**The code description for the establishment of three performance prediction models.**

Code execution tools: Jupyter

Programming language: Python 3.11.4

The packages used for code execution and their versions are shown in Table C1.

**Table C1 | The packages involved in code execution and their versions.**

|  |  |
| --- | --- |
| Package | Version |
| numpy | 1.24.0 |
| pandas | 1.5.3 |
| matplotlib | 3.7.1 |
| scikit-learn | 1.3.0 |
| xgboost | 1.7.3 |
| scipy | 1.10.1 |
| seaborn | 0.12.2 |
| shap | 0.46.0 |
| openpyxl | 3.0.10 |

**Note: Since the research is still in progress, the raw datasets involved in the code (supplemental Dataset\_UTS.xlsx, supplemental Dataset\_EL.xlsx, supplemental Dataset\_Hardness.xlsx) are temporarily not available to the public. If needed, you can contact the author.**

**The specific code is as follows, where the establishment of each performance prediction model takes UTS as an example. The process for building the EL and Hardness models is similar.**

**Part 1 of the code: Pearson and Spearman correlation coefficient.**

# *Import necessary packages*

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from matplotlib.patches import Wedge

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

# *Read Excel data*

**data = pd.read\_excel('all\_UTS.xlsx')** # *The "all\_UTS.xlsx" file is obtained by removing the last column "Ref." from the "supplemental Dataset\_UTS.xlsx" file. (When applied to EL and Hardness performance, change to "all\_EL" and "all\_Hardness." Specific classifications are available in the GitHub repository.)*

**train\_data, test\_data = train\_test\_split(data, test\_size=0.2, random\_state=4)** # *Split the dataset into training and test sets with a test set proportion of 0.2.*

# *Calculate the Pearson correlation coefficient / Spearman correlation coefficient matrix.*

**corr = train\_data.corr()** # *Perform Pearson correlation coefficient calculation on the training set. When performing Spearman's correlation calculation, replace it with “corr = train\_data.corr(method='spearman')*

# *Visualize the Pearson correlation coefficient / Spearman correlation coefficient calculation results.*

# *Create an empty plotting area.*

**fig, ax = plt.subplots(figsize=(12, 10))**

**ax.set\_xlim(0, corr.shape[1])**

**ax.set\_ylim(0, corr.shape[0])**

# *Set the y-axis label and remove the x-axis.*

**ax.set\_yticks(np.arange(0.5, len(corr.columns), 1))**

**ax.set\_yticklabels(corr.columns[::-1])**

**ax.set\_xticks([])**

# *Keep the upper triangular part of the correlation coefficient visualization results.*

**for i in range(corr.shape[0]):**

**for j in range(i + 1, corr.shape[1]):**

**value = corr.iloc[i, j]**

**size = abs(value)**

**if value > 0:**

**color = plt.cm.Blues(size) # Use blue for positive correlation.**

**else:**

**color = plt.cm.Reds(size) # Use red for negative correlation.**

# Draw a circular shape with a gray background to represent the absolute value.

**bg\_wedge = Wedge((j + 0.5, corr.shape[0] - i - 0.5), 0.4, 0, 360, facecolor='white', edgecolor='black', alpha=1)**

**ax.add\_patch(bg\_wedge)**

#*The size of the sectors (or wedges) represents the magnitude of the correlation coefficient value.*

**wedge = Wedge((j + 0.5, corr.shape[0] - i - 0.5), 0.4, 90, 90 + 360 \* size, facecolor=color,**

**edgecolor='black', alpha=1)**

**ax.add\_patch(wedge)**

#*The center of the circle displays the specific value of the correlation coefficient.*

**ax.text(j + 0.5, corr.shape[0] - i - 0.5, f'{value:.2f}', horizontalalignment='center', verticalalignment='center', fontsize=8)#**

# *Add a color bar.*

**norm = plt.Normalize(-1, 1)**

**cbar = plt.colorbar(plt.cm.ScalarMappable(norm=norm, cmap='coolwarm'), ax=ax, orientation='vertical')**

**cbar.set\_label('Pearson correlation coefficient')**

# *Add a title to the plot.*

**plt.title('Pearson Correlation Coefficient Plot (Upper Triangle)')**

# *Display the image.*

**plt.show()**

**Part 2 of the code: Random Forest Dual Indicator Repeat Sorting.**

# *Import necessary packages*

**import numpy as np**

**import pandas as pd**

**import matplotlib.pyplot as plt**

**from sklearn.ensemble import RandomForestRegressor**

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

**from sklearn.preprocessing import MinMaxScaler**

**from sklearn.inspection import permutation\_importance**

**import warnings**

**warnings.filterwarnings("ignore")**

# *Read Excel data*

**data = pd.read\_excel('all\_UTS.xlsx')** # *The "all\_UTS.xlsx" file is obtained by removing the last column "Ref." from the "supplemental Dataset\_UTS.xlsx" file. (When applied to EL and Hardness performance, change to "all\_EL" and "all\_Hardness." Specific classifications are available in the GitHub repository.)*

**train\_data, test\_data = train\_test\_split(data, test\_size=0.2, random\_state=4)** *# Split the dataset into training and test sets with a test set proportion of 0.2.*

**train\_data.fillna(0, inplace=True)** # *Translate: Fill missing values with 0.*

**X = train\_data.iloc[:, 1:]**

**Y = train\_data.iloc[:, 0]**

**column = train\_data.columns.tolist()[1:]** *# Get the names of the input features.*

**#** *Initialize an array to store the importance.*

**mdi\_importances = np.zeros((100, len(column)))**

**pi\_importances = np.zeros((100, len(column)))**

#*Calculate the feature importance for 100 random splits.*

**for seed in range(100):**

**X\_1 = MinMaxScaler().fit\_transform(X)** *# Normalized feature data*

**x\_train, x\_test, y\_train, y\_test = train\_test\_split(X\_1, Y, test\_size=0.2, random\_state=seed)**

*# Train a Random Forest model*

**RF = RandomForestRegressor()**

**RF.fit(x\_train, y\_train)**

*# Calculate MDI (Mean Decrease Impurity) feature importance*

**mdi\_importances[seed, :] = RF.feature\_importances\_**

*# Calculate PI (Permutation Importance) feature importance*

**result = permutation\_importance(RF, x\_test, y\_test, n\_repeats=30, random\_state=seed)**

**pi\_importances[seed, :] = result.importances\_mean**

*# Calculate the mean and standard deviation of MDI/PI*

**mdi\_mean = mdi\_importances.mean(axis=0)**

**mdi\_std = mdi\_importances.std(axis=0)**

**pi\_mean = pi\_importances.mean(axis=0)**

**pi\_std = pi\_importances.std(axis=0)**

*# Save the computation results*

**importance\_df = pd.DataFrame({**

**'Feature': column,**

**'MDI\_Mean': mdi\_mean,**

**'MDI\_Std': mdi\_std,**

**'PI\_Mean': pi\_mean,**

**'PI\_Std': pi\_std**

**})**

*# Create a DataFrame to store the results of each computation*

**all\_importances\_df = pd.DataFrame(mdi\_importances, columns=[f'MDI\_{col}' for col in column])**

**pi\_importances\_df = pd.DataFrame(pi\_importances, columns=[f'PI\_{col}' for col in column])**

**all\_importances\_df = pd.concat([all\_importances\_df, pi\_importances\_df], axis=1)**

*# Save to an Excel file*

**with pd.ExcelWriter('feature\_importances\_detailed.xlsx') as writer:**

**importance\_df.to\_excel(writer, sheet\_name='Summary', index=False)**

**all\_importances\_df.to\_excel(writer, sheet\_name='Detailed', index=False)**

*# Visualize the computation results*

**plt.figure(dpi=300, figsize=(14, 8))**

**plt.bar(np.arange(len(column)) - 0.2, mdi\_mean, yerr=mdi\_std, width=0.4, label='MDI')**

**plt.bar(np.arange(len(column)) + 0.2, pi\_mean, yerr=pi\_std, width=0.4, label='PI')**

**plt.xlabel('Features')**

**plt.ylabel('Importance')**

**plt.title('Feature Importances with MDI and PI')**

**plt.xticks(ticks=np.arange(len(column)), labels=column, rotation=90)**

**plt.legend()**

**plt.tight\_layout()**

**plt.show()**

**Part 3 of the code: Performance evaluation of different training ratios.**

# *Import necessary packages*

**import numpy as np**

**import pandas as pd**

**import matplotlib.pyplot as plt**

**from sklearn.linear\_model import LinearRegression, Ridge, Lasso**

**from sklearn.preprocessing import PolynomialFeatures**

**from sklearn.pipeline import Pipeline**

**from sklearn.neighbors import KNeighborsRegressor**

**from sklearn.ensemble import RandomForestRegressor, AdaBoostRegressor**

**import xgboost as xgb**

**from sklearn.gaussian\_process import GaussianProcessRegressor**

**from sklearn.gaussian\_process.kernels import RBF, ConstantKernel as C**

**from sklearn.model\_selection import GridSearchCV, train\_test\_split**

**from sklearn.metrics import mean\_squared\_error, r2\_score**

**import warnings**

**warnings.filterwarnings("ignore")**

# *Read Excel data*

**data = pd.read\_excel('UTS.xlsx')** # *The "UTS.xlsx" file is obtained by removing the columns corresponding to features* *that were eliminated based on correlation coefficients and random forest ranking from "all\_UTS.xlsx". (When applied to EL and Hardness performance, change to "EL" and "Hardness." Specific classifications are available in the GitHub repository.)*

**X = data.iloc[:, 1:]**

**Y = data.iloc[:, 0]**

*# Create an array to store the results.*

**results = []**

*# Define the range for generating the random seed.*

**seeds = range(100)**

*# Define the list of models.*

**models = {**

**'LinearRegression': LinearRegression(),**

**'ResponseSurface': Pipeline([**

**('poly', PolynomialFeatures(degree=2)),**

**('linear', LinearRegression())**

**]),**

**'Ridge': Ridge(),**

**'Lasso': Lasso(),**

**'KNN': KNeighborsRegressor(),**

**'RandomForest': RandomForestRegressor(n\_estimators=100),**

**'XGBoost': xgb.sklearn.XGBRegressor(learning\_rate=0.1, n\_estimators=100, max\_depth=3),**

**'GaussianProcess': GaussianProcessRegressor(kernel=C(0.1, (0.001, 0.1)) \* RBF(0.5, (1e-4, 10)), n\_restarts\_optimizer=10, alpha=0.1),**

**'AdaBoost': AdaBoostRegressor(n\_estimators=50, random\_state=42)**

**}**

*# Add simple hyperparameters and search ranges for each model.*

**param\_grids = {**

**'ResponseSurface': {**

**'poly\_\_degree': [2, 3, 4, 5]**

**},**

**'Ridge': {**

**'alpha': [0.1, 1.0, 10.0]**

**},**

**'Lasso': {**

**'alpha': [0.1, 1.0, 10.0]**

**},**

**'KNN': {'n\_neighbors': np.arange(1, 100)},**

**'RandomForest': {**

**'n\_estimators': [50, 75, 100],**

**'max\_features': ['auto', 'sqrt'],**

**'max\_depth': [3, 4, 5, 10],**

**'min\_samples\_split': [2, 5],**

**'min\_samples\_leaf': [1, 2],**

**'bootstrap': [True]**

**},**

**'XGBoost': {**

**'learning\_rate': [0.1],**

**'n\_estimators': [50, 75, 100],**

**'max\_depth': [1, 2, 3]**

**}**

**}**

*# Perform multiple random splits of the dataset.*

**for seed in seeds:**

**x\_train, x\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size=0.7, random\_state=seed)** *# Adjust the value of “test\_size” to change the split ratio between the training set and the test set. For example, test\_size=0.7 means that 70% of the data is used as the test set and 30% as the training set. By changing this parameter, the evaluation results for the model under different split ratios can be obtained.*

**for model\_name, model in models.items():**

*# Perform hyperparameter search for the model.*

**if model\_name in param\_grids:**

**grid = GridSearchCV(model, param\_grids[model\_name], cv=10, n\_jobs=-1)**

**model = grid.fit(x\_train, y\_train)**

**best\_params = model.best\_params\_**

**else:**

**model.fit(x\_train, y\_train)**

**best\_params = {}**

*# Evaluate the model performance.*

**y\_train\_pred = model.predict(x\_train)**

**y\_test\_pred = model.predict(x\_test)**

**train\_mse = mean\_squared\_error(y\_train, y\_train\_pred)**

**train\_rmse = np.sqrt(train\_mse)**

**train\_r2 = r2\_score(y\_train, y\_train\_pred)**

**test\_mse = mean\_squared\_error(y\_test, y\_test\_pred)**

**test\_rmse = np.sqrt(test\_mse)**

**test\_r2 = r2\_score(y\_test, y\_test\_pred)**

*# Save the evaluation results.*

**results.append({**

**'Seed': seed,**

**'Model': model\_name,**

**'Train\_RMSE': train\_rmse,**

**'Train\_R2': train\_r2,**

**'Test\_RMSE': test\_rmse,**

**'Test\_R2': test\_r2,**

**'Best\_Params': best\_params**

**})**

*# Store the results in an Excel file.*

**results\_df = pd.DataFrame(results)**

**results\_df.to\_excel('UTS\_model\_evaluation\_results\_0.3.xlsx', index=False)**

*# Calculate the mean and standard deviation of the evaluation results.*

**summary\_df = results\_df.groupby('Model').agg({**

**'Train\_R2': ['mean', 'std'],**

**'Test\_R2': ['mean', 'std']**

**}).reset\_index()**

**summary\_df.columns = ['Model', 'Train\_R2\_Mean', 'Train\_R2\_Std', 'Test\_R2\_Mean', 'Test\_R2\_Std']**

*# Output the mean and standard deviation of the evaluation results.*

**print(summary\_df)**

*# Visualize the results.*

**plt.figure(dpi=300, figsize=(14, 8))**

**plt.bar(np.arange(len(summary\_df)) - 0.2, summary\_df['Train\_R2\_Mean'], yerr=summary\_df['Train\_R2\_Std'], width=0.4, label='Train R²')**

**plt.bar(np.arange(len(summary\_df)) + 0.2, summary\_df['Test\_R2\_Mean'], yerr=summary\_df['Test\_R2\_Std'], width=0.4, label='Test R²')**

**plt.xlabel('Models')**

**plt.ylabel('R²')**

**plt.title('Model Evaluation: Train and Test R²')**

**plt.xticks(ticks=np.arange(len(summary\_df)), labels=summary\_df['Model'], rotation=45)**

**plt.legend()**

**plt.tight\_layout()**

**plt.show()**

**Part 4 of the code: Model Hyperparameter Optimization.**

**import numpy as np**

**import pandas as pd**

**import matplotlib.pyplot as plt**

**import seaborn as sns**

**from scipy import stats**

**import warnings**

**warnings.filterwarnings("ignore")**

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

**from sklearn.metrics import mean\_squared\_error, r2\_score**

**import xgboost as xgb**

**from bayes\_opt import BayesianOptimization**

# *Read Excel data*

**data = pd.read\_excel('UTS.xlsx')** # *The "UTS.xlsx" file is obtained by removing the columns corresponding to features* *that were eliminated based on correlation coefficients and random forest ranking from "all\_UTS.xlsx". (When applied to EL and Hardness performance, change to "EL" and "Hardness." Specific classifications are available in the GitHub repository.)*

*# Data Splitting*

**X = data.iloc[:, 1:]**

**Y = data.iloc[:, 0]**

**x\_train, x\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size=0.2, random\_state=4)**

*# Define Objective Function*

**def xgb\_evaluate(learning\_rate, n\_estimators, max\_depth, min\_child\_weight, subsample, colsample\_bytree, reg\_alpha, reg\_lambda):**

**params = {**

**'learning\_rate': learning\_rate,**

**'n\_estimators': int(n\_estimators),**

**'max\_depth': int(max\_depth),**

**'min\_child\_weight': min\_child\_weight,**

**'subsample': subsample,**

**'colsample\_bytree': colsample\_bytree,**

**'reg\_alpha': reg\_alpha,**

**'reg\_lambda': reg\_lambda,**

**'objective': 'reg:squarederror',**

**'eval\_metric': 'rmse'**

**}**

**xgb\_model = xgb.XGBRegressor(\*\*params)**

**xgb\_model.fit(x\_train, y\_train)**

**predictions = xgb\_model.predict(x\_test)**

**mse = mean\_squared\_error(y\_test, predictions)**

**return -np.sqrt(mse)**

*# Define Hyperparameter Range*

**param\_bounds = {**

**'learning\_rate': (0.01, 0.1),**

**'n\_estimators': (50, 300),**

**'max\_depth': (3, 10),**

**'min\_child\_weight': (1, 10),**

**'subsample': (0.5, 1.0),**

**'colsample\_bytree': (0.5, 1.0),**

**'reg\_alpha': (0, 1), # L1 正则化**

**'reg\_lambda': (0, 1) # L2 正则化**

**}**

*# Perform Bayesian Optimization*

**optimizer = BayesianOptimization(**

**f=xgb\_evaluate,**

**pbounds=param\_bounds,**

**random\_state=37,**

**verbose=2**

**)**

**optimizer.maximize(init\_points=20, n\_iter=50)** *# “init\_points” is the number of initial sample points, and “n\_iter” is the number of iterations.*

*# Obtain Best Parameters*

**best\_params = optimizer.max['params']**

**best\_params['n\_estimators'] = int(best\_params['n\_estimators'])**

**best\_params['max\_depth'] = int(best\_params['max\_depth'])**

*# Output Best Parameters*

**print("Best parameters found by Bayesian Optimization:")**

**for param, value in best\_params.items():**

**print(f"{param}: {value}")**

*# Train the Final Model Using the Best Parameters*

**xgb\_model = xgb.XGBRegressor(\*\*best\_params)**

**xgb\_model.fit(x\_train, y\_train)**

*# Output Model Evaluation Metrics*

**xgb\_MSE = mean\_squared\_error(y\_train, xgb\_model.predict(x\_train))**

**xgb\_RMSE = np.sqrt(xgb\_MSE)**

**xgb\_R2 = r2\_score(y\_train, xgb\_model.predict(x\_train))**

**print('RMSE\_train: ' + str(xgb\_RMSE))**

**print('R2\_train: ' + str(xgb\_R2))**

**xgb\_MSE\_t = mean\_squared\_error(y\_test, xgb\_model.predict(x\_test))**

**xgb\_RMSE\_t = np.sqrt(xgb\_MSE\_t)**

**xgb\_R2\_t = r2\_score(y\_test, xgb\_model.predict(x\_test))**

**print('RMSE\_test: ' + str(xgb\_RMSE\_t))**

**print('R2\_test: ' + str(xgb\_R2\_t))**

*# Visualize Model Goodness-of-Fit Plot*

**plt.figure(figsize=(5, 5))**

**plt.scatter(y\_train, xgb\_model.predict(x\_train), c="blue", label='training')**

**plt.scatter(y\_test, xgb\_model.predict(x\_test), c='red', label='test', marker='^')**

**plt.plot([min(y\_train), max(y\_train)], [min(y\_train), max(y\_train)], 'b--', lw=1)**

**plt.legend()**

**plt.xlabel('True Values')**

**plt.ylabel('Predictions')**

**plt.title('XGBoost Predictions')**

**plt.show()**

**Part 5 of the code: SHAP with PSO Integration.**

# *Import necessary packages*

**import numpy as np**

**import pandas as pd**

**from sklearn.preprocessing import MinMaxScaler**

**from scipy import stats**

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

**import xgboost as xgb**

**import matplotlib.pyplot as plt**

**import seaborn as sns**

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

**from sklearn.metrics import mean\_squared\_error, r2\_score**

**import warnings**

**warnings.filterwarnings("ignore")**

# *Read Excel data*

**data = pd.read\_excel('UTS.xlsx')** # *The "UTS.xlsx" file is obtained by removing the columns corresponding to features* *that were eliminated based on correlation coefficients and random forest ranking from "all\_UTS.xlsx". (When applied to EL and Hardness performance, change to "EL" and "Hardness." Specific classifications are available in the GitHub repository.)*

*#Split the data.*

**X = data.iloc[:, 1:]**

**Y = data.iloc[:, 0]**

**x\_train, x\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size=0.2, random\_state=4)**

*# Data normalization processing.*

**scaler = MinMaxScaler()**

**x\_train\_normalized = scaler.fit\_transform(x\_train)**

*# Model hyperparameter settings*

**best\_params = {**

**'colsample\_bytree': 1.0,**

**'learning\_rate': 0.1,**

**'max\_depth': 5,**

**'min\_child\_weight': 5.411729501146951,**

**'n\_estimators': 139,**

**'reg\_alpha': 1.0,**

**'reg\_lambda': 1.0,**

**'subsample': 0.5,**

**'objective': 'reg:squarederror',**

**'eval\_metric': 'rmse'**

**}**

*# Build the predictive model.*

**xgb\_model = xgb.XGBRegressor(\*\*best\_params)**

**xgb\_model.fit(x\_train, y\_train)**

*# Output the model's performance metrics to ensure the correctness of the model establishment.*

**xgb\_MSE = mean\_squared\_error(y\_train, xgb\_model.predict(x\_train))**

**xgb\_RMSE = np.sqrt(xgb\_MSE)**

**xgb\_R2 = r2\_score(y\_train, xgb\_model.predict(x\_train))**

**print('RMSE\_train: ' + str(xgb\_RMSE))**

**print('R2\_train: ' + str(xgb\_R2))**

**xgb\_MSE\_t = mean\_squared\_error(y\_test, xgb\_model.predict(x\_test))**

**xgb\_RMSE\_t = np.sqrt(xgb\_MSE\_t)**

**xgb\_R2\_t = r2\_score(y\_test, xgb\_model.predict(x\_test))**

**print('RMSE\_test: ' + str(xgb\_RMSE\_t))**

**print('R2\_test: ' + str(xgb\_R2\_t))**

*# Calculate the central tendency of the samples in the original dataset.*

**def compute\_central\_location(X):**

**return np.mean(X, axis=0)**

*# Calculate the skewness (left or right) based on the original dataset.*

**def compute\_skewness(X, CL):**

**N\_L = np.sum(X < CL, axis=0)**

**N\_U = np.sum(X > CL, axis=0)**

**skewness\_L = N\_L / (N\_L + N\_U + 1)**

**skewness\_U = N\_U / (N\_L + N\_U + 1)**

**return skewness\_L, skewness\_U**

*# Calculate the range of the asymmetric acceptance domain.*

**def compute\_asymmetric\_range(X, CL, skewness\_L, skewness\_U):**

**LB = CL - (1 / skewness\_U) \* (CL - np.min(X, axis=0))**

**UB = CL + (1 / skewness\_L) \* (np.max(X, axis=0) - CL)**

**return LB, UB**

*# The PSO algorithm generates virtual input features.*

**def generate\_virtual\_samples(X, num\_samples=100, num\_particles=30, num\_iterations=100):**

**CL = compute\_central\_location(X)**

**skewness\_L, skewness\_U = compute\_skewness(X, CL)**

**LB, UB = compute\_asymmetric\_range(X, CL, skewness\_L, skewness\_U)**

**all\_samples = []**

**while len(all\_samples) < num\_samples:**

*# PSO algorithm initialization settings.*

**particles = np.random.uniform(LB, UB, (num\_particles, X.shape[1]))** *# Initialize particle positions.*

**velocities = np.zeros\_like(particles)** *# Initialize particle velocity.*

**personal\_best = particles.copy()** *# Initialize personal best position.*

**global\_best = particles[np.argmin(np.std(particles, axis=1))]** *# Initialize global best position.*

*# Iteratively update the velocity and position of the new particles.*

**for \_ in range(num\_iterations):**

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

**velocities = 0.7 \* velocities + 1.5 \* r1 \* (personal\_best - particles) + 1.5 \* r2 \* (global\_best - particles)**

**particles = particles + velocities**

**particles = np.clip(particles, LB, UB)**

**particles = np.clip(particles, 0, None)**

**current\_best = particles[np.argmin(np.std(particles, axis=1))]**

**if np.std(current\_best) < np.std(global\_best):**

**global\_best = current\_best**

*# Add each particle to the 'all\_samples' list, ensuring that there are no duplicate virtual samples.*

**for particle in particles:**

**if len(all\_samples) < num\_samples and not any(np.array\_equal(particle, s) for s in all\_samples):**

**all\_samples.append(particle)**

*# Convert all the generated virtual samples into an array*

**virtual\_samples = np.array(all\_samples)**

**print(f"Generated virtual samples shape: {virtual\_samples.shape}")** *# Output iteration information.*

**return virtual\_samples**

*# Generate virtual samples*

**virtual\_features\_normalized = generate\_virtual\_samples(x\_train\_normalized, num\_samples=200)** # The parameter "num\_samples" is the required number of virtual samples

*# Anti-normalization processing*

**virtual\_features = scaler.inverse\_transform(virtual\_features\_normalized)**

*# Ensure that the shape of the virtual input feature is consistent with the shape of the training set feature*

**assert virtual\_features.shape[1] == x\_train.shape[1],**

*# Predict the output label of a virtual sample*

**virtual\_labels = xgb\_model.predict(virtual\_features)**

*# Outlier detection with Z-Score*

**z\_scores = np.abs(stats.zscore(virtual\_features, axis=0))**

**outliers = (z\_scores > 3).any(axis=1)**

**virtual\_features\_filtered = virtual\_features[~outliers]**

**virtual\_labels\_filtered = virtual\_labels[~outliers]**

*# Export virtual sample data as an excel table*

**virtual\_data = pd.DataFrame(virtual\_features\_filtered, columns=X.columns)**

**virtual\_data['Predicted\_Output'] = virtual\_labels\_filtered**

**virtual\_data.to\_excel('uts\_virtual\_samples\_filtered.xlsx', index=False)**

*# Perform SHAP analysis*

*# Read in data*

**data = pd.read\_excel('uts\_virtual\_samples\_filtered.xlsx')**

*# Normalization treatment*

**scaler = MinMaxScaler()**

**data\_normalized = scaler.fit\_transform(data)**

*# Import SHAP package*

**import shap**

*# Data Splitting*

**X = data\_normalized.iloc[:, 1:]**

**Y = data\_normalized.iloc[:, 0]**

**x\_train, x\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size=0.2, random\_state=4)**

*# Hyperparameter Settings*

**best\_params = {**

**'colsample\_bytree': 1.0,**

**'learning\_rate': 0.1,**

**'max\_depth': 5,**

**'min\_child\_weight': 5.411729501146951,**

**'n\_estimators': 139,**

**'reg\_alpha': 1.0,**

**'reg\_lambda': 1.0,**

**'subsample': 0.5,**

**'objective': 'reg:squarederror',**

**'eval\_metric': 'rmse'**

**}**

*# Model Construction*

**xgb\_model = xgb.XGBRegressor(\*\*best\_params)**

**xgb\_model.fit(x\_train, y\_train)**

*# Calculate SHAP Values*

**explainer = shap.Explainer(xgb\_model)**

**shap\_values = explainer(x\_test)**

*# Initialize JavaScript Library*

**shap.initjs()**

*# Plot Feature Importance Ranking Bar Chart*

**shap.plots.bar(shap\_values)**

*# Plot SHAP Force Plot Using Matplotlib*

**plt.figure()**

**shap\_values\_sample = shap\_values[0]**

**feature\_importance = np.abs(shap\_values\_sample.values).argsort()[::-1] 序特征**

**top\_features = feature\_importance[:8]**

**shap.force\_plot(explainer.expected\_value, shap\_values\_sample.values[top\_features], x\_test.iloc[0, top\_features], matplotlib=True)**

**plt.show()**

*# SHAP Value Summary Plot*

**shap.summary\_plot(shap\_values, x\_test, plot\_type="bar")**

*# SHAP summary plot with distribution*

**shap.summary\_plot(shap\_values, x\_test)**

**Part 6 of the code: Multi-modal materials inverse design framework.**

*# Import necessary packages*

**import numpy as np**

**import pandas as pd**

**import matplotlib.pyplot as plt**

**import seaborn as sns**

**from scipy import stats**

**import os**

**import warnings**

**from bayes\_opt import BayesianOptimization**

**import xgboost as xgb**

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

**from sklearn.metrics import mean\_squared\_error, r2\_score**

**from sklearn.ensemble import RandomForestRegressor**

**from openpyxl import Workbook**

**warnings.filterwarnings("ignore")**

*#Build a UTS prediction model.*

*# read the data*

**data = pd.read\_excel('UTS.xlsx')**

*# Data splitting.*

**X = data.iloc[:, 1:]**

**Y = data.iloc[:, 0]**

**x\_train, x\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size=0.2, random\_state=4)**

*#Hyperparameter settings.*

**best\_params = {**

**'colsample\_bytree': 1.0,**

**'learning\_rate': 0.1,**

**'max\_depth': 8,**

**'min\_child\_weight': 10.0,**

**'n\_estimators': 113,**

**'reg\_alpha': 0.0,**

**'reg\_lambda': 1.0,**

**'subsample': 1.0,**

**'objective': 'reg:squarederror',**

**'eval\_metric': 'rmse'**

**}**

*#Build the model.*

**UTS\_model = xgb.XGBRegressor(\*\*best\_params)**

**UTS\_model.fit (x\_train, y\_train)**

*#Build a EL prediction model.*

*# read the data*

**data = pd.read\_excel('EL.xlsx')**

*# Data splitting.*

**X = data.iloc[:, 1:]**

**Y = data.iloc[:, 0]**

**x\_train, x\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size=0.2, random\_state=5)**

*#Hyperparameter settings.*

**best\_params = {**

**'bootstrap': False,**

**'max\_depth': 10,**

**'max\_features': 0.5,**

**'min\_impurity\_decrease': 0.0,**

**'min\_samples\_leaf': 1,**

**'min\_samples\_split': 5,**

**'n\_estimators': 55**

**}**

*#Build the model.*

**EL\_model = RandomForestRegressor(\*\*best\_params, random\_state=25)**

**EL\_model.fit (x\_train, y\_train)**

*#Build a Hardness prediction model.*

*# read the data*

**data = pd.read\_excel('Hardness.xlsx')**

*# Data splitting.*

**X = data.iloc[:, 1:]**

**Y = data.iloc[:, 0]**

**x\_train, x\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size=0.2, random\_state=8)**

*#Hyperparameter settings.*

**best\_params = {**

**'colsample\_bytree': 1.0,**

**'learning\_rate': 0.07141712725507017,**

**'max\_depth': 3,**

**'min\_child\_weight': 1.036994365622379,**

**'n\_estimators': 179,**

**'reg\_alpha': 0.07844930740270828,**

**'reg\_lambda': 0.4659923785137676,**

**'subsample': 0.5124079817369307,**

**'objective': 'reg:squarederror',**

**'eval\_metric': 'rmse'**

**}**

*#Build the model.*

**Hardness\_model = xgb.XGBRegressor(\*\*best\_params)**

**Hardness\_model.fit(x\_train, y\_train)**

*# Multi-modal materials inverse design framework.*

*# Set the target value.*

**target\_UTS = 300 #** *Set the target UTS value.*

**target\_EL = 3.5 #** *Set the target EL value.*

**target\_Hardness = 130 #** *Set the target Hardness value.*

*# Set the weights. Set the range to 0-1.* *The larger the weight corresponding to the performance, the more focus is placed on that particular performance.*

**weight\_UTS = 1**

**weight\_EL = 1**

**weight\_Hardness = 1**

*# Define the objective function.*

**def target\_function(\*\*params):**

**params\_df = pd.DataFrame([params], columns=X.columns) #**

**pred\_UTS = UTS\_model.predict(params\_df)[0] + UTS\_RMSE\_t**

**pred\_EL = EL\_model.predict(params\_df)[0] + 2 \* EL\_RMSE\_t**

**pred\_Hardness = Hardness\_model.predict(params\_df)[0] + Hardness\_RMSE\_t**

*# Calculate the target value: the sum of the ratios between the three predicted performances and the target performance.*

**ratio\_UTS = pred\_UTS / target\_UTS**

**ratio\_EL = pred\_EL / target\_EL**

**ratio\_Hardness = pred\_Hardness / target\_Hardness**

**score = weight\_UTS \* abs(ratio\_UTS - 1) + weight\_EL \* abs(ratio\_EL - 1) + weight\_Hardness \* abs(ratio\_Hardness - 1)**

*# Save the results to an Excel sheet.*

**iteration\_result = {**

**'Al': params.get('Al', None),**

**'Cu': params.get('Cu', None),**

**'Mg': params.get('Mg', None),**

**'Ti': params.get('Ti', None),**

**'Mn': params.get('Mn', None),**

**'Si': params.get('Si', None),**

**'La': params.get('La', None),**

**'Ce': params.get('Ce', None),**

**'pred\_UTS': pred\_UTS,**

**'pred\_EL': pred\_EL,**

**'pred\_Hardness': pred\_Hardness,**

**'score': -score**

**}**

**save\_iteration\_to\_excel(iteration\_result)**

**return -score** *# The optimizer will maximize the objective function, so here, we return the negative value to minimize the difference.*

*# Define the allowable range for the features.*

**pbounds = {**

**'Al': (3, 8), #** *The range for aluminum is from 3 to 8.*

**'Cu': (0, 6), *#*** *The range for copper is from 0 to 6.*

**'Mg': (0.0, 0.05), #** *The range for magnesium is from 0 to 0.05.*

**'Ti': (0.0, 0.0),** *# The range for titanium is fixed at 0.*

**'Mn': (0.0, 0.0),** *# The range for manganese is fixed at 0.*

**'Si': (0.0, 0.0),** *# The range for silicon is fixed at 0.*

**'La': (0.0, 0.0), #** *The range for lanthanum is fixed at 0.*

**'Ce': (0.0, 0.0) #** *The range for cerium is fixed at 0.*

**}**

*# Initialize the Excel file.*

**def initialize\_excel(file\_name):**

**if not os.path.exists(file\_name):**

**wb = Workbook()**

**ws = wb.active**

**ws.title = "Results"**

**ws.append(['Al', 'Cu', 'Mg', 'Ti', 'Mn', 'Si', 'La', 'Ce', 'pred\_UTS', 'pred\_EL', 'pred\_Hardness', 'score'])**

**wb.save(file\_name)**

*# Save the results of each iteration to the Excel file.*

**def save\_iteration\_to\_excel(result):**

**file\_name = "optimization\_results.xlsx"**

**initialize\_excel(file\_name)**

**df = pd.DataFrame([result])**

**with pd.ExcelWriter(file\_name, mode='a', if\_sheet\_exists='overlay', engine='openpyxl') as writer:**

**df.to\_excel(writer, sheet\_name="Results", index=False, header=False, startrow=writer.sheets['Results'].max\_row)**

*# Initialize Excel.*

**initialize\_excel("optimization\_results.xlsx")**

*# Set the Bayesian optimization parameters.*

**optimizer = BayesianOptimization(**

**f=target\_function,**

**pbounds=pbounds,**

**random\_state=1**

**)**

*# Maximum number of iterations.*

**n\_iter = 300**

*# Bayesian optimization.*

**optimizer.maximize(init\_points=50, n\_iter=n\_iter)**

*# Obtain the optimal feature parameters.*

**best\_params = optimizer.max['params']**

**best\_params\_df = pd.DataFrame([best\_params], columns=X.columns) #** *Ensure the consistency of the feature order.*

**best\_pred\_UTS = UTS\_model.predict(best\_params\_df)[0] + UTS\_RMSE\_t**

**best\_pred\_EL = EL\_model.predict(best\_params\_df)[0] + 2 \* EL\_RMSE\_t**

**best\_pred\_Hardness = Hardness\_model.predict(best\_params\_df)[0] + Hardness\_RMSE\_t**

*# Set the tolerance for error.*

**tolerance\_UTS = 20**

**tolerance\_EL = 0.5**

**tolerance\_Hardness = 10**

*# Check if it is within the tolerance range.*

**if (abs(best\_pred\_UTS - target\_UTS) <= tolerance\_UTS and**

**abs(best\_pred\_EL - target\_EL) <= tolerance\_EL and**

**abs(best\_pred\_Hardness - target\_Hardness) <= tolerance\_Hardness):**

**print(f"** **The optimal parameters found: {best\_params}")**

**print(f"** **Predicted UTS value: {best\_pred\_UTS}")**

**print(f"Predicted EL value: {best\_pred\_EL}")**

**print(f"Predicted Hardness value: {best\_pred\_Hardness}")**

**else:**

**print("No solution meeting the criteria was found within the specified number of iterations.")**

**print("The closest solution is:")**

**print(f"Parameters: {best\_params}")**

**print(f" Predicted UTS value: {best\_pred\_UTS}")**

**print(f" Predicted EL value: {best\_pred\_EL}")**

**print(f" Predicted Hardness value: {best\_pred\_Hardness}")**