Koushik Sahu
118CS0597
Soft Computing Lab – V
7th February 2022

```
def beats_types():
    normal_beats = ['N', 'L', 'R', 'e', 'j']
    sup_beats = ['A', 'a', 'J', 'S']
    ven_beats = ['V', 'E']
    fusion_beats = ['F']
    unknown_beat = ['/', 'f', 'Q']

    return normal_beats, sup_beats, ven_beats, fusion_beats, unknown_beat
def split_dataset():
    '''
        Split dataset into training and testing follow AAMI Standard
    '''
    training = [101, 106, 108, 109, 112, 114,
            115, 116, 118, 119, 122, 124,
            201, 203, 205, 207, 208, 209,
            215, 220, 223, 230]
    testing = [100, 103, 105, 111, 113, 117,
            121, 123, 200, 202, 210,212,
            213, 214, 219, 221, 222, 228,
            231, 232, 233, 234]

    return training, testing

def Split_Dataset_Types(Dataset):
    if Dataset is None:
        raise ("Input specific dataset file name.")

    Normal, Sup, Ven, Fusion, Unknown = list(), list(), list(), list(), list()
    Beat_types = beats_types()
    Resample = 128
    for t in Dataset:
        Ann_path = f'mitbih_database/{t}annotations.txt'
        Data_path = f'mitbih_database/{t}.csv'

        Raw_ECG = load_data(Data_path)
        R_Indexs, Symbols = load_ann(Ann_path)

        Length_RRI = len(R_Indexs)

        for L in range(Length_RRI-2):
            Ind1 = int((R_Indexs[L] + R_Indexs[L+1]) / 2)
            Ind2 = int((R_Indexs[L+1] + R_Indexs[L+2]) / 2)

            Symb = Symbols[L+1]
            Sign = Raw_ECG[Ind1:Ind2]
            Resamp = resample(Sign, Resample)
```

```
        plt.plot(Sign)

        plt.show()

        plt.plot(Resamp)
        plt.show()
        if Symb in Beat_types[0]:
            Normal.append(np.array(Resamp))
        elif Symb in Beat_types[1]:
            Sup.append(np.array(Resamp))
        elif Symb in Beat_types[2]:
            Ven.append(np.array(Resamp))
        elif Symb in Beat_types[3]:
            Fusion.append(np.array(Resamp))
        elif Symb in Beat_types[4]:
            Unknown.append(np.array(Resamp))

    Normal = np.asarray(Normal, dtype=int)
    Sup = np.asarray(Sup, dtype=int)
    Ven = np.asarray(Ven, dtype=int)
    Fusion = np.asarray(Fusion, dtype=int)
    Unknown = np.asarray(Unknown, dtype=int)

    return Normal, Sup, Ven, Fusion, Unknown
def create_labels(N, S, V, F, U):
    N_Labels = np.zeros(len(N), dtype=int)
    S_Labels = np.ones(len(S), dtype=int)
    V_Labels = np.full((len(V)), 2, dtype=int)
    F_Labels = np.full((len(F)), 3, dtype=int)
    U_Labels = np.full((len(U)), 4, dtype=int)

    N_Labels = np.reshape(N_Labels, (-1, 1))
    S_Labels = np.reshape(S_Labels, (-1, 1))
    V_Labels = np.reshape(V_Labels, (-1, 1))
    F_Labels = np.reshape(F_Labels, (-1, 1))
    U_Labels = np.reshape(U_Labels, (-1, 1))

    return N_Labels, S_Labels, V_Labels, F_Labels, U_Labels
import numpy as np
import itertools

def f_activation(z):

    a = np.zeros_like(z)

    idx = (z >= 0.0)
    a[idx] = 1.0 / (1.0 + np.exp(-z[idx]))

    idx = np.invert(idx)          # Same as idx = (z < 0.0)
    a[idx] = np.exp(z[idx]) / (1.0 + np.exp(z[idx]))
```

```
    return a

def logsig(z):

    a = np.zeros_like(z)

    idx = (z < -33.3)
    a[idx] = z[idx]

    idx = (z >= -33.3) & (z < -18.0)
    a[idx] = z[idx] - np.exp(z[idx])

    idx = (z >= -18.0) & (z < 37.0)
    a[idx] = - np.log1p(np.exp(-z[idx]))

    idx = (z >= 37.0)
    a[idx] = - np.exp(-z[idx])

    return a

def build_class_matrix(Y):
    """
    Builds the output array <Yout> for a classification problem. Array <Y> has
    dimensions (n_samples, 1) and <Yout> has dimension (n_samples, n_classes).
    Yout[i,j] = 1 specifies that the i-th sample belongs to the j-th class.
    """
    n_samples = Y.shape[0]

    # Classes and corresponding number
    Yu, idx = np.unique(Y, return_inverse=True)
    n_classes = len(Yu)

    # Build the array actually used for classification
    Yout = np.zeros((n_samples, n_classes))
    Yout[np.arange(n_samples), idx] = 1.0

    return Yout, Yu

class ANFIS:

    def __init__(self, n_mf, n_outputs, problem=None):
        """
        n_mf        (n_inputs, )        Number of MFs in each feature/input
        n_outputs                       Number of labels/classes
        problem     C = classification problem, otherwise continuous problem
        """
        self.n_mf = np.asarray(n_mf)
        self.n_outputs = n_outputs
        self.problem = problem

        self.n_inputs = len(n_mf)             # Number of features/inputs
```

```
    self.n_pf = self.n_mf.sum()           # Number of premise MFs
    self.n_cf = self.n_mf.prod()          # Number of consequent MFs

    # Number of variables
    self.n_var = 3 * self.n_pf \
            + (self.n_inputs + 1) * self.n_cf * self.n_outputs

    self.init_prob = True                 # Initialization flag
    self.Xe = np.array([])                # Extended input array

    # For logistic regression only
    if (self.problem == 'C'):
      self.Yout = np.array([])            # Actual output
      self.Yu = np.array([])              # Class list

  def create_model(self, theta, args):
    """
    Creates the model for the regression problem.
    """
    # Unpack
    X = args[0]              # Input dataset
    Y = args[1]              # Output dataset

    # First time only
    if (self.init_prob):
      self.init_prob = False

      # Build all combinations of premise MFs
      self.build_combs()

      # Expand the input dataset to match the number of premise MFs.
      self.Xe = self.expand_input_dataset(X)

      # For classification initialize Yout (output) and Yu (class list)
      if (self.problem == 'C'):
        self.Yout, self.Yu = build_class_matrix(Y)

    # Builds the premise/consequent parameters mu, s, c, and A
    self.build_param(theta)

    # Calculate the output
    f = self.forward_steps(X, self.Xe)

    # Cost function for classification problems (the activation value is
    # calculated in the logsig function)
    if (self.problem == 'C'):
      error = (1.0 - self.Yout) * f - logsig(f)
      J = error.sum() / float(X.shape[0])

    # Cost function for continuous problems
    else:
```

```
            error = f - Y
            J = (error ** 2).sum() / 2.0

        return J

    def eval_data(self, Xp):
        """
        Evaluates the input dataset with the model created in <create_model>.
        """
        # Expand the input dataset to match the number of premise MFs.
        Xpe = self.expand_input_dataset(Xp)

        # Calculate the output
        f = self.forward_steps(Xp, Xpe)

        # Classification problem
        if (self.problem == 'C'):
            A = f_activation(f)
            idx = np.argmax(A, axis=1)
            Yp = self.Yu[idx].reshape((len(idx), 1))

        # Continuous problem
        else:
            Yp = f

        return Yp

    def build_combs(self):
        """
        Builds all combinations of premise functions.

        For example if <n_mf> = [3, 2], the MF indexes for the first feature
        would be [0, 1, 2] and for the second feature would be [3, 4]. The
        resulting combinations would be <combs> = [[0 0 1 1 2 2],
                                                   [3 4 3 4 3 4]].
        """
        idx = np.cumsum(self.n_mf)
        v = [np.arange(0, idx[0])]

        for i in range(1, self.n_inputs):
            v.append(np.arange(idx[i-1], idx[i]))

        list_combs = list(itertools.product(*v))
        self.combs = np.asarray(list_combs).T

    def expand_input_dataset(self, X):
        """
        Expands the input dataset to match the number of premise MFs. Each MF
        will be paired with the correct feature in the dataset.
        """
        n_samples = X.shape[0]
```

```
        Xe = np.zeros((n_samples, self.n_pf))      # Expanded array
        idx = np.cumsum(self.n_mf)
        i1 = 0

        for i in range(self.n_inputs):
            i2 = idx[i]
            Xe[:, i1:i2] = X[:, i].reshape(n_samples, 1)
            i1 = idx[i]

        return Xe

    def build_param(self, theta):
        """
        Builds the premise/consequent parameters  mu, s, c, and A.
        """
        i1 = self.n_pf
        i2 = 2 * i1
        i3 = 3 * i1
        i4 = self.n_var

        # Premise function parameters (generalized Bell functions)
        self.mu = theta[0:i1]
        self.s = theta[i1:i2]
        self.c = theta[i2:i3]

        # Consequent function parameters (hyperplanes)
        self.A = \
            theta[i3:i4].reshape(self.n_inputs + 1, self.n_cf * self.n_outputs)

    def forward_steps(self, X, Xe):
        """
        Calculate the output giving premise/consequent parameters and the
        input dataset.
        """
        n_samples = X.shape[0]

        # Layer 1: premise functions (pf)
        d = (Xe - self.mu) / self.s
        pf = 1.0 / (1.0 + (d * d) ** self.c)

        # Layer 2: firing strenght (W)
        W = np.prod(pf[:, self.combs], axis=1)

        # Layer 3: firing strenght ratios (Wr)
        Wr = W / W.sum(axis=1, keepdims=True)

        # Layer 4and 5: consequent functions (cf) and output (f)
        X1 = np.hstack((np.ones((n_samples, 1)), X))
        f = np.zeros((n_samples, self.n_outputs))
        for i in range(self.n_outputs):
            i1 = i * self.n_cf
```

```
            i2 = (i + 1) * self.n_cf
            cf = Wr * (X1 @ self.A[:, i1:i2])
            f[:, i] = cf.sum(axis=1)

        return f

    def param_anfis(self):
        """
        Returns the premise MFs parameters.
        """
        mu = self.mu
        s = self.s
        c = self.c
        A = self.A

        return mu, s, c, A
import sys
import numpy as np
import anfis as anf
import utils as utl
import pso as pso

# ======= Examples ======= #

# Read example to run
if len(sys.argv) != 2:
    print("Usage: python test.py <example>")
    sys.exit(1)
example = sys.argv[1]

np.random.seed(1294404794)

# Default values common to all examples
problem = None
split_factor = 0.70
K = 3
phi = 2.05
vel_fact = 0.5
conf_type = 'RB'
IntVar = None
normalize = False
rad = 0.1
mu_delta = 0.2
s_par = [0.5, 0.2]
c_par = [1.0, 3.0]
A_par = [-10.0, 10.0]

# Multi-class classification problem example
if (example == 'ECG'):
    data_file = 'ECG_dataset.csv'
    problem = 'C'
```

```
    n_mf = [3, 4, 2]
    nPop = 40
    epochs = 200

else:
    print("Example not found")
    sys.exit(1)

# ======= Data ======= #

# Read data from a csv file
data = np.loadtxt(data_file, delimiter=',')
n_samples, n_cols = data.shape

# Classification problem (the label column is always the last one)
if (problem == 'C'):
    n_inputs = n_cols - 1
    n_outputs, class_list = utl.get_classes(data[:, -1])

# Continuous problem (the label columns are always at the end)
else:
    n_inputs = len(n_mf)
    n_outputs = n_cols - n_inputs

# ANFIS info
n_pf, n_cf, n_var = utl.info_anfis(n_mf, n_outputs)

# Randomly build the training (tr) and test (te) datasets
rows_tr = int(split_factor * n_samples)
rows_te = n_samples - rows_tr
idx_tr = np.random.choice(np.arange(n_samples), size=rows_tr, replace=False)
idx_te = np.delete(np.arange(n_samples), idx_tr)
data_tr = data[idx_tr, :]
data_te = data[idx_te, :]

# Split the data
X_tr = data_tr[:, 0:n_inputs]
Y_tr = data_tr[:, n_inputs:]
X_te = data_te[:, 0:n_inputs]
Y_te = data_te[:, n_inputs:]

# System info
print("\nNumber of samples = ", n_samples)
print("Number of inputs = ", n_inputs)
print("Number of outputs = ", n_outputs)

if (problem == 'C'):
    print("\nClasses: ", class_list)

print("\nNumber of training samples = ", rows_tr)
print("Number of test samples= ", rows_te)
```

```
print("\nANFIS layout = ", n_mf)
print("Number of premise functions = ", n_pf)
print("Number of consequent functions = ", n_cf)
print("Number of variables = ", n_var)

# ======= PSO ======= #

def interface_PSO(theta, args):
    """
    Function to interface the PSO with the ANFIS. Each particle has its own
    ANFIS instance.

    theta          (nPop, n_var)
    learners       (nPop, )
    J              (nPop, )
    """
    args_PSO = (args[0], args[1])
    learners = args[2]
    nPop = theta.shape[0]

    J = np.zeros(nPop)
    for i in range(nPop):
        J[i] = learners[i].create_model(theta[i, :], args_PSO)

    return J

# Init learners (one for each particle)
learners = []
for i in range(nPop):
    learners.append(anf.ANFIS(n_mf=n_mf, n_outputs=n_outputs, problem=problem))

# Always normalize inputs
Xn_tr, norm_param = utl.normalize_data(X_tr)
Xn_te = utl.normalize_data(X_te, norm_param)

# Build boundaries using heuristic rules
LB, UB = utl.bounds_pso(Xn_tr, n_mf, n_outputs, mu_delta=mu_delta, s_par=s_par,
                        c_par=c_par, A_par=A_par)

# Scale output(s) in continuous problems to reduce the range in <A_par>
if (problem != 'C'):
    Y_tr, scal_param = utl.scale_data(Y_tr)
    Y_te = utl.scale_data(Y_te, scal_param)

# Optimize using PSO
# theta = best solution (min)
# info[0] = function value in theta
# info[1] = index of the learner with the best solution
# info[2] = number of learners close to the learner with the best solution
func = interface_PSO
```

```
args = (Xn_tr, Y_tr, learners)
theta, info = pso.PSO(func, LB, UB, nPop=nPop, epochs=epochs, K=K, phi=phi,
                      vel_fact=vel_fact, conf_type=conf_type, IntVar=IntVar,
                      normalize=normalize, rad=rad, args=args)

# ======= Solution ======= #

best_learner = learners[info[1]]
mu, s, c, A = best_learner.param_anfis()

print("\nSolution:")
print("J minimum = ", info[0])
print("Best learner = ", info[1])
print("Close learners = ", info[2])

print("\nCoefficients:")
print("mu = ", mu)
print("s  = ", s)
print("c  = ", c)
print("A =")
print(A)

# Plot resulting MFs
utl.plot_mfs(n_mf, mu, s, c, Xn_tr)

# Evaluate training and test datasets with best learner
# (in continuous problems these are already scaled values)
Yp_tr = best_learner.eval_data(Xn_tr)
Yp_te = best_learner.eval_data(Xn_te)

# Results for classification problems (accuracy and correlation)
if (problem == 'C'):
    print("\nAccuracy training data = ", utl.calc_accu(Yp_tr, Y_tr))
    print("Corr. training data = ", utl.calc_corr(Yp_tr, Y_tr))
    print("\nAccuracy test data = ", utl.calc_accu(Yp_te, Y_te))
    print("Corr. test data = ", utl.calc_corr(Yp_te, Y_te))

# Results for continuous problems (RMSE and correlation)
else:
    print("\nRMSE training data = ", utl.calc_rmse(Yp_tr, Y_tr))
    print("Corr. training data = ", utl.calc_corr(Yp_tr, Y_tr))
    print("\nRMSE test data = ", utl.calc_rmse(Yp_te, Y_te))
    print("Corr. test data = ", utl.calc_corr(Yp_te, Y_te))

# ======= Closed-Form Solution ======= #

"""
- For continuous problems if there is one premise function for each
  feature then the <A> parameters from the PSO solution should be equal
  to the <theta_sol> values.
- The solution when there are more than one premise function for each
```

```
  feature is still useful to compare correlations and RMSEs/accuracies.
- Classification problems are solved just like continuous problems.
"""
# Solve using the training dataset
X1n_tr = np.block([np.ones((Xn_tr.shape[0], 1)), Xn_tr])
theta_sol = utl.regression_sol(X1n_tr, Y_tr)

# Evaluate training and test datasets
Yp_tr_sol = X1n_tr @ theta_sol
X1n_te = np.block([np.ones((Xn_te.shape[0], 1)), Xn_te])
Yp_te_sol = X1n_te @ theta_sol

# Show results
print("\nClosed-form solution:")
print("theta =")
print(theta_sol)
print("\nCorr. training data = ", utl.calc_corr(Yp_tr_sol, Y_tr))
print("Corr. test data = ", utl.calc_corr(Yp_te_sol, Y_te))
```

Output: