Code Explanation of paper titled "*Machine Learning Based Insights for Metal-Organic Frameworks Synthesis: A Comparative and Explainable Analysis of ZIF-8 Morphology*" by Du Y *et al.*, on *Materials Today Communications*

1. Overview

This is the code generated from our experimental and computational integrated study that investigated the synthesis of ZIF-8 (a type of zeolitic imidazolate framework) by varying three key experimental parameters: the molar ratio of precursors, temperature, and solvent properties. For the detailed experiment procedures, please refer to the paper.

2. Data Preparation

The data shared on Zenodo include the synthesis conditions and the characterized sizes of ZIF-8 from our experiments. The ZIF-8 size was estimated from transmission electro microscopy (TEM) images using ImageJ. For access to the raw imaging data, please contact the corresponding author. To ensure model accuracy, data augmentation techniques will be applied based on the estimated mean size and standard deviation of the sizes under individual conditions.

3. Major Components of the Code

3.1 Data splitting

The following code is used to divide data into training and testing datasets for developing machine learning models. It is important to note that data (or feature) normalization is not used in our work, but these commands are provided here for these interested in testing the effect of normalization on model accuracy. The "DependentVariable" in the dataset should be adjusted to meet the modeling objective. For example, it should be changed to "size" if the objective is to identify how synthesis conditions affect ZIF-8 size, while it should be "variations" when the goal is to quantify how these conditions introduce variations in the size of ZIF-8 crystals.

```
# Set random seeds for reproducibility
random_seed = 42
np.random.seed(random_seed)
tf.random.set_seed(random_seed)

# Extract actual data of input varialbes
```

```
Xall = pd.read_excel('data_zif8.xlsx')
# print(Xall.columns)

X = Xall[['Hmim', 'Temp', 'Viscosity', 'Density', 'Polarity']]

'''
# Normalize the data
scaler = StandardScaler()
X = scaler.fit_transform(X)
'''

# Extract the dependent variable
y = Xall['DependentVariable']

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=random_seed)
```

3.2 Hyperparameter optimization

The selection of hyperparameters can affect model performance, thus the following code use the identification of the parameters for the random forest model as an example. Specifically, a grid search method is used here.

In this work, only five hyperparameters are optimized, but the code can be modified and expanded to other parameters for improved accuracy. Also, hyperparameter domain can be also changed, as indicated in the comments in the code below.

```
# Optimize hyperparameters of the Random forest model

from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.inspection import partial_dependence
from sklearn.inspection import PartialDependenceDisplay
from sklearn.utils import resample

# Create a Random Forest Regressor
rf_model = RandomForestRegressor()

# Define the parameter grid for GridSearchCV
# This set of parameters try smaller values for each
param_grid = {
    'n_estimators': [5, 10, 15, 20, 25, 30],  # Adjust as needed
    'max_depth': [None, 5, 10, 20],  # Adjust as needed
```

```
    'min_samples_split': [2, 3, 5], # [2, 5, 10]
    'min_samples_leaf': [1, 2, 4],
    'bootstrap': [True, False]
}

# Create GridSearchCV object
grid_search = GridSearchCV(rf_model, param_grid, cv=5, scoring='r2', verbose = 1)

# Fit the model to the data
grid_search.fit(X_train, y_train)

# Get the best hyperparameters
best_params = grid_search.best_params_
best_rf_model = grid_search.best_estimator_

print(f"Best hyperparameters: {best_params}")
```

3.3 Evaluation of model performance

Based on the optimized model, such as the random forest model, the following code can be used to calculate the *mean squared error* (*MSE*) and R^2 for both training and testing datasets. The code can be also modified for testing the effect of non-optimal parameters on model prediction. For example, the “best_max_features” is commented out in the code as a demonstration for such testing for these interested.

```
# Use the best hyperparameters obtained from the grid search

best_n_estimators = best_params['n_estimators']
best_max_depth = best_params['max_depth']
best_min_samples_split = best_params['min_samples_split']
best_min_samples_leaf = best_params['min_samples_leaf']
#best_max_features = best_params['max_features']
best_bootstrap = best_params['bootstrap']

# Create a new instance of RandomForestRegressor with the best hyperparameters
best_rf_model = RandomForestRegressor(
    n_estimators=best_n_estimators,
    max_depth=best_max_depth,
    min_samples_split=best_min_samples_split,
    min_samples_leaf=best_min_samples_leaf,
    # max_features=best_max_features,
    max_features=1.0,
    bootstrap=best_bootstrap
```

```
)

# Fit the model to the training data
best_rf_model.fit(X_train, y_train)

# Make predictions on training data
y_pred_rfT = best_rf_model.predict(X_train)

# Evaluate the model on testing data
r_squaredT = r2_score(y_train, y_pred_rfT)
mseT = mean_squared_error(y_train, y_pred_rfT)

print(f"r_squared on training data: {r_squaredT}")
print(f"MSE on training data: {mseT}")

# Make predictions on testing data
y_pred_rf = best_rf_model.predict(X_test)

# Evaluate the model on testing data
r_squared = r2_score(y_test, y_pred_rf)
mse = mean_squared_error(y_test, y_pred_rf)

print(f"r_squared on testing data: {r_squared}")
print(f"MSE on testing data: {mse}")
```

3.4 Testing to demonstrate the effect of hyperparameters on model performance

While there are multiple hyperparameters for each of the model considered in this work, three case studies are conducted to demonstrate the effect of them on model accuracy by using a heatmap figures. The code use R^2 as an evaluation metric to see how different combinations of hyperparameters for each model affect the results. Adjusting parameter ranges can be also conducted to identify the optimal domain for the grid search as shown in the code above. In addition of using the R^2 as an evaluation criterion, the MSE might be used, but this is not investigated in our work, which can be tested for those interested in the performance.

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV

# Create a Random Forest Regressor
```

```
rf_model = RandomForestRegressor()

# Define the parameter grid for GridSearchCV
param_grid = {
    'n_estimators': [5, 10, 15, 20, 25, 30],
    'max_depth': [None, 5, 10, 20],
    'min_samples_split': [2, 3, 5],
    'min_samples_leaf': [1, 2, 4],
    'bootstrap': [True, False]
}

# Create GridSearchCV object
grid_search = GridSearchCV(rf_model, param_grid, cv=5, scoring='r2', verbose=1)

# Fit the model to the data
grid_search.fit(X_train, y_train)

# Extract results into a DataFrame
results = pd.DataFrame(grid_search.cv_results_)

# Create a pivot table for heatmap visualization
heatmap_data = results.pivot_table(
    index='param_min_samples_leaf',
    columns='param_n_estimators',
    values='mean_test_score'
)

# Plotting the heatmap
plt.figure(figsize=(5, 3))
sns.heatmap(heatmap_data, annot=True, cmap='viridis', fmt='.3f', cbar_kws={'label': 'R2'})
plt.title('Grid Search Mean Test Score for Different Hyperparameters')
plt.xlabel('Number of Trees')
plt.ylabel('Minimum Samples Leaf')
plt.show()

# Create a pivot table for heatmap visualization
heatmap_data = results.pivot_table(
    index='param_min_samples_split',
    columns='param_n_estimators',
    values='mean_test_score'
)

# Plotting the heatmap
plt.figure(figsize=(5, 3))
sns.heatmap(heatmap_data, annot=True, cmap='viridis', fmt='.3f', cbar_kws={'label': 'R2'})
```

```
plt.title('Grid Search Mean Test Score for Different Hyperparameters')
plt.xlabel('Number of Tress')
plt.ylabel('min_samples_split')
plt.show()


# Get the best hyperparameters
best_params = grid_search.best_params_
best_rf_model = grid_search.best_estimator_

print(f"Best hyperparameters: {best_params}")
```

3.5 Figures to visualize model performance

Once the model is calibrated, the following code can be used to visualize the predicted vs. actual ZIF-8 for comparison. The scatter plots for both the training and testing data of the random forest model are used as a demo in the code below. The presentation can be adjusted. For example, the font size and style of figure caption can be changed as indicated in these commands. The type of marker used in the figure can be also modified.

```
# Create a scatter plot for random forest results
import matplotlib.pyplot as plt

# Create a figure with a specified size
plt.figure(figsize=(3, 2))  # Adjust the width and height as needed

plt.scatter(y_train, y_pred_rfT, alpha=0.5, s = 100, c = 'red', marker = '^')
plt.scatter(y_test, y_pred_rf, alpha=0.5, s = 100, c = 'blue', marker = 'o')

# Plot the perfect prediction line (y = x)
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], '--r')

# Set labels and title
plt.xlabel('Actual Data', fontsize = 12, fontweight='bold')
plt.ylabel('Model Predictions', fontsize = 12, fontweight='bold')
plt.title('Actual vs Predicted Data (Random forest)', fontsize = 12, fontweight='bold')

# Customize x and y-axis font size and style
#plt.xticks(fontsize=10, fontstyle='italic')  # Set font size and style for x-axis ticks
plt.xticks(fontsize=12, fontweight='bold')  # Set font size and style for x-axis ticks
plt.yticks(fontsize=12, fontweight='bold')   # Set font size and style for y-axis ticks

# Show the plot
```

```
plt.show()
```

3.6 Ranking the role of individual synthesis conditions in determining ZIF-8 size

The Shapley Additive exPlanations (SHAP) analysis is used in this work and below the random forest model is used as an example for explanation. Importantly, it may be required to install "shap" first before running the code, depending on the version of Python. For installation, it is required to comment out command "# !pip install shap" by removing "#" at the beginning.

Additionally, the name of the model should be adjusted for different models that are being tested.

```
# SHapley Additive exPlanations (SHAP) with randome forest models
# If shap is not installed, run the first command below first and then comment it out
# !pip install shap

import shap
import matplotlib.pyplot as plt
import pandas as pd

# Custom feature names
feature_names = ['Hmim', 'Temp', 'Viscosity', 'Density', 'Polarity']

# Initialize SHAP explainer
explainer = shap.Explainer(best_rf_model.predict, X)
shap_values = explainer(X)

# Summary plot with custom feature names
shap.summary_plot(shap_values, X, feature_names=feature_names)
```

3.7 Visualization using Violin plots

Violin plots are used in this work to compare model performance and below is the code that use the random forest model as an example. For testing other models, the name should be changed. It is possible to adjust the number of trials and the splitting of training and testing datasets. The code below also exerts hard constraints for R^2 values; however, it is not required and these commands can be commented out.

```
# Generate data for Violin plots -- Random forest regression.

# Initialize lists to store performance metrics
mse_scores = []
```

```
r2_scores = []

# Perform multiple trials
n_trials = 500

# Perform trials until we collect the required number of positive scores
while len(mse_scores) < n_trials:

    # Split the data into training and testing sets for each trial
    X_train_trial, X_test_trial, y_train_trial, y_test_trial = train_test_split(X_train, y_train, test_size=0.3,
random_state=np.random.randint(0, 10000))

    # Train the model
    best_rf_model.fit(X_train_trial, y_train_trial)

    # Make predictions
    y_pred_train_trial = best_rf_model.predict(X_train_trial)
    y_pred_test_trial = best_rf_model.predict(X_test_trial)

    # Evaluate the model
    mse_train = mean_squared_error(y_train_trial, y_pred_train_trial)
    r2_train = r2_score(y_train_trial, y_pred_train_trial)
    mse_test = mean_squared_error(y_test_trial, y_pred_test_trial)
    r2_test = r2_score(y_test_trial, y_pred_test_trial)

    # Save only positive scores
    if mse_train > 0 and mse_test > 0 and r2_train > 0 and r2_test > 0:
        mse_scores.append([mse_train, mse_test])
        r2_scores.append([r2_train, r2_test])

# Convert lists to numpy arrays for easier plotting
mse_scores_rf = np.array(mse_scores)
r2_scores_rf = np.array(r2_scores)

# Create violin plots for MSE and R-squared scores
plt.figure(figsize=(5, 4))

# MSE distribution plot
plt.subplot(1, 2, 1)
plt.violinplot([mse_scores_rf[:, 0], mse_scores_rf[:, 1]], showmeans=True, showmedians=True)
plt.xticks([1, 2], ['Training MSE', 'Testing MSE'])
plt.ylabel('Mean Squared Error')
plt.title('Distribution of MSE')

# R-squared distribution plot
```

```
plt.subplot(1, 2, 2)
plt.violinplot([r2_scores_rf[:, 0], r2_scores_rf[:, 1]], showmeans=True, showmedians=True)
plt.xticks([1, 2], ['Training R2', 'Testing R2'])
plt.ylabel('R-Squared')
plt.title('Distribution of R-Squared')

plt.tight_layout()
plt.show()

print(f'The total number of runs: len(mse_scores)')
print(len(mse_scores))
```

3.8 Training of the neural network model

Training the neural network model requires additional library for hyperparameter optimization. Thus, the following code should be executed first, followed by the other parts to visualization and analysis as shared on GitHub.

The rest components for both the neural network and support vector regression models are similar to the development of the random forest model, following the similar adjustment in the codes as discussed above.

```
# Install Keras-tuner for hyperparameter tuning
!pip install keras-tuner
```

3.9 Predictive analysis using machine learning models

When model calibration is complete, it is possible to use them for predictive and explainable analysis. Since the neural network model provides better performance, it is chosen in our paper. Below is an example of the code to demonstrate the effect of Hmim concentration and polarity on ZIF-8 size by fixing the other synthesis conditions at their corresponding mean values.

The similar relationships for other synthesis conditions can be visually generated, which are included in our code. Of note, our paper only contains several different combinations between these synthesis conditions, but the code has included additional ones and their corresponding results.

```
# Using mean values of factors that are not considered for 2D plot -- neural network model

import numpy as np
import pandas as pd
```

```
import matplotlib.pyplot as plt

data = pd.read_excel('data_zif8.xlsx')

# Relationship between Factor1 (Hmim) and Factor5 (polarity)
# Create a grid for Factor1 and Factor5
factor1Range = np.linspace(np.min(data['Hmim']), np.max(data['Hmim']), 20)
factor5Range = np.linspace(np.min(data['Polarity']), np.max(data['Polarity']), 20)

factor1Grid, factor5Grid = np.meshgrid(factor1Range, factor5Range)

# Create a grid of input data for prediction
inputGrid = pd.DataFrame({
    'Hmim': factor1Grid.flatten(),
    'Temp': np.ones_like(factor1Grid.flatten()) * np.mean(data['Temp']),
    'Viscosity': np.ones_like(factor1Grid.flatten()) * np.mean(data['Viscosity']),
    'Density': np.ones_like(factor1Grid.flatten()) * np.mean(data['Density']),
    'Polarity': factor5Grid.flatten()
})

# Make predictions on the grid
predictions = nn_model.predict(inputGrid)

# Reshape predictions to match the grid dimensions
predictionsGrid = predictions.reshape(factor1Grid.shape)

# Create a 2D contour plot
plt.figure()

contourf = plt.contourf(factor1Grid, factor5Grid, predictionsGrid, 20, cmap='plasma')

# Add contour lines with custom line styles
contour = plt.contour(factor1Grid, factor5Grid, predictionsGrid, 20, colors='white', linestyles='dashed')

# Customize font and size for title, x-axis label, and y-axis label
plt.title('Contour Plot of Dependent Variable', fontdict={'fontsize': 12, 'fontweight': 'bold', 'fontfamily': 'sans-serif'})
plt.xlabel('Hmim', fontdict={'fontsize': 12, 'fontweight': 'bold', 'fontfamily': 'serif'})
plt.ylabel('Polarity', fontdict={'fontsize': 12, 'fontweight': 'bold', 'fontfamily': 'serif'})

# Show the colorbar
plt.colorbar(contourf)

# Customize font and size for x-axis and y-axis numbers
plt.xticks(fontsize=12, fontweight='bold', fontfamily='serif')
plt.yticks(fontsize=12, fontweight='bold', fontfamily='serif')
```

```
# Show the plot
plt.show()
```

3.10 Closing remarks

We do not anticipate any difficulties running the code as provided; however, users may need to fine-tune the program slightly due to differences in Python environments or compilers. Comments are also included in the code for explaining each command when changes are needed for specific objectives. Due to potential variations in coding compliers, the results may change slightly, such as model hyperparameters; however, we do not anticipate that affects the conclusions and analysis significantly.