**GWO-SVR**

from sklearn.svm import SVR

from sklearn.metrics import mean\_squared\_error

from sklearn.model\_selection import train\_test\_split

import plotly.express as px

import pandas as pd

import numpy as np

# Assuming you have a DataFrame 'df' with features and target column

# Replace 'your\_target\_column\_name' with the actual name of your target column

X = df.drop('MTD(m)', axis=1)

y = df['MTD(m)']

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Define the SVR model

svr\_model = SVR()

# Define the hyperparameter bounds for GWO

bounds = [

(0.01, 0.5), # C

(0.1, 1.0), # epsilon

(0.1, 1.0) # gamma

]

# Grey Wolf Optimization

def initialize\_wolves(num\_wolves, bounds):

wolves = [np.random.uniform(low=bounds[i][0], high=bounds[i][1], size=num\_wolves) for i in range(len(bounds))]

return np.array(wolves).T

def update\_position(alpha, beta, delta, wolf, a, A, C):

D = len(wolf)

for i in range(D):

r1, r2 = np.random.random(), np.random.random()

A1 = 2 \* a \* r1 - a

C1 = 2 \* r2

D\_alpha = abs(C1 \* alpha[i] - wolf[i])

X1 = alpha[i] - A1 \* D\_alpha

r1, r2 = np.random.random(), np.random.random()

A2 = 2 \* a \* r1 - a

C2 = 2 \* r2

D\_beta = abs(C2 \* beta[i] - wolf[i])

X2 = beta[i] - A2 \* D\_beta

r1, r2 = np.random.random(), np.random.random()

A3 = 2 \* a \* r1 - a

C3 = 2 \* r2

D\_delta = abs(C3 \* delta[i] - wolf[i])

X3 = delta[i] - A3 \* D\_delta

wolf[i] = (X1 + X2 + X3) / 3

# Ensure the new position is within the bounds

wolf[i] = np.clip(wolf[i], bounds[i][0], bounds[i][1])

return wolf

def gwo\_optimization(objective\_function, bounds, num\_wolves=5, iterations=50):

wolves = initialize\_wolves(num\_wolves, bounds)

best\_wolf = None

best\_fitness = np.inf

for \_ in range(iterations):

a = 2 - 2 \* \_ / iterations # linearly decreased from 2 to 0

for wolf in wolves:

fitness = objective\_function(wolf)

if fitness < best\_fitness:

best\_fitness = fitness

best\_wolf = wolf

for wolf in wolves:

wolf = update\_position(best\_wolf, best\_wolf, best\_wolf, wolf, a, 2 \* np.random.random(), 2 \* np.random.random())

return best\_wolf

# Define the objective function for GWO

def objective\_function(params):

svr\_model.set\_params(

C=params[0],

epsilon=params[1],

gamma=params[2]

)

svr\_model.fit(X\_train, y\_train)

y\_pred = svr\_model.predict(X\_test)

mse = mean\_squared\_error(y\_test, y\_pred)

return mse

# Perform GWO optimization

best\_params = gwo\_optimization(objective\_function, bounds)

# Create a DataFrame to store the results

df\_results = pd.DataFrame([best\_params])

df\_results.columns = ['C', 'epsilon', 'gamma']

df\_results['mse'] = objective\_function(best\_params)

# Print the best parameters and MSE

print("Best Parameters:", best\_params)

print("Best MSE:", df\_results['mse'].values[0])

# Plotting the results using Plotly

fig\_gwo = px.scatter\_matrix(

df\_results,

dimensions=['C', 'epsilon', 'gamma', 'mse'],

color='mse',

opacity=0.7,

width=1600,

height=800

)

# Update dot size, font size, and tick size

fig\_gwo.update\_traces(marker=dict(size=15)) # Increase dot size

fig\_gwo.update\_layout(font=dict(size=15)) # Increase font size

fig\_gwo.update\_layout(yaxis=dict(tickfont=dict(size=15))) # Increase tick size for y-axis

fig\_gwo.update\_layout(xaxis=dict(tickfont=dict(size=15))) # Increase tick size for x-axis

fig\_gwo.show()

**GWO-KNN**

from sklearn.neighbors import KNeighborsRegressor

from sklearn.metrics import mean\_squared\_error

from sklearn.model\_selection import train\_test\_split

import plotly.express as px

import pandas as pd

import numpy as np

# Assuming you have a DataFrame 'df' with features and target column

# Replace 'your\_target\_column\_name' with the actual name of your target column

X = df.drop('MTD(m)', axis=1)

y = df['MTD(m)']

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Define the KNN regressor

knn\_reg = KNeighborsRegressor()

# Define the hyperparameter bounds for GWO

bounds = [

(1, 10), # n\_neighbors

(1, 30), # leaf\_size

(1, 2) # p

]

# Grey Wolf Optimization

def initialize\_wolves(num\_wolves, bounds):

wolves = [np.random.uniform(low=bounds[i][0], high=bounds[i][1], size=num\_wolves) for i in range(len(bounds))]

return np.array(wolves).T

def update\_position(alpha, beta, delta, wolf, a, A, C):

D = len(wolf)

for i in range(D):

r1, r2 = np.random.random(), np.random.random()

A1 = 2 \* a \* r1 - a

C1 = 2 \* r2

D\_alpha = abs(C1 \* alpha[i] - wolf[i])

X1 = alpha[i] - A1 \* D\_alpha

r1, r2 = np.random.random(), np.random.random()

A2 = 2 \* a \* r1 - a

C2 = 2 \* r2

D\_beta = abs(C2 \* beta[i] - wolf[i])

X2 = beta[i] - A2 \* D\_beta

r1, r2 = np.random.random(), np.random.random()

A3 = 2 \* a \* r1 - a

C3 = 2 \* r2

D\_delta = abs(C3 \* delta[i] - wolf[i])

X3 = delta[i] - A3 \* D\_delta

wolf[i] = (X1 + X2 + X3) / 3

# Ensure the new position is within the bounds

wolf[i] = np.clip(wolf[i], bounds[i][0], bounds[i][1])

return wolf

def gwo\_optimization(objective\_function, bounds, num\_wolves=5, iterations=50):

wolves = initialize\_wolves(num\_wolves, bounds)

best\_wolf = None

best\_fitness = np.inf

for \_ in range(iterations):

a = 2 - 2 \* \_ / iterations # linearly decreased from 2 to 0

for wolf in wolves:

fitness = objective\_function(wolf)

if fitness < best\_fitness:

best\_fitness = fitness

best\_wolf = wolf

for wolf in wolves:

wolf = update\_position(best\_wolf, best\_wolf, best\_wolf, wolf, a, 2 \* np.random.random(), 2 \* np.random.random())

return best\_wolf

# Define the objective function for GWO

def objective\_function(params):

knn\_reg.set\_params(

n\_neighbors=int(params[0]),

leaf\_size=int(params[1]),

p=int(params[2])

)

knn\_reg.fit(X\_train, y\_train)

y\_pred = knn\_reg.predict(X\_test)

mse = mean\_squared\_error(y\_test, y\_pred)

return mse

# Perform GWO optimization

best\_params = gwo\_optimization(objective\_function, bounds)

# Create a DataFrame to store the results

df\_results = pd.DataFrame([best\_params])

df\_results.columns = ['n\_neighbors', 'leaf\_size', 'p']

df\_results['mse'] = objective\_function(best\_params)

# Print the best parameters and MSE

print("Best Parameters:", best\_params)

print("Best MSE:", df\_results['mse'].values[0])

# Plotting the results using Plotly

fig\_gwo = px.scatter\_matrix(

df\_results,

dimensions=['n\_neighbors', 'leaf\_size', 'p', 'mse'],

color='mse',

opacity=0.7,

width=1600,

height=800

)

# Update dot size, font size, and tick size

fig\_gwo.update\_traces(marker=dict(size=15)) # Increase dot size

fig\_gwo.update\_layout(font=dict(size=15)) # Increase font size

fig\_gwo.update\_layout(yaxis=dict(tickfont=dict(size=15))) # Increase tick size for y-axis

fig\_gwo.update\_layout(xaxis=dict(tickfont=dict(size=15))) # Increase tick size for x-axis

fig\_gwo.show()

**GWO-RF**

from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import mean\_squared\_error

from sklearn.model\_selection import train\_test\_split

import plotly.express as px

import pandas as pd

import numpy as np

# Assuming you have a DataFrame 'df' with features and target column

# Replace 'your\_target\_column\_name' with the actual name of your target column

X = df.drop('MTD(m)', axis=1)

y = df['MTD(m)']

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Define the Random Forest regressor

rf\_reg = RandomForestRegressor(random\_state=42)

# Define the hyperparameter bounds for GWO

bounds = [

(1, 10), # n\_estimators

(1, 100), # max\_depth

(2, 10), # min\_samples\_split

(1, 10), # min\_samples\_leaf

(0.1, 1.0) # max\_features

]

# Grey Wolf Optimization

def initialize\_wolves(num\_wolves, bounds):

wolves = [np.random.uniform(low=bounds[i][0], high=bounds[i][1], size=num\_wolves) for i in range(len(bounds))]

return np.array(wolves).T

def update\_position(alpha, beta, delta, wolf, a, A, C):

D = len(wolf)

for i in range(D):

r1, r2 = np.random.random(), np.random.random()

A1 = 2 \* a \* r1 - a

C1 = 2 \* r2

D\_alpha = abs(C1 \* alpha[i] - wolf[i])

X1 = alpha[i] - A1 \* D\_alpha

r1, r2 = np.random.random(), np.random.random()

A2 = 2 \* a \* r1 - a

C2 = 2 \* r2

D\_beta = abs(C2 \* beta[i] - wolf[i])

X2 = beta[i] - A2 \* D\_beta

r1, r2 = np.random.random(), np.random.random()

A3 = 2 \* a \* r1 - a

C3 = 2 \* r2

D\_delta = abs(C3 \* delta[i] - wolf[i])

X3 = delta[i] - A3 \* D\_delta

wolf[i] = (X1 + X2 + X3) / 3

# Ensure the new position is within the bounds

wolf[i] = np.clip(wolf[i], bounds[i][0], bounds[i][1])

return wolf

def gwo\_optimization(objective\_function, bounds, num\_wolves=5, iterations=50):

wolves = initialize\_wolves(num\_wolves, bounds)

best\_wolf = None

best\_fitness = np.inf

for \_ in range(iterations):

a = 2 - 2 \* \_ / iterations # linearly decreased from 2 to 0

for wolf in wolves:

fitness = objective\_function(wolf)

if fitness < best\_fitness:

best\_fitness = fitness

best\_wolf = wolf

for wolf in wolves:

wolf = update\_position(best\_wolf, best\_wolf, best\_wolf, wolf, a, 2 \* np.random.random(), 2 \* np.random.random())

return best\_wolf

# Define the objective function for GWO

def objective\_function(params):

rf\_reg.set\_params(

n\_estimators=int(params[0]),

max\_depth=int(params[1]),

min\_samples\_split=int(params[2]),

min\_samples\_leaf=int(params[3]),

max\_features=params[4]

)

rf\_reg.fit(X\_train, y\_train)

y\_pred = rf\_reg.predict(X\_test)

mse = mean\_squared\_error(y\_test, y\_pred)

return mse

# Perform GWO optimization

best\_params = gwo\_optimization(objective\_function, bounds)

# Create a DataFrame to store the results

df\_results = pd.DataFrame([best\_params])

df\_results.columns = ['n\_estimators', 'max\_depth', 'min\_samples\_split', 'min\_samples\_leaf', 'max\_features']

df\_results['mse'] = objective\_function(best\_params)

# Print the best parameters and MSE

print("Best Parameters:", best\_params)

print("Best MSE:", df\_results['mse'].values[0])

# Plotting the results using Plotly

fig\_gwo = px.scatter\_matrix(

df\_results,

dimensions=['n\_estimators', 'max\_depth', 'min\_samples\_split', 'min\_samples\_leaf', 'max\_features', 'mse'],

color='mse',

opacity=0.7,

width=1600,

height=800

)

# Update dot size, font size, and tick size

fig\_gwo.update\_traces(marker=dict(size=15)) # Increase dot size

fig\_gwo.update\_layout(font=dict(size=15)) # Increase font size

fig\_gwo.update\_layout(yaxis=dict(tickfont=dict(size=15))) # Increase tick size for y-axis

fig\_gwo.update\_layout(xaxis=dict(tickfont=dict(size=15))) # Increase tick size for x-axis

**GWO-XGB**

from mmap import MADV\_SEQUENTIAL

import xgboost as xgb

from sklearn.metrics import mean\_squared\_error

from sklearn.model\_selection import train\_test\_split

import plotly.express as px

import pandas as pd

import numpy as np

from sklearn.metrics import r2\_score

from google.colab import drive

drive.mount('/gdrive')

df=pd.read\_csv('/gdrive/My Drive/Colab Notebooks/Modified values.csv')

y = df[df.columns[4]]

X = df.iloc[:, 0:3]

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Define the XGBoost regressor

xgb\_reg = xgb.XGBRegressor(objective='reg:squarederror', random\_state=42)

# Define the hyperparameter bounds for GWO

bounds = [

    (0.01, 0.5),  # learning\_rate

    (50, 500),    # n\_estimators

    (3, 9),       # max\_depth

    (1, 7),       # min\_child\_weight

    (0.6, 1.0),   # subsample

    (0.6, 1.0),   # colsample\_bytree

    (0, 0.4)      # gamma

]

# Grey Wolf Optimization

def initialize\_wolves(num\_wolves, bounds):

    wolves = [np.random.uniform(low=bounds[i][0], high=bounds[i][1], size=num\_wolves) for i in range(len(bounds))]

    return np.array(wolves).T

def update\_position(alpha, beta, delta, wolf, a, A, C):

    D = len(wolf)

    for i in range(D):

        r1, r2 = np.random.random(), np.random.random()

        A1 = 2 \* a \* r1 - a

        C1 = 2 \* r2

        D\_alpha = abs(C1 \* alpha[i] - wolf[i])

        X1 = alpha[i] - A1 \* D\_alpha

        r1, r2 = np.random.random(), np.random.random()

        A2 = 2 \* a \* r1 - a

        C2 = 2 \* r2

        D\_beta = abs(C2 \* beta[i] - wolf[i])

        X2 = beta[i] - A2 \* D\_beta

        r1, r2 = np.random.random(), np.random.random()

        A3 = 2 \* a \* r1 - a

        C3 = 2 \* r2

        D\_delta = abs(C3 \* delta[i] - wolf[i])

        X3 = delta[i] - A3 \* D\_delta

        wolf[i] = (X1 + X2 + X3) / 3

        # Ensure the new position is within the bounds

        wolf[i] = np.clip(wolf[i], bounds[i][0], bounds[i][1])

    return wolf

def gwo\_optimization(objective\_function, bounds, num\_wolves=5, iterations=50):

    wolves = initialize\_wolves(num\_wolves, bounds)

    best\_wolf = None

    best\_fitness = np.inf

    for \_ in range(iterations):

        a = 2 - 2 \* \_ / iterations  # linearly decreased from 2 to 0

        for wolf in wolves:

            fitness = objective\_function(wolf)

            if fitness < best\_fitness:

                best\_fitness = fitness

                best\_wolf = wolf

        for wolf in wolves:

            wolf = update\_position(best\_wolf, best\_wolf, best\_wolf, wolf, a, 2 \* np.random.random(), 2 \* np.random.random())

    return best\_wolf

# Define the objective function for GWO

def objective\_function(params):

    xgb\_reg.set\_params(

        learning\_rate=params[0],

        n\_estimators=int(params[1]),

        max\_depth=int(params[2]),

        min\_child\_weight=int(params[3]),

        subsample=params[4],

        colsample\_bytree=params[5],

        gamma=params[6]

    )

    xgb\_reg.fit(X\_train, y\_train)

    y\_predtrain = xgb\_reg.predict(X\_train)

    mse\_train = mean\_squared\_error(y\_train, y\_predtrain)

    r2\_train=r2\_score(y\_train, y\_predtrain)

    y\_predtest = xgb\_reg.predict(X\_test)

    mse\_test = mean\_squared\_error(y\_test, y\_predtest)

    r2\_test=r2\_score(y\_test,y\_predtest)

   # with open('/gdrive/My Drive/Colab Notebooks/filename.txt', 'a') as f:

      #print("R2 score",r2, file=f)

    if r2\_test>0.9:

       print("R2 score",r2\_test)

       print("Actual labels (y\_test): ", y\_test.tolist())

       print("Predicted labels (y\_predtest): ", y\_predtest)

       print("R2 score",r2\_train)

       print("Actual labels (y\_train): ", y\_train.tolist())

       print("Predicted labels (y\_predtrain): ", y\_predtrain)

    #error=[mse, r2, [y\_test, y\_pred]]

    return mse\_test

# Perform GWO optimization

best\_params = gwo\_optimization(objective\_function, bounds)

# Create a DataFrame to store the results

df\_results = pd.DataFrame([best\_params])

df\_results.columns = ['learning\_rate', 'n\_estimators', 'max\_depth', 'min\_child\_weight', 'subsample', 'colsample\_bytree', 'gamma']

df\_results['mse'] = objective\_function(best\_params)

print("Best Parameters:", best\_params)

print("Best MSE:", df\_results['mse'].values[0])

# Plotting the results using Plotly

fig\_gwo = px.scatter\_matrix(

    df\_results,

    dimensions=['n\_estimators', 'learning\_rate', 'max\_depth', 'min\_child\_weight', 'gamma', 'mse'],

    color='mse',

    opacity=0.7,

    width=1600,

    height=800

)

# Update dot size, font size, and tick size

fig\_gwo.update\_traces(marker=dict(size=15))  # Increase dot size

fig\_gwo.update\_layout(font=dict(size=15))  # Increase font size

fig\_gwo.update\_layout(yaxis=dict(tickfont=dict(size=15)))  # Increase tick size for y-axis

fig\_gwo.update\_layout(xaxis=dict(tickfont=dict(size=15)))  # Increase tick size for x-axis

fig\_gwo.show()