382
Epoch [81/100], Loss: 0.4522
Epoch [82/100], Loss: 0.4574
Epoch [83/100], Loss: 0.4529
Epoch [84/100], Loss: 0.4441
Epoch [85/100], Loss: 0.4592
Epoch [86/100], Loss: 0.4888
Epoch [87/100], Loss: 0.4332
Epoch [88/100], Loss: 0.4039
Epoch [89/100], Loss: 0.4684
Epoch [90/100], Loss: 0.4263
Epoch [91/100], Loss: 0.4360
Epoch [92/100], Loss: 0.4245
Epoch [93/100], Loss: 0.4292
Epoch [94/100], Loss: 0.4371
```

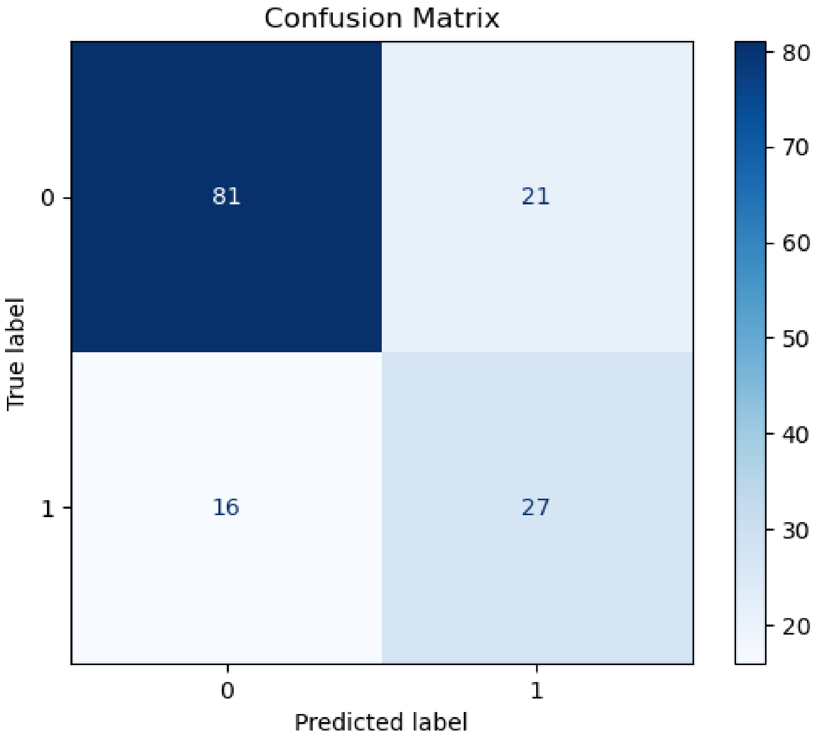
```
Epoch [95/100], Loss: 0.4524
Epoch [96/100], Loss: 0.4075
Epoch [97/100], Loss: 0.4413
Epoch [98/100], Loss: 0.4369
Epoch [99/100], Loss: 0.4529
Epoch [100/100], Loss: 0.4235
```

[10]:
```
y_pred=model(Xte)
print(f'ACC:{accuracy_score(yte.detach().numpy(),y_pred.detach().numpy()>0.
↪5)}') #classification
```

```
ACC:0.7931034482758621
```

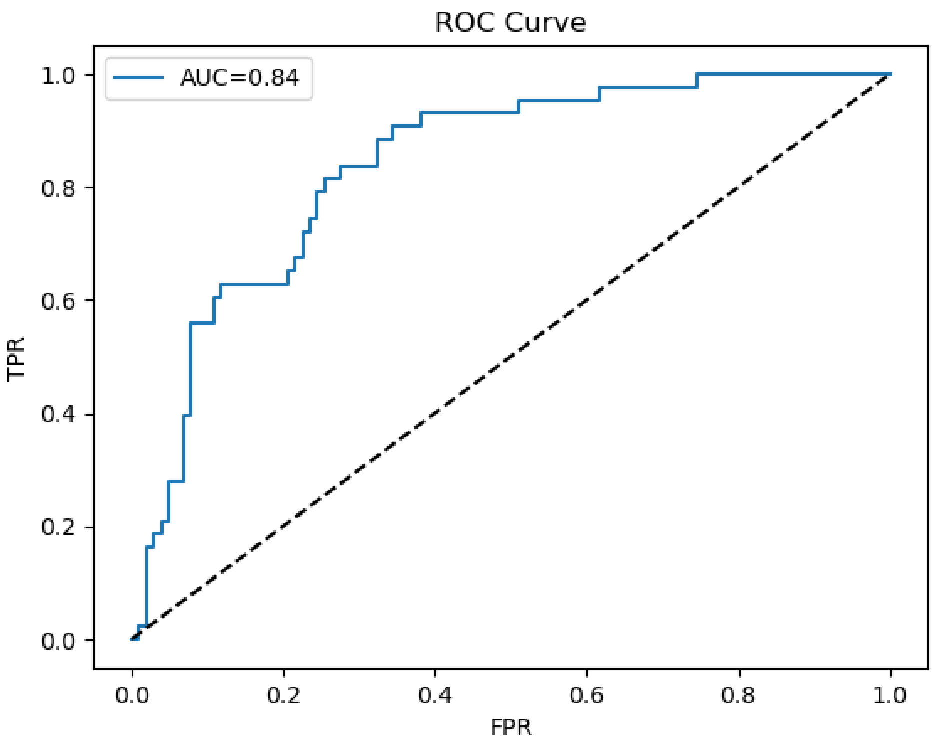
[11]:
```
model.eval()
device = next(model.parameters()).device

Xte_t = torch.as_tensor(Xte, dtype=torch.float32, device=device)
with torch.no_grad():
    logits = model(Xte_t)
    # make it 1D
    if logits.ndim > 1: logits = logits.squeeze(-1)
    y_proba = torch.sigmoid(logits).cpu().numpy()

print('proba min/mean/max:', y_proba.min(), y_proba.mean(), y_proba.max())
y_pred = (y_proba > 0.5).astype(int)

# ---- confusion matrix ----
cm = confusion_matrix(yte, y_pred)
ConfusionMatrixDisplay(cm).plot(cmap="Blues")
plt.title("Confusion Matrix")
plt.show()
plt.savefig("../Plots/ConfusionMatrixMLP.pdf", format="pdf",␣
↪bbox_inches="tight")
```

```
proba min/mean/max: 0.0020300266 0.36886007 0.93190086
```



<Figure size 640x480 with 0 Axes>

[12]:
```
fpr, tpr, _ = roc_curve(yte, y_proba)
roc_auc = auc(fpr, tpr)
plt.plot(fpr, tpr, label=f"AUC={roc_auc:.2f}")
plt.plot([0,1],[0,1],'k--')
plt.xlabel("FPR"); plt.ylabel("TPR"); plt.title("ROC Curve"); plt.legend(); plt.
↪show()
plt.savefig("../Plots/RoCMLP.pdf", format="pdf", bbox_inches="tight")
```




```
<Figure size 640x480 with 0 Axes>
```

```
[ ]:
```