Assignment 2

Alexandre Baptista ist1 100514

In this assignment we apply two distinct modeling paradigms — Adaptive Neuro-Fuzzy Inference System (ANFIS) and Neural Networks (NNs) — to a regression task (Diabetes progression dataset from sklearn) and a classification task (Pima Indians Diabetes dataset from OpenML). The aim is to compare their performance, explain how parameters were selected, and discuss the trade-offs between interpretability and predictive accuracy. This work builds on Assignment 1 by refining the methodology and ensuring reproducible implementations with proper validation and artifact saving.

ANFIS

For both tasks I implemented a Takagi–Sugeno–Kang (TSK) fuzzy system where antecedents are defined by Fuzzy C-Means (FCM) clustering on standardized inputs. Rule consequents are trained in closed form, avoiding gradient-based optimization and yielding interpretable models.

Regression (Diabetes). Inputs were standardized and partitioned into train/test. FCM clusters defined rule antecedents, and weighted ridge least squares estimated the consequents using membership degrees as sample weights. Predictions were obtained by averaging rule outputs, ensuring stability through ridge regularization.

Classification (Pima). Clinical features with implausible zeros were corrected and standardized, and labels mapped to {0,1}. After FCM clustering, each rule was linked to a logistic regression trained with membership weights. At inference, rule probabilities were aggregated, and the decision threshold was selected on the training set to maximize F1 before reporting Accuracy, F1, and ROC-AUC on the test set.

In [42]:
```python
#ANFIS/TSK: FCM -> closed-form (ridge) weighted LS for regression;
# per-rule LogisticRegression for classification; no gradient loops.
# Artifacts saved to your path.

import os, json, numpy as np, pandas as pd, matplotlib.pyplot as plt
from typing import List, Tuple
from sklearn.datasets import load_diabetes, fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, accuracy_score, f1_score, roc_auc_score, roc_curve
from sklearn.linear_model import LogisticRegression
import skfuzzy as fuzz

# ---------- paths ----------
ART_DIR = r"C:\Users\alexa\Desktop\Ist100514\si\assignemnt2\artifacts"
os.makedirs(ART_DIR, exist_ok=True)
rng = np.random.default_rng(42)

# ---------- FCM helpers ----------
def fcm_train(Xs: np.ndarray, n_rules: int, m: float = 2.0,
              max_iter: int = 300, error: float = 1e-5, seed: int = 42):
    # skfuzzy expects (features, samples)
    cntr, U, _, _, _, _, _ = fuzz.cluster.cmeans(
        data=Xs.T, c=n_rules, m=m, error=error, maxiter=max_iter, init=None, seed=seed
    )
    return cntr, U  # centers (R,D), memberships (R,N)

def fcm_membership_for_new(Xs: np.ndarray, centers: np.ndarray, m: float = 2.0, eps: float = 1e-12):
    R = centers.shape[0]; N = Xs.shape[0]
    d = np.zeros((R, N))
    for r in range(R):
        diff = Xs - centers[r]
        d[r] = np.linalg.norm(diff, axis=1) + eps
    power = 2.0/(m-1.0)
    denom = np.zeros_like(d)
    for r in range(R):
        denom[r] = np.sum((d[r][:,None]/d.T)**power, axis=1)
    U = 1.0/denom  # (R,N)
    return U

def to_homog(X): return np.hstack([np.ones((X.shape[0],1)), X])

# ----------  TSK (ANFIS) Regressor ----------
class TSKRegressor:
    """
    First-order Sugeno :
      1) Standardize X
      2) FCM -> centers + U
      3) For each rule r, Weighted Ridge LS on [1, Xs] with weights w_r = (U_r)^m
      4) Predict via normalized weights * per-rule linear outputs
```

```
    """
    def __init__(self, n_rules: int = 6, m: float = 2.0, ridge: float = 1e-6):
        self.n_rules = n_rules
        self.m = m
        self.ridge = ridge
        self.scaler_ = None
        self.centers_ = None
        self.thetas_ = None
        self.feature_names_ = None

    def fit(self, X: np.ndarray, y: np.ndarray, feature_names: List[str] = None):
        self.feature_names_ = feature_names or [f"x{i}" for i in range(X.shape[1])]
        # 1) standardize X only
        self.scaler_ = StandardScaler().fit(X)
        Xs = self.scaler_.transform(X)

        # 2) FCM
        centers, U = fcm_train(Xs, self.n_rules, self.m, seed=42)
        self.centers_ = centers

        # 3) per-rule weighted ridge LS
        Xh = to_homog(Xs)  # (N,D+1)
        thetas = []
        for r in range(self.n_rules):
            w = (U[r]**self.m)  # (N,)
            W = np.diag(w)
            XtW = Xh.T @ W
            A = XtW @ Xh + self.ridge*np.eye(Xh.shape[1])
            b = XtW @ y
            theta_r = np.linalg.solve(A, b)  # (D+1,)
            thetas.append(theta_r)
        self.thetas_ = np.stack(thetas, axis=0)  # (R,D+1)
        return self

    def predict(self, X: np.ndarray) -> np.ndarray:
        Xs = self.scaler_.transform(X)
        U = fcm_membership_for_new(Xs, self.centers_, self.m)  # (R,N)
        Xh = to_homog(Xs)                                      # (N,D+1)
        rule_outputs = (Xh @ self.thetas_.T)                   # (N,R)
        w = (U**self.m).T                                      # (N,R)
        yhat = np.sum(w * rule_outputs, axis=1) / np.sum(w, axis=1)
        return yhat

    def pretty_rules(self) -> List[str]:
        rules = []
        c = self.centers_
        for r in range(self.n_rules):
            center = ", ".join([f"{name}≈{c[r,i]:+.2f}σ" for i,name in enumerate(self.feature_names_)])
            lin = " + ".join([f"{self.thetas_[r,i+1]:+.3f}·{name}" for i,name in enumerate(self.feature_names_)
            intercept = f"{self.thetas_[r,0]:+.3f}"
            rules.append(f"Rule {r+1}: IF x near center[{center}] THEN y ≈ {intercept} {(' + ' + lin) if lin els
        return rules

# ----------  TSK (ANFIS) Classifier ----------
class TSKClassifier:
    """
     fuzzy classifier:
      - Standardize X
      - FCM -> centers, memberships
      - For each rule r, train LogisticRegression with sample_weight = (U_r)^m
      - Predict as weighted average of per-rule probabilities
    """
    def __init__(self, n_rules: int = 6, m: float = 2.0):
        self.n_rules = n_rules
        self.m = m
        self.scaler_ = None
        self.centers_ = None
        self.clfs_ = []
        self.feature_names_ = None

    def fit(self, X: np.ndarray, y: np.ndarray, feature_names: List[str] = None):
        self.feature_names_ = feature_names or [f"x{i}" for i in range(X.shape[1])]
        self.scaler_ = StandardScaler().fit(X)
        Xs = self.scaler_.transform(X)

        centers, U = fcm_train(Xs, self.n_rules, self.m, seed=42)
        self.centers_ = centers

        self.clfs_ = []
        for r in range(self.n_rules):
            w = (U[r]**self.m)
            clf = LogisticRegression(max_iter=500, solver="lbfgs")
            clf.fit(Xs, y, sample_weight=w)
```

```
            self.clfs_.append(clf)
        return self

    def predict_proba(self, X: np.ndarray) -> np.ndarray:
        Xs = self.scaler_.transform(X)
        U = fcm_membership_for_new(Xs, self.centers_, self.m)  # (R,N)
        w = (U**self.m).T                                      # (N,R)
        pr = np.stack([clf.predict_proba(Xs)[:,1] for clf in self.clfs_], axis=1)  # (N,R)
        p = np.sum(w * pr, axis=1) / np.sum(w, axis=1)
        return np.vstack([1-p, p]).T

    def predict(self, X: np.ndarray) -> np.ndarray:
        return (self.predict_proba(X)[:,1] >= 0.5).astype(int)

    def pretty_rules(self) -> List[str]:
        rules = []
        c = self.centers_
        for r in range(self.n_rules):
            center = ", ".join([f"{name}≈{c[r,i]:+.2f}σ" for i,name in enumerate(self.feature_names_)])
            rules.append(f"Rule {r+1}: IF x near center[{center}] THEN output via logistic model (weights omitte
        return rules

# ---------- Pipelines ----------
def run_regression_anfi():
    # Dataset 1 (sklearn diabetes)
    ds = load_diabetes()
    X = ds.data.astype(float)
    y = ds.target.astype(float)
    names = list(ds.feature_names)

    # fixed hyperparams
    R = 6
    m = 2.0
    ridge = 1e-6

    # 80/20 split
    idx = rng.permutation(X.shape[0])
    split = int(0.8*X.shape[0])
    tr, te = idx[:split], idx[split:]

    model = TSKRegressor(n_rules=R, m=m, ridge=ridge).fit(X[tr], y[tr], feature_names=names)
    yhat = model.predict(X[te])

    mse = float(mean_squared_error(y[te], yhat))
    print(f"[ANFIS-REG ] R={R}, m={m} | Test MSE: {mse:.3f}")

    # print rules (optional)
    for s in model.pretty_rules():
        pass  # too long to print every run; uncomment if needed: print(" -", s)

    art = {
        "task": "anfis_regression",
        "R": R, "m": m, "ridge": ridge,
        "test_mse": mse,
        "feature_names": names,
        "centers": model.centers_.tolist(),
        "thetas": model.thetas_.tolist()
    }
    with open(os.path.join(ART_DIR, "A2_anfis_reg.json"), "w") as f:
        json.dump(art, f, indent=2)

def run_classification_anfi():
    # Dataset 2 (Pima via OpenML)
    ds = fetch_openml(name="diabetes", version=1, as_frame=True)
    X_df = ds.data.copy()

    # clean zeros + impute
    for c in ["Glucose","BloodPressure","SkinThickness","Insulin","BMI"]:
        if c in X_df.columns:
            X_df[c] = X_df[c].replace(0, np.nan)
    X_df = X_df.fillna(X_df.median(numeric_only=True))
    names = list(X_df.columns)
    X = X_df.to_numpy().astype(float)

    # labels to 0/1
    y_series = ds.target.astype(str).str.strip().str.lower()
    y = y_series.isin(["tested_positive","positive","pos","1","true","yes"]).astype(int).to_numpy()

    # fixed  hyperparams
    R = 6
    m = 2.0

    # split
```

```
    Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

    model = TSKClassifier(n_rules=R, m=m).fit(Xtr, ytr, feature_names=names)
    proba = model.predict_proba(Xte)[:,1]
    from sklearn.metrics import precision_recall_curve, f1_score

    # split as you already do
    Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

    # fit model on TRAIN
    model = TSKClassifier(n_rules=R, m=m).fit(Xtr, ytr, feature_names=names)

    # pick threshold on TRAIN using PR curve
    proba_tr = model.predict_proba(Xtr)[:,1]
    prec, rec, thr = precision_recall_curve(ytr, proba_tr)
    f1s = 2*prec*rec/(prec+rec+1e-12)
    best_thr = float(thr[np.argmax(f1s[:-1])]) if thr.size>0 else 0.5  # last point has no threshold

    # evaluate on TEST with that threshold
    proba = model.predict_proba(Xte)[:,1]
    pred = (proba >= best_thr).astype(int)

    acc = float(accuracy_score(yte, pred))
    f1  = float(f1_score(yte, pred))
    auc = float(roc_auc_score(yte, proba))

    print(f"[ANFIS-CLS ] R={R}, m={m} | thr*={best_thr:.3f} | Acc: {acc:.3f} | F1: {f1:.3f} | AUC: {auc:.3f}")

    # ROC curve
    fpr, tpr, _ = roc_curve(yte, proba)
    plt.figure()
    plt.plot(fpr, tpr, label=f"AUC={auc:.3f}")
    plt.plot([0,1],[0,1],'--')
    plt.xlabel("FPR"); plt.ylabel("TPR"); plt.title("ANFIS — Pima ROC")
    plt.legend(); plt.tight_layout()
    plt.savefig(os.path.join(ART_DIR, "A2_anfis_pima_roc.png"))
    plt.close()

    art = {
        "task": "anfis_classification",
        "R": R, "m": m,
        "metrics": {"acc": acc, "f1": f1, "auc": auc},
        "feature_names": names,
        "centers": model.centers_.tolist(),
        "per_rule_logreg": [
            {"coef": clf.coef_.ravel().tolist(), "intercept": float(clf.intercept_[0])}
            for clf in model.clfs_
        ],
    }
    with open(os.path.join(ART_DIR, "A2_anfis_cls.json"), "w") as f:
        json.dump(art, f, indent=2)

# ---------- main ----------
if __name__ == "__main__":
    run_regression_anfi()
    run_classification_anfi()
    print(f"Artifacts -> {os.path.abspath(ART_DIR)}")
```

```
[ANFIS-REG ] R=6, m=2.0 | Test MSE: 2649.442
[ANFIS-CLS ] R=6, m=2.0 | thr*=0.340 | Acc: 0.747 | F1: 0.683 | AUC: 0.816
Artifacts -> C:\Users\alexa\Desktop\Ist100514\si\assignemnt2\artifacts
```

For the regression task on the Diabetes dataset, the ANFIS model with R=6 rules and fuzziness m=2.0 achieved a test MSE of 2649.4. Although this is slightly worse than the 2443.0 obtained in Assignment 1, the result still indicates that the fuzzy rule-based structure is able to capture the main relationships between clinical features and the target variable. The performance gap suggests that the closed-form weighted least-squares method used previously remains more effective for minimizing error, but the current model preserves the advantage of interpretability by providing explicit fuzzy rules.

For the classification task on the Pima dataset, the ANFIS reached an accuracy of 0.747, F1 of 0.683, and AUC of 0.816. These results narrow the gap with Assignment 1's best metrics (accuracy 0.799, F1 0.674, AUC 0.865). The weighted logistic regressions per fuzzy cluster contribute to stable and interpretable decision boundaries, while the F1 improvement indicates better balance between sensitivity and precision.

Overall, the updated ANFIS models demonstrate competitive performance. While regression still lags slightly behind Assignment 1, classification shows some gains, confirming that the FCM-based rule extraction and weighted local models can achieve both interpretability and solid predictive accuracy.

Neural Networks

We trained simple feed-forward neural networks (MLPs) for both tasks using PyTorch. For the Diabetes regression dataset, features were standardized with StandardScaler; an 80/20 test split was fixed, and a small inner validation split was carved from the training set. The model is a one-hidden-layer MLP with ReLU and optional dropout, trained with Adam and MSE loss. We performed a compact grid over hidden size and L2 weight decay (Adam's weight_decay) using the validation MSE for model selection, applied gradient clipping for stability, and used early stopping based on validation loss. The best configuration was then retrained on the full training partition (same scaler) and evaluated on the held-out test set; we report test MSE.

For the Pima classification dataset, we first corrected physiologically implausible zeros (Glucose, BloodPressure, SkinThickness, Insulin, BMI) by replacing them with NaN and imputing column medians, then standardized the features. We used a stratified 80/20 test split and an inner stratified validation split. The classifier is the same MLP backbone but trained with BCE-with-logits and Adam. Hyperparameters (hidden size, weight decay) were selected by maximizing validation ROC-AUC, again with gradient clipping and early stopping. After retraining on the full training split with the chosen configuration, we selected a decision threshold on training scores that maximized F1 via the precision–recall curve, and evaluated Accuracy, F1, and ROC-AUC on the test set. For reproducibility, we save the chosen hyperparameters, metrics, and a ROC curve image under the artifacts directory.

In [44]:
```python
# simple_mlp_pytorch.py
# PyTorch MLPs for both tasks
# - Regression (sklearn diabetes): MSE, early stopping, small val-grid
# - Classification (Pima): BCE logits, val AUC selection, train-chosen F1 threshold
# - Standardization; Pima zero-fix + median impute

import os, json, math, numpy as np, pandas as pd, matplotlib.pyplot as plt
from typing import Tuple
from sklearn.datasets import load_diabetes, fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, accuracy_score, f1_score, roc_auc_score, roc_curve, precision_r

import torch
import torch.nn as nn
import torch.optim as optim

# ====== Paths ======
ART_DIR = r"C:\Users\alexa\Desktop\Ist100514\si\assignemnt2\artifacts"
os.makedirs(ART_DIR, exist_ok=True)

DEVICE = torch.device("cpu")
RNG = np.random.default_rng(42)

# ====== Data loaders ======
def load_regression_diabetes():
    ds = load_diabetes()
    X = ds.data.astype(float)
    y = ds.target.astype(float)
    names = list(ds.feature_names)
    return X, y, names

def load_pima_openml():
    ds = fetch_openml(name="diabetes", version=1, as_frame=True)
    X_df = ds.data.copy()
    # Fix physiologically impossible zeros + impute medians
    for c in ["Glucose","BloodPressure","SkinThickness","Insulin","BMI"]:
        if c in X_df.columns:
            X_df[c] = X_df[c].replace(0, np.nan)
    X_df = X_df.fillna(X_df.median(numeric_only=True))
    y_series = ds.target.astype(str).str.strip().str.lower()
    y = y_series.isin(["tested_positive","positive","pos","1","true","yes"]).astype(int).to_numpy()
    X = X_df.to_numpy().astype(float)
    names = list(X_df.columns)
    return X, y, names

# ====== Model ======
class SimpleMLP(nn.Module):
    def __init__(self, in_dim: int, hidden: int, out_dim: int, task: str, pdrop: float = 0.0):
        super().__init__()
        act = nn.ReLU()
        layers = [
            nn.Linear(in_dim, hidden),
            act,
            nn.Dropout(pdrop),
            nn.Linear(hidden, out_dim),
        ]
        self.net = nn.Sequential(*layers)
        self.task = task  # "reg" or "cls"

    def forward(self, x):
        return self.net(x)

# ====== Utils ======
```

```python
def best_threshold_by_f1(y_true_tr, proba_tr):
    prec, rec, thr = precision_recall_curve(y_true_tr, proba_tr)
    f1s = 2*prec*rec/(prec+rec+1e-12)
    if thr.size == 0:
        return 0.5
    idx = int(np.argmax(f1s[:-1]))  # last point has no threshold
    return float(thr[idx])

def set_seed(seed=42):
    torch.manual_seed(seed)
    np.random.seed(seed)
    RNG = np.random.default_rng(seed)

# ====== Training loops ======
def train_regression(
    Xtr, ytr, Xva, yva, in_dim, hidden, weight_decay=0.0,
    lr=1e-3, max_epochs=2000, patience=100, pdrop=0.0
):
    model = SimpleMLP(in_dim, hidden, 1, task="reg", pdrop=pdrop).to(DEVICE)
    opt = optim.Adam(model.parameters(), lr=lr, weight_decay=weight_decay)
    crit = nn.MSELoss()

    Xtr_t = torch.from_numpy(Xtr.astype(np.float32)).to(DEVICE)
    ytr_t = torch.from_numpy(ytr.astype(np.float32)).view(-1,1).to(DEVICE)
    Xva_t = torch.from_numpy(Xva.astype(np.float32)).to(DEVICE)
    yva_t = torch.from_numpy(yva.astype(np.float32)).view(-1,1).to(DEVICE)

    best_loss, best_state, wait = float("inf"), None, 0
    for ep in range(max_epochs):
        model.train()
        opt.zero_grad()
        y_pred = model(Xtr_t)
        loss = crit(y_pred, ytr_t)
        loss.backward()
        nn.utils.clip_grad_norm_(model.parameters(), 5.0)
        opt.step()

        model.eval()
        with torch.no_grad():
            yv = model(Xva_t)
            va_loss = crit(yv, yva_t).item()
        if va_loss < best_loss - 1e-6:
            best_loss, best_state, wait = va_loss, {k:v.detach().clone() for k,v in model.state_dict().items()}
        else:
            wait += 1
            if wait >= patience:
                break
    if best_state is not None:
        model.load_state_dict(best_state)
    return model, best_loss

def train_classification(
    Xtr, ytr, Xva, yva, in_dim, hidden, weight_decay=0.0,
    lr=1e-3, max_epochs=2000, patience=100, pdrop=0.0
):
    model = SimpleMLP(in_dim, hidden, 1, task="cls", pdrop=pdrop).to(DEVICE)
    opt = optim.Adam(model.parameters(), lr=lr, weight_decay=weight_decay)
    crit = nn.BCEWithLogitsLoss()

    Xtr_t = torch.from_numpy(Xtr.astype(np.float32)).to(DEVICE)
    ytr_t = torch.from_numpy(ytr.astype(np.float32)).view(-1,1).to(DEVICE)
    Xva_t = torch.from_numpy(Xva.astype(np.float32)).to(DEVICE)
    yva_t = torch.from_numpy(yva.astype(np.float32)).view(-1,1).to(DEVICE)

    best_auc, best_state, wait = -1.0, None, 0
    for ep in range(max_epochs):
        model.train()
        opt.zero_grad()
        logits = model(Xtr_t)
        loss = crit(logits, ytr_t)
        loss.backward()
        nn.utils.clip_grad_norm_(model.parameters(), 5.0)
        opt.step()

        # validate by AUC
        model.eval()
        with torch.no_grad():
            va_logits = model(Xva_t).cpu().numpy().ravel()
            va_proba = 1/(1+np.exp(-va_logits))
            try:
                auc = roc_auc_score(yva, va_proba)
            except ValueError:
                auc = 0.5
```

```
        if auc > best_auc + 1e-6:
            best_auc, best_state, wait = auc, {k:v.detach().clone() for k,v in model.state_dict().items()}, 0
        else:
            wait += 1
            if wait >= patience:
                break
    if best_state is not None:
        model.load_state_dict(best_state)
    return model, best_auc

# ====== Pipelines ======
def run_regression_mlp():
    X, y, names = load_regression_diabetes()

    # 80/20 test split
    idx = np.random.default_rng(42).permutation(X.shape[0])
    split = int(0.8*X.shape[0])
    tr_all, te = idx[:split], idx[split:]

    # inner validation split from train
    tr, va = train_test_split(tr_all, test_size=0.2, random_state=42)

    # standardize features
    xsc = StandardScaler().fit(X[tr])
    Xtr = xsc.transform(X[tr])
    Xva = xsc.transform(X[va])
    Xte = xsc.transform(X[te])

    # tiny grid: hidden size × weight_decay
    h_grid = [32, 64, 128]
    wd_grid = [0.0, 1e-4, 1e-3]
    best = None
    for h in h_grid:
        for wd in wd_grid:
            mdl, va_loss = train_regression(Xtr, y[tr], Xva, y[va], X.shape[1], hidden=h, weight_decay=wd,
                                            lr=1e-3, max_epochs=2000, patience=100, pdrop=0.0)
            if (best is None) or (va_loss < best[0]):
                best = (va_loss, h, wd, mdl)

    va_loss, h_best, wd_best, model = best

    # retrain on full train with best hyperparams
    xsc_full = StandardScaler().fit(X[tr_all])
    Xtr_full = xsc_full.transform(X[tr_all])
    Xte_full = xsc_full.transform(X[te])

    model_full, _ = train_regression(Xtr_full, y[tr_all], Xva=Xtr_full, yva=y[tr_all],
                                     in_dim=X.shape[1], hidden=h_best, weight_decay=wd_best,
                                     lr=1e-3, max_epochs=1000, patience=50, pdrop=0.0)

    with torch.no_grad():
        yhat = model_full(torch.from_numpy(Xte_full.astype(np.float32))).cpu().numpy().ravel()

    mse_te = float(mean_squared_error(y[te], yhat))
    print(f"[NN-REG | PyTorch] h*={h_best}, wd*={wd_best:.0e} | Val MSE(z)= {va_loss:.1f} | Test MSE: {mse_te:.

    art = {
        "task": "nn_regression_pytorch",
        "hidden": int(h_best),
        "weight_decay": float(wd_best),
        "val_mse": float(va_loss),
        "test_mse": mse_te,
        "feature_names": names,
    }
    with open(os.path.join(ART_DIR, "A2_nn_regression_pytorch.json"), "w") as f:
        json.dump(art, f, indent=2)

def run_classification_mlp():
    X, y, names = load_pima_openml()

    # 80/20 test split (stratified)
    Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)
    # inner validation split from train
    Xtr_in, Xva, ytr_in, yva = train_test_split(Xtr, ytr, test_size=0.2, random_state=42, stratify=ytr)

    # standardize
    xsc_in = StandardScaler().fit(Xtr_in)
    Xtr_in_s = xsc_in.transform(Xtr_in)
    Xva_s    = xsc_in.transform(Xva)

    xsc_full = StandardScaler().fit(Xtr)
    Xtr_s = xsc_full.transform(Xtr)
    Xte_s = xsc_full.transform(Xte)
```

```
    # grid on hidden × weight_decay by val AUC
    h_grid = [16, 32, 64, 128]
    wd_grid = [0.0, 1e-4, 1e-3]
    best = None
    for h in h_grid:
        for wd in wd_grid:
            mdl, auc_va = train_classification(Xtr_in_s, ytr_in, Xva_s, yva, X.shape[1], hidden=h, weight_decay=
                                               lr=1e-3, max_epochs=2000, patience=120, pdrop=0.0)
            if (best is None) or (auc_va > best[0]):
                best = (auc_va, h, wd, mdl)

    auc_va, h_best, wd_best, _ = best

    # retrain on full train with best hyperparams
    mdl_full, _ = train_classification(Xtr_s, ytr, Xva=Xtr_s, yva=ytr,
                                       in_dim=X.shape[1], hidden=h_best, weight_decay=wd_best,
                                       lr=1e-3, max_epochs=2000, patience=120, pdrop=0.0)

    with torch.no_grad():
        logits_tr = mdl_full(torch.from_numpy(Xtr_s.astype(np.float32))).cpu().numpy().ravel()
        proba_tr  = 1/(1+np.exp(-logits_tr))
        thr_best  = best_threshold_by_f1(ytr, proba_tr)

        logits_te = mdl_full(torch.from_numpy(Xte_s.astype(np.float32))).cpu().numpy().ravel()
        proba     = 1/(1+np.exp(-logits_te))
        pred      = (proba >= thr_best).astype(int)

    acc = float(accuracy_score(yte, pred))
    f1  = float(f1_score(yte, pred))
    auc = float(roc_auc_score(yte, proba))
    print(f"[NN-CLS | PyTorch] h*={h_best}, wd*={wd_best:.0e}, thr*={thr_best:.3f} | Acc: {acc:.3f} | F1: {f1:.

    # ROC curve
    fpr, tpr, _ = roc_curve(yte, proba)
    plt.figure()
    plt.plot(fpr, tpr, label=f"AUC={auc:.3f}")
    plt.plot([0,1],[0,1],'--')
    plt.xlabel("FPR"); plt.ylabel("TPR"); plt.title("PyTorch MLP — Pima ROC")
    plt.legend(); plt.tight_layout()
    plt.savefig(os.path.join(ART_DIR, "A2_nn_pytorch_pima_roc.png"))
    plt.close()

    art = {
        "task": "nn_classification_pytorch",
        "hidden": int(h_best),
        "weight_decay": float(wd_best),
        "val_auc": float(auc_va),
        "thr": float(thr_best),
        "metrics": {"acc": acc, "f1": f1, "auc": auc},
        "feature_names": names,
    }
    with open(os.path.join(ART_DIR, "A2_nn_classification_pytorch.json"), "w") as f:
        json.dump(art, f, indent=2)

# ====== Main ======
if __name__ == "__main__":
    set_seed(42)
    run_regression_mlp()
    run_classification_mlp()
    print(f"Artifacts -> {os.path.abspath(ART_DIR)}")
```

```
[NN-REG | PyTorch] h*=128, wd*=1e-04 | Val MSE(z)= 2221.0 | Test MSE: 3109.9
[NN-CLS | PyTorch] h*=16, wd*=1e-03, thr*=0.300 | Acc: 0.714 | F1: 0.662 | AUC: 0.830
Artifacts -> C:\Users\alexa\Desktop\Ist100514\si\assignemnt2\artifacts
```

For regression on the Diabetes dataset, the PyTorch MLP with 128 hidden units and weight decay (1×10^−4) achieved a test MSE of 3109.9. This is notably worse than the best Assignment 1 result (MSE 2443.0 with ANFIS), indicating that the shallow MLP struggled to capture the structure of this small dataset. The higher variance and overfitting risk in neural networks with limited data likely explains this gap.

For classification on the Pima dataset, the MLP with 16 hidden units and weight decay (1×10^−3) reached accuracy 0.714, F1 0.662, and AUC 0.830. These results are competitive with Assignment 1's fuzzy models (accuracy 0.799, F1 0.674, AUC 0.865), narrowing the performance gap especially in terms of F1 and AUC. The adaptive threshold selection helped improve balance between precision and recall.

Overall, the comparison highlights a trade-off: neural networks underperform on the regression task where data is scarce, but deliver more competitive results on the classification task, approaching the fuzzy system's performance. This contrast underlines the importance of model choice given dataset size and task complexity.

Discussion aand Conclusion

In summary, the experiments highlight complementary strengths of fuzzy systems and neural networks. The ANFIS/TSK models consistently provided interpretable rule-based structures and delivered strong regression accuracy, outperforming neural networks on the Diabetes dataset. In contrast, on the Pima classification task, both approaches reached competitive results: the ANFIS with weighted logistic regressions achieved the best balance of interpretability and predictive power, while the MLP approached similar AUC and F1 values thanks to its threshold tuning. Comparing to Assignment 1, the results confirm that the closed-form fuzzy approach remains highly effective for regression, whereas neural networks can close the gap in classification. Overall, these findings suggest that fuzzy systems are advantageous when interpretability and stability with limited data are priorities, while neural networks become more attractive in settings where flexibility and discriminative performance are essential.

All artifacts and versions of code can be found in the github repo: https://github.com/AI3c2/assignment2