PROGRAMMING IN PYTHON FOR BUSINESS AND LIFE SCIENCE ANALYTICS

2024 SS

A report on

MGT001437_ProjectC
"Biomass microwave pyrolysis characterization by machine learning for sustainable rural biorefineries"

Submitted by

GROUP 4

Jitesh Pravin Mhatre - 03771319
Vincent Kellerer - 03741768
Yunchung Ma - 03750836

Under Guidance of

Yanfei Shan

28th July, 2024

Table of Contents

List of Tables

List of Figures

Abstract

Microwave pyrolysis is a promising technique for converting plastic waste into synthetic gas, bio-oil, and biochar, offering a solution to plastic pollution and generating fuels with lower CO2 footprints than those derived from crude oil. This report explores microwave pyrolysis, focusing on reproducing and extending a study by Yang et al. by applying machine learning techniques to analyze various parameters affecting pyrolysis outcomes. We compare the performance of different machine learning models in predicting these outcomes, highlighting the potential of combining microwave pyrolysis with machine learning for effective plastic waste management.

1. Introduction

Plastic waste is a growing environmental concern due to its non-biodegradable nature and widespread use in various industries. Traditional disposal methods, such as landfilling and incineration, have significant ecological impacts. Landfilling can lead to soil and water contamination due to the leaching of harmful chemicals from the plastic waste, while incineration contributes to greenhouse gas emissions. Consequently, there is an urgent need for innovative and sustainable approaches to managing plastic waste.

Microwave pyrolysis, an emerging technology, utilizes microwave radiation to heat and decompose organic materials in an oxygen-depleted environment. This process can transform plastic waste into valuable products like synthetic gas (syngas), bio-oil, and biochar. These products can serve as alternative fuels or raw materials for chemical synthesis, offering a twofold advantage of reducing waste and recovering resources.

Microwave pyrolysis has yet to be widely adopted despite its potential, primarily due to its high energy consumption and unpredictable results, which affect its economic viability. However, recent advancements in machine learning (ML) offer new opportunities to enhance the efficiency and predictability of microwave pyrolysis. By analyzing large datasets of experimental results, ML can identify patterns and optimize process parameters to improve yield and reduce energy consumption. This could lead to significant cost savings and make microwave pyrolysis more attractive for plastic waste management.

A recent study by Yang et al. demonstrated the application of ML in analyzing various documented microwave pyrolysis experiments. Their approach helped predict outcomes based on different types of biomasses, making it easier for researchers and industry professionals to determine the suitability of this technique for their needs. This project, part of the course "Programming in Python for Business and Life Science Analytics," aims to reproduce Yang et al.'s results and conduct further analysis to expand their findings.

2. Methodology

2.1 Data Collection and Preparation

The first step in our study involved collecting and preparing the dataset. We utilized data from various documented microwave pyrolysis experiments, focusing on input parameters such as Biomass Ultimate Analysis, Biomass Proximate Analysis, and Operating Parameters. The output parameters were categorized into three main products: Bio-oil, Syngas, and Biochar.

The dataset included the following input parameters:

A. **Biomass Ultimate Analysis:**
    a. Carbon content
    b. Hydrogen content
    c. Nitrogen content
    d. Oxygen content
    e. Sulfur content
B. **Biomass Proximate Analysis:**
    a. Volatile matter
    b. Fixed carbon content
    c. Ash content
C. **Operating Parameters:**
    a. Operating temperature
    b. Microwave power
    c. Reaction time
    d. Microwave absorber percentage
    e. Dielectric constant of absorber
    f. Dielectric loss factor of the absorber

The output parameters included:

A. Bio-oil yield
B. Syngas composition and yield
C. Biochar yield

2.2 Data Imputation Using Scikit-Learn's Iterative Imputer

Significant missing values in the dataset necessitated data imputation to ensure robust and reliable analysis. Initially, the MissForest approach was considered for imputation. However, we achieved satisfactory results using the Iterative Imputer from the scikit-learn library in Python. This imputation method iteratively models each feature with missing values as a function of other features, effectively filling in the gaps based on observed data patterns. The Iterative Imputer helps maintain the integrity of the data, allowing for more accurate downstream analysis and model training.

2.3 Spearman Correlation Heatmap Analysis

We conducted a Spearman correlation analysis to understand the relationships between input and output parameters and visualized the results using a heatmap. The Spearman correlation coefficient measures the strength and direction of the monotonic relationship between two variables. This non-parametric measure is handy for capturing non-linear relationships that might not be evident through Pearson correlation. The heatmap visualization allowed us to identify strong correlations and potential interactions between the different parameters, providing insights into which variables might significantly impact the outcomes of microwave pyrolysis.

2.4 Principal Component Analysis (PCA)

To reduce the dimensionality of the dataset and identify key patterns, we performed Principal Component Analysis (PCA) using OriginPro. PCA helps transform the dataset into a set of orthogonal (uncorrelated) principal components, which explain the maximum variance in the data.

Steps involved in PCA:

A. **Loading the Data:** The cleaned and imputed dataset was loaded into OriginPro.
B. **Setting Up PCA:** We configured the PCA analysis, selecting the variables corresponding to the input and output parameters.
C. **Running the Analysis:** PCA was performed, and the results were interpreted through eigenvalues, variance, loading plots, and score plots.

2.5 Machine Learning Models

We compared the performance of several machine learning models to predict the outcomes of microwave pyrolysis. The models evaluated included:

A. **Support Vector Regression (SVR):** A regression technique that uses support vector machines to predict continuous outcomes.
B. **Random Forest Regressor (RFR):** An ensemble learning method that combines multiple decision trees to improve predictive accuracy.
C. **Gradient Boosting Regressor (GBR):** An iterative boosting technique that builds a series of weak learners (decision trees) to form a robust predictive model.
D. **Neural Network (NN):** A flexible and robust model capable of capturing complex patterns in the data.

The models were evaluated based on the following metrics:

A. **R-squared (R^2):** Measures the proportion of variance the model explains.
B. **Root Mean Squared Error (RMSE):** Measures the average magnitude of the prediction errors.
C. **Relative Root Mean Squared Error (RRMSE):** Normalizes RMSE by the mean of the observed values

3. Discussion and Results

The data analyzed in the paper is extracted from surveyed literature. Since only some experiments monitored the same parameters, imputing missing data was necessary. Data cleaning and imputation are critical steps in preparing data for analysis, mainly when dealing with real-world datasets that often contain missing or inconsistent values. For the data imputation, our initial plan was to use the MissForest approach, but for unknown reasons, we did not achieve satisfactory results with this method. We decided to use the Iterative Imputer library from scikit-learn instead. Hence for all our further datasets, we utilized the Iterative Imputer from the scikit-learn library in Python.

The initial dataset comprised experimental data on various Bio-oil, Syngas, and Biochar properties. It included measurements such as carbon content, hydrogen content, nitrogen content, oxygen content, sulfur content, volatile matter, fixed carbon, ash content, reaction temperature, microwave power, reaction time, and several others. Upon preliminary inspection, it was observed that the dataset contained a significant amount of missing values, particularly in the columns related to chemical compositions and yields. This posed a challenge, as missing values can distort analysis outcomes. Therefore, imputation was necessary to ensure the completeness and integrity of the dataset.

The imputed data showed significant improvements in completeness and consistency compared to the initial dataset. Missing values, which could have led to biased or incomplete analysis, were effectively estimated and filled. This imputed dataset was now suitable for PCA, allowing us to derive meaningful insights and patterns without the distortions caused by missing values. The Iterative Imputer's capability to use multiple imputations to estimate missing data resulted in a more robust and reliable dataset, enhancing the overall quality of our subsequent analysis.

3.1 Spearman Heatmap Results

In our analysis, we utilized the Spearman correlation to examine the relationships between different parameters. Spearman correlation is a non-parametric measure of rank correlation, which assesses how well the relationship between two variables can be described using a monotonic function. Unlike Pearson correlation, which measures linear relationships, Spearman correlation is based on the ranked values of the data, making it more robust to outliers and able to capture non-linear associations.

To visualize the correlation coefficients, we used masked triangle heat maps generated using the seaborn and matplotlib libraries.

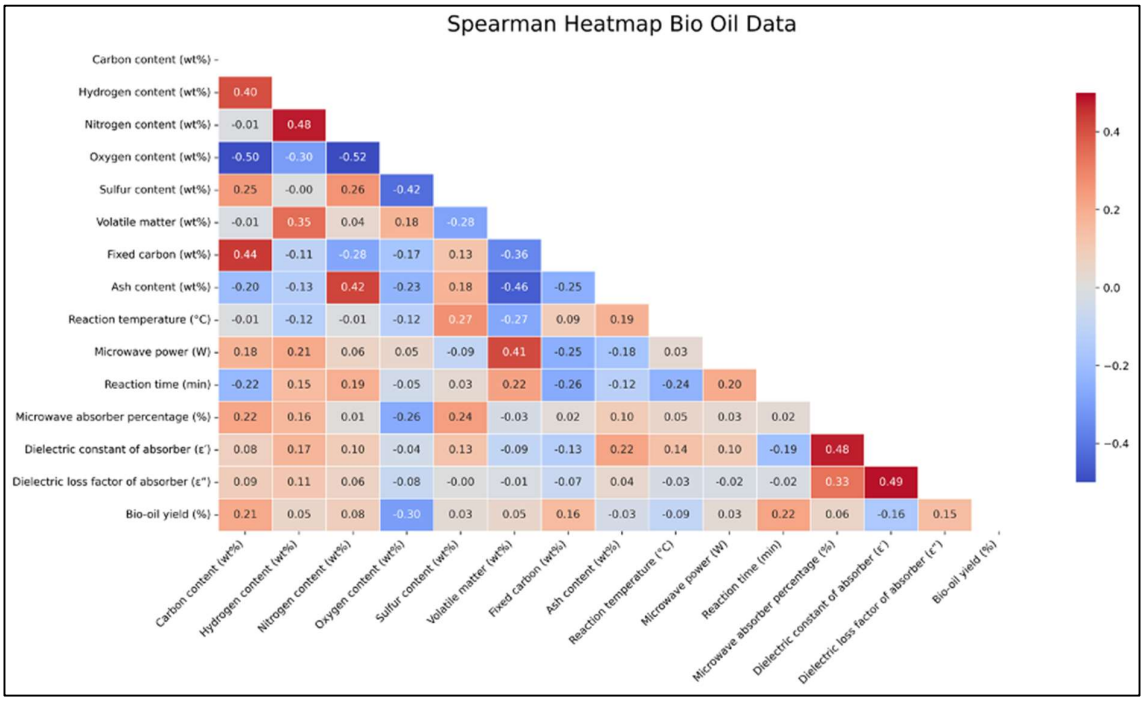


Figure 3. 1 Spearman Heatmap Bio Oil Data

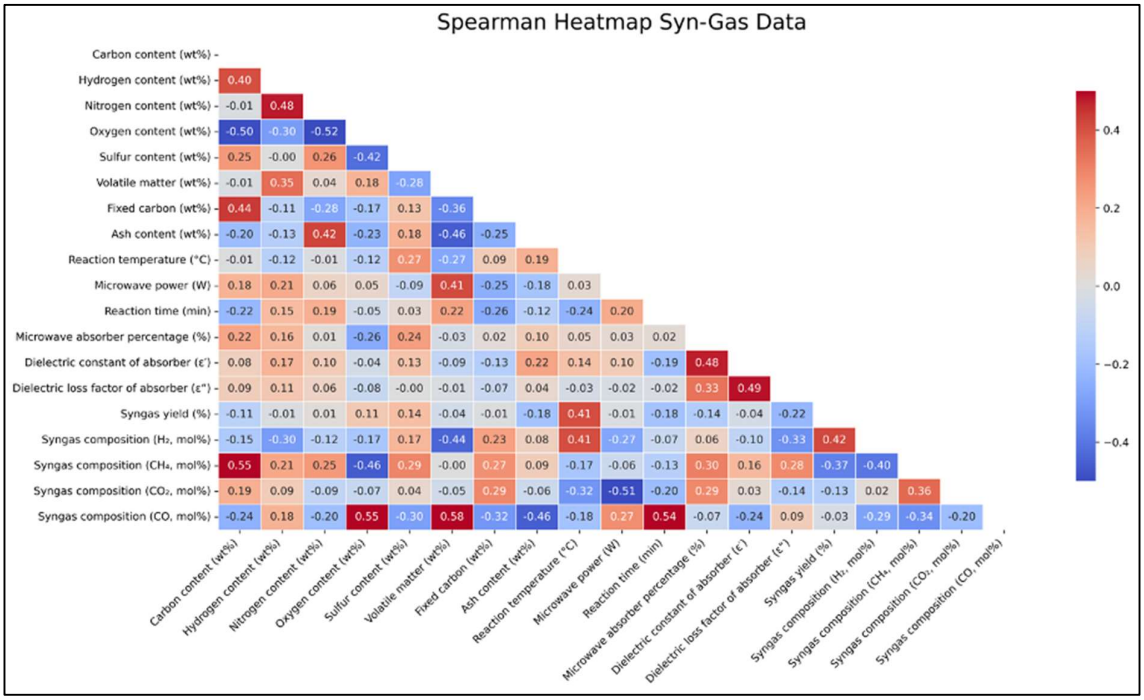


Figure 3. 2 Spearman Heatmap Syngas Data

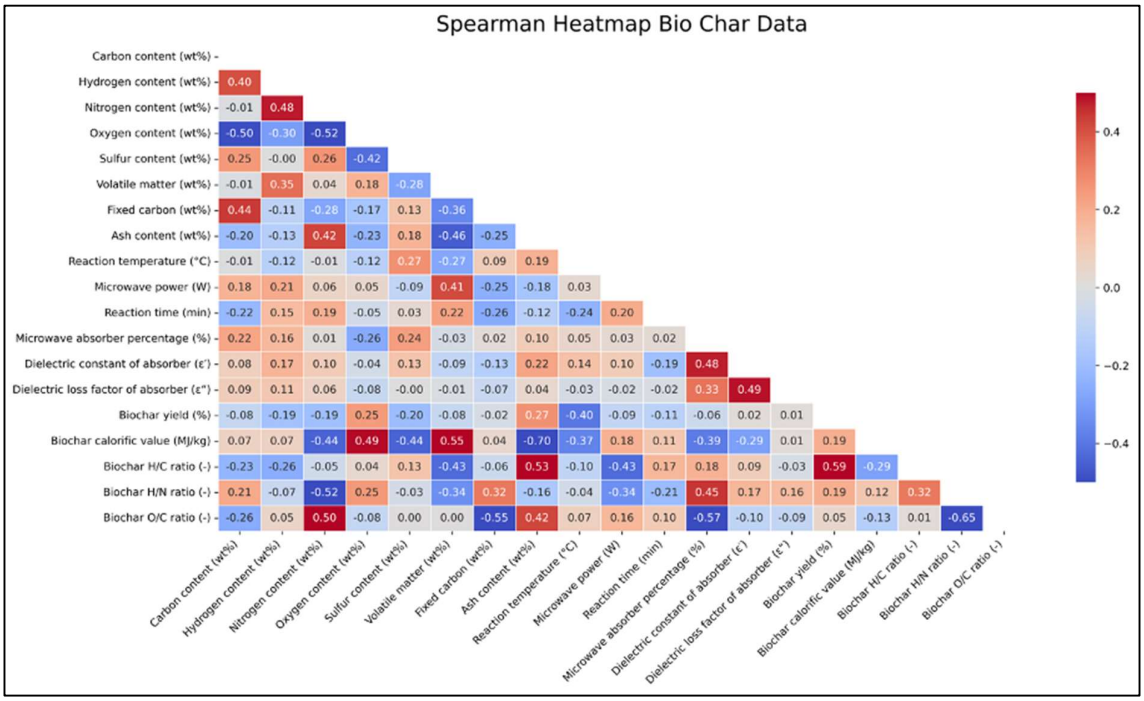


Figure 3. 3 Spearman Heatmap Bio Char Data

For an additional visualization of the operating parameters versus yield, we attempted to implement ternary contour plots. These plots aim to show which values of the three directly adjustable operating parameters (reaction temperature, reaction time, and microwave power) lead to the best bio-oil, syngas, or biochar yield. Unfortunately, all available libraries, such as Seaborn and Plotly, only allow the visualization of normalized data to values between 0 and 1 or 1 and 100. This visualisation method does not fit our goals and results in a rather nonsensical output. However, for completeness, we decided to include the script anyway.

3.2 Principal Component Analysis Results

Principal Component Analysis (PCA) is a statistical procedure that uses an orthogonal transformation to convert a set of possibly correlated variables into values of linearly uncorrelated variables called principal components. This report details the steps for PCA analysis on given datasets using OriginPro.

3.2.1 Data Preparation

The data used for the PCA analysis was sourced from the cleaned imputed data file and stored in 3 separate Excel files.

The datasets included:

A. Bio oil data
B. Syngas data
C. Bio char data

3.2.2 Steps to Perform PCA in OriginPro

A. **Loading the Data:**
    a. Open OriginPro.
    b. Import the cleaned and imputed data from Excel files. This can be done by navigating to File > Import > Excel and selecting the appropriate file.
B. **Setting Up the PCA:**
    a. Go to Analysis > Multivariate Analysis > Principal Component Analysis.
    b. In the dialog box, select the data range containing your variables.
C. **Configuring the PCA:**
    a. Ensure that the correct data columns are selected for analysis.
    b. Choose whether to standardize the data. Standardization is often necessary if the variables have different units or scales.
D. **Running the PCA:**
    a. Click on OK to run the PCA.
    b. OriginPro will generate several outputs, including:
        i. Eigenvalues and explained variance for each principal component.
        ii. Loading plots showing the contribution of each variable to the principal components.
        iii. Score plots showing the samples' distribution in the principal components' space.

3.2.3 Interpretation of Results

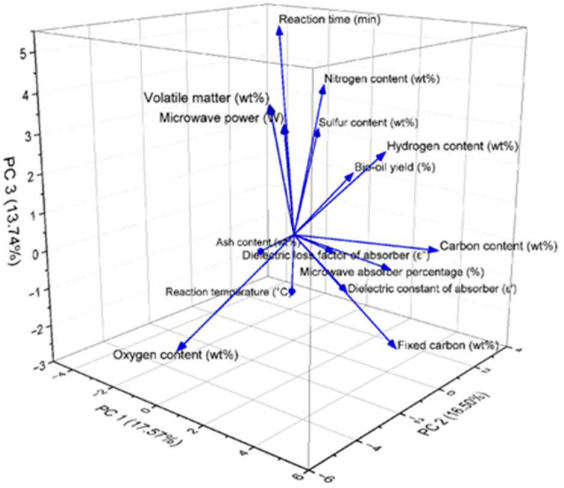
A. **Eigenvalues and Variance:**
    a. The eigenvalues indicate the amount of variance captured by each principal component.
    b. A scree plot can be used to visualize the variance explained by each component, helping to determine the number of elements to retain.
B. **Loading Plots:**
    a. These plots show the weight of each original variable in the principal components.
    b. Variables with high loadings on a principal component are more influential in defining that component.
C. **Score Plots:**
    a. These plots project the original data onto the principal component space.
    b. They help identify patterns, clusters, and outliers in the data.

3.2.4 PCA Results

A. Bio-oil:
    a. The first two principal components explained a significant portion of the variance.
    b. Specific variables, such as those related to chemical composition, had high loadings.
    c. Compared with the research paper by Tabatabaei et al., the loading plots generated by our PCA analysis had some variations. The main reason is the data being used. We cleaned and imputed the data using sklearn iterative

imputer. Also, the authors scaled and standardised the whole data, but the exact amount of unit variance and scaling done is unknown. Hence, we generated variations in the PCA.

d. The PC1, PC2 and PC3 values generated had variations of approximately 12.58%, 1.85% and 11.35% respectively.



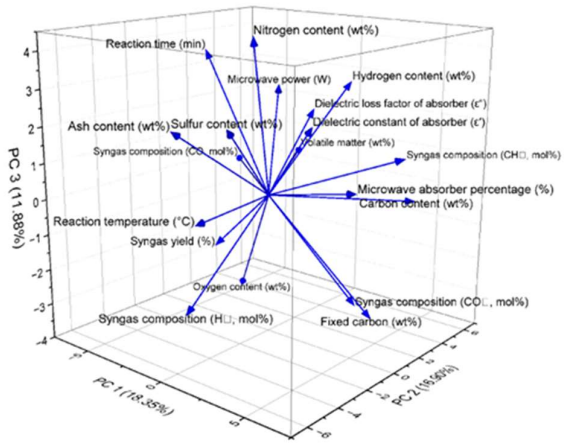
	Coefficients of PC1	Coefficients of PC2	Coefficients of PC3
Carbon content (wt%)	0,51669	0,21066	-0,00896
Hydrogen content (wt%)	0,32047	0,13491	0,22982
Nitrogen content (wt%)	-0,10914	0,28323	0,34807
Oxygen content (wt%)	-0,04942	-0,535	-0,19735
Sulfur content (wt%)	-0,16582	0,31796	0,22277
Volatile matter (wt%)	0,2068	-0,34125	0,38667
Fixed carbon (wt%)	0,39176	0,14226	-0,27513
Ash content (wt%)	-0,44637	0,26208	-0,15862
Reaction temperature (°C)	-0,18872	0,19191	-0,23554
Microwave power (W)	0,12826	-0,191	0,31892
Reaction time (min)	-0,20072	0,12683	0,50659
Microwave absorber percentage (%)	0,25121	0,27835	-0,11114
Dielectric constant of absorber (ε')	0,077	0,22618	-0,1911
Dielectric loss factor of absorber (ε")	0,1282	0,09156	-0,0479
Bio-oil yield (%)	0,12314	0,19516	0,14838

Figure 3. 4 Loading plot of Bio oil and coefficients of PC1, PC2 and PC3

B. Syngas:

a. Similar patterns were observed, with significant variables influencing the principal components related to gas composition.

b. The PC1, PC2 and PC3 values generated had variations of approximately 12.62%, 13.77% and 30.12% respectively.



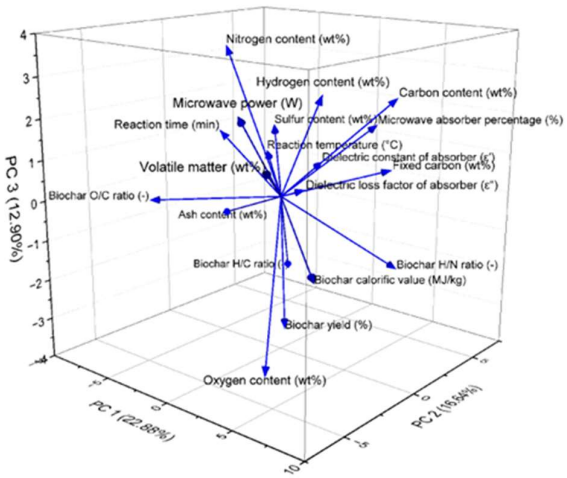
	Coefficients of PC1	Coefficients of PC2	Coefficients of PC3
Carbon content (wt%)	0,40102	0,20744	0,01179
Hydrogen content (wt%)	0,1518	0,22442	0,28821
Nitrogen content (wt%)	0,04189	-0,1295	0,42156
Oxygen content (wt%)	-0,28823	0,21602	-0,35942
Sulfur content (wt%)	0,0363	-0,24518	0,20996
Volatile matter (wt%)	-0,16217	0,39208	0,04982
Fixed carbon (wt%)	0,35672	0,04592	-0,27271
Ash content (wt%)	-0,05727	-0,38271	0,21099
Reaction temperature (°C)	-0,0272	-0,3188	-0,02757
Microwave power (W)	-0,15335	0,25303	0,24724
Reaction time (min)	-0,20254	-0,0604	0,36501
Microwave absorber percentage (%)	0,31941	0,00957	0,05174
Dielectric constant of absorber (ε')	0,17501	-0,01507	0,20278
Dielectric loss factor of absorber (ε")	0,06147	0,15333	0,21582
Syngas yield (%)	-0,09508	-0,14954	-0,1246
Syngas composition (H_2, mol%)	0,0031	-0,39736	-0,23459
Syngas composition (CH_4, mol%)	0,40044	0,14164	0,13009
Syngas composition (CO_2, mol%)	0,27184	0,08804	-0,26702
Syngas composition (CO, mol%)	-0,358	0,28205	0,01319

Figure 3. 5 Loading plot of Syngas and coefficients of PC1, PC2 and PC3

C. Biochar:

a. The analysis revealed distinct clustering, indicating different properties or conditions of the biochar samples.

b. The PC1, PC2 and PC3 values generated had variations of approximately 44.06%, 22.24% and 7.86% respectively.



	Coefficients of PC1	Coefficients of PC2	Coefficients of PC3
Carbon content (wt%)	0,28567	0,2685	0,26222
Hydrogen content (wt%)	0,12618	0,06328	0,28067
Nitrogen content (wt%)	-0,21171	-0,04515	0,39626
Oxygen content (wt%)	0,15915	-0,24633	-0,41814
Sulfur content (wt%)	-0,19022	0,16813	0,15548
Volatile matter (wt%)	0,22954	-0,31546	0,14694
Fixed carbon (wt%)	0,26172	0,2769	0,06044
Ash content (wt%)	-0,41284	0,12336	-0,12964
Reaction temperature (°C)	-0,16396	0,10242	0,0943
Microwave power (W)	0,09117	-0,29275	0,26505
Reaction time (min)	-0,1847	-0,10588	0,17747
Microwave absorber percentage (%)	0,11313	0,38131	0,15668
Dielectric constant of absorber (ε')	-0,02619	0,24083	0,05427
Dielectric loss factor of absorber (ε'')	0,05521	0,06318	0,01463
Biochar yield (%)	0,00274	0,01845	-0,387
Biochar calorific value (MJ/kg)	0,38544	-0,26681	-0,11599
Biochar H/C ratio (-)	-0,22646	0,28551	-0,307
Biochar H/N ratio (-)	0,24583	0,34197	-0,24092
Biochar O/C ratio (-)	-0,39798	-0,2382	-0,02274

Figure 3. 6 Loading plot of Biochar and coefficients of PC1, PC2 and PC3

Table 3. 1 Comparison of generated PC values with original paper

OUTPUT	PC1		PC2		PC3	
	Testing	Paper	Testing	Paper	Testing	Paper
Bio-oil	17.57	20.1	16.5	16.2	13.74	15.5
Syngas	18.35	21.0	16.9	19.6	11.88	17.0
Bio Char	22.88	40.9	16.64	21.4	12.9	14.0

The PCA results were visualized using:

A. Heat maps are used to show the correlation between variables.
B. Loading plots to understand the influence of each variable.
C. Score plots to visualize sample distribution.

PCA provided a comprehensive method to reduce dimensionality and uncover patterns in the datasets. The analysis highlighted the key variables influencing the principal components and provided insights into the underlying structure of the data. OriginPro proved to be an effective tool for performing and visualizing PCA, facilitating a deeper understanding of the experimental results.

3.3 Comparison of Machine Learning Models: SVR, GBR, RFR and NN

In the original paper, machine learning is utilized as the primary data analysis approach. Specifically, three models are included in the analysis: Support Vector Regression (SVR), Random Forest Regressor (RFR), and Gradient Boosting Regressor (GBR).

Additionally, we tried to utilize a Neural Network (NN) as an alternative to the traditional Machine Learning Models used in the paper.

Below, we introduce the theory behind these four models and proceed with a detailed comparison of their performance and applications.

3.3.1 Theoretical Background on ML models

A. Support Vector Regression (SVR)
    a. **Concept**: SVR is a Support Vector Machine (SVM) used for regression tasks. It aims to find a function that deviates from the actual observed values by a value no more significant than a specified margin (epsilon) while being as flat as possible.
    b. **Key Features**:
        i. Effective in high-dimensional spaces.
        ii. Robust to overfitting, especially in high-dimensional space.
        iii. Utilizes kernel functions to handle non-linear relationships.

B. Random Forest Regressor (RFR)
    a. **Concept**: RFR is an ensemble learning method that constructs multiple decision trees during training and outputs the mean prediction of the individual trees to improve predictive accuracy and control overfitting.
    b. **Key Features**:
        i. Reduces overfitting by averaging multiple trees.
        ii. Handles large datasets efficiently.
        iii. Provides feature importance estimates.

C. Gradient Boosting Regressor (GBR)
    a. **Concept**: GBR builds an ensemble of trees sequentially, where each tree corrects the errors of the previous one, using gradient descent to minimize the loss function.
    b. **Key Features**:
        i. Highly accurate and robust.
        ii. Can handle complex data structures.
        iii. Effective in capturing non-linear patterns.
        iv. Less interpretable than single decision trees.

D. Neural Networks (NN)
    a. **Concept**: Neural networks are computational models inspired by the human brain, consisting of interconnected layers of neurons that process input data to generate outputs. They are capable of learning complex patterns through training on large datasets.
    b. **Key Features**:
        i. Capable of modelling complex and non-linear relationships.
        ii. Highly flexible and can be adapted to various types of data.
        iii. Significant computational resources and large amounts of data are required for practical training.
        iv. Potentially prone to overfitting, requiring techniques like dropout and regularization to mitigate.

3.3.2 Analysis Process

In this report, we aim to recreate the results from the original paper. To achieve this, we first replicate the dataset following the same data preprocessing procedures described in the article. Using this "clean" dataset, we then conduct further machine learning testing.

The dataset comprises various columns representing different independent and dependent variables that we incorporate into the model. In the model, these variables are categorized as "target column" and "feature columns." The target column represents the dependent variable, while the feature columns represent the independent variables. Our objective is to understand how the feature variables affect the target column.

The detailed code and analysis process can be found in the accompanying code. The final result of the three traditional models is presented in a graph, which includes a table of the performance of the models, showing metrics such as R-squared (R^2), Root Mean Squared Error (RMSE), and Relative Root Mean Squared Error (RRMSE) for both the training and testing sets. The NNs performance was solely evaluated using the three metrics mentioned above.

3.3.3 NN Implementation Details:

The neural network architecture consisted of multiple layers, including linear layers, batch normalization, ReLU activation functions, and dropout layers to prevent overfitting.

A. **Data Preparation:** We loaded the dataset, scaled the features using StandardScaler, and split the data into training and validation sets.
B. **Model Architecture:** The ImprovedPyrolysisNN class defined the neural network, featuring an input layer, four hidden layers with batch normalization, and an output layer. Dropout was added to each hidden layer to reduce overfitting.
C. **Training:** The model was trained using the Adam optimizer and mean squared error loss function over 200 epochs. We used PyTorch's DataLoader for efficient mini-batch processing.
D. **Hyperparameter Optimization:** We employed Optuna to optimize the neural network's hyperparameters, including the number of layers, hidden layer size, dropout rate, and learning rate. This resulted in an optimal architecture for the task.
E. **Evaluation:** The final model was evaluated on the validation set, achieving a good balance between prediction accuracy and robustness. Key metrics such as R-Squared (R^2), Mean Absolute Error (MAE), and Root Mean Squared Error (RMSE) were computed to assess the model's performance.

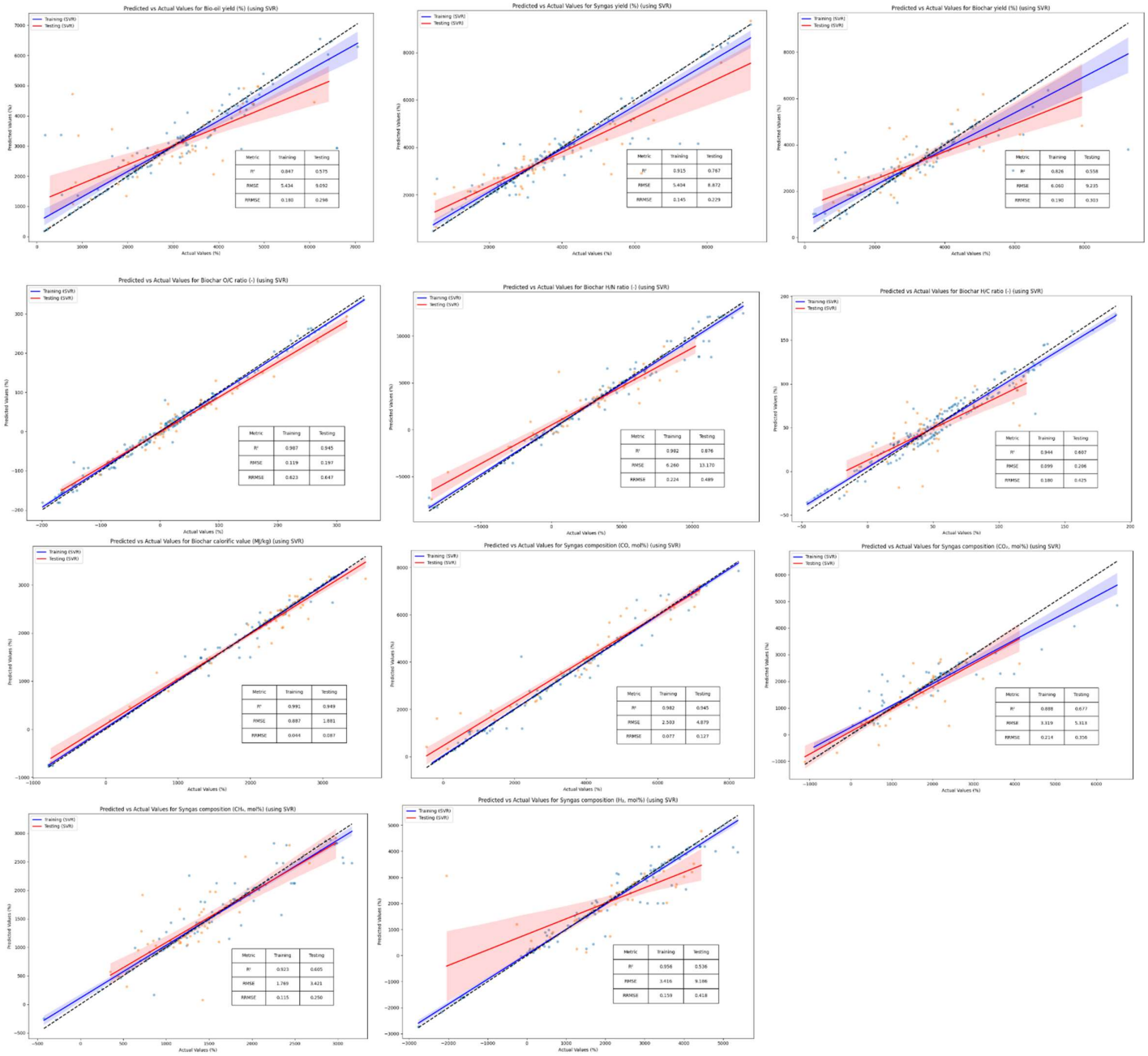


Figure 3. 7 Predicted Values Against Experimental Data (A) SVR Model

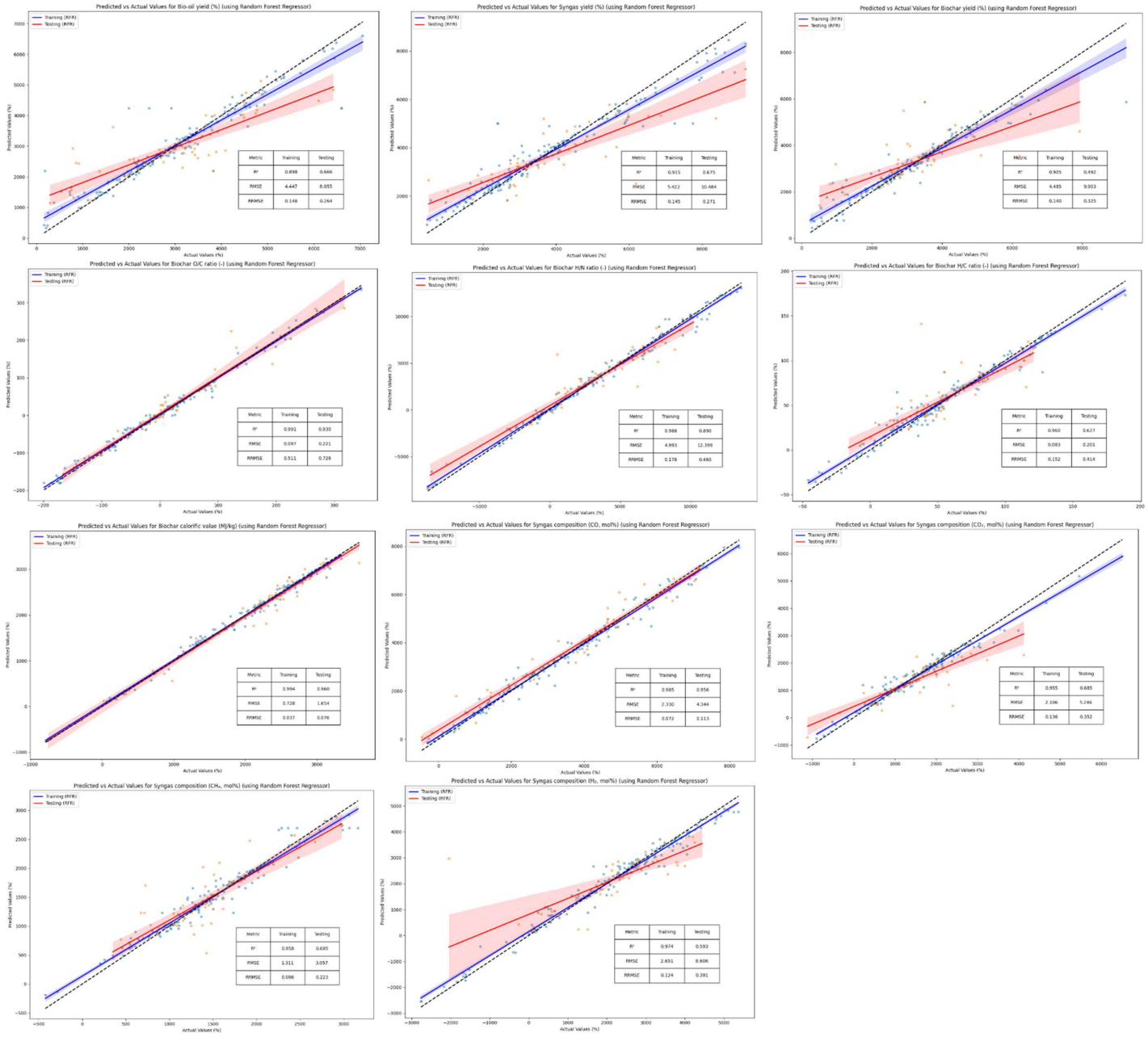


Figure 3. 8 Predicted Values Against Experimental Data (B) RFR Model

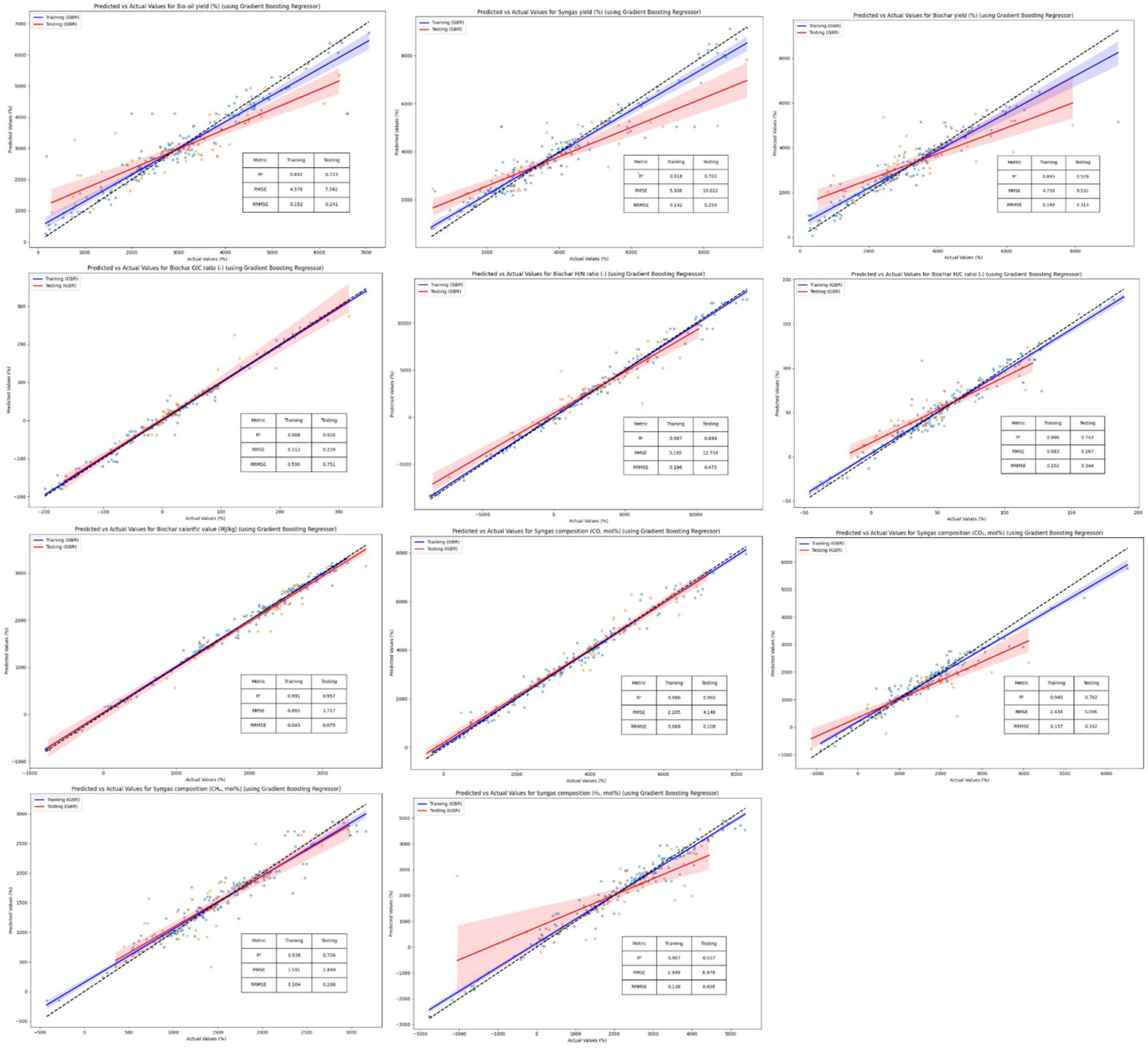


Figure 3. 9 Predicted Values Against Experimental Data (C) GBR Model

3.3.4 ML Models Results

In this research, unfortunately, we could not completely recreate the identical analysis results from the paper. This could be due to several issues:

A. **Lack of Original Code**: Without access to the original code, some parameters might need adjustment in the models.
B. **Visualization Reproduction**: Without seeing the original code, it is challenging to recreate identical visualizations.

As for the performance across the models:

A. **Gradient Boosting Regressor (GBR)** performed the best, showing superior accuracy and robustness.
B. **Random Forest Regressor (RFR)** also performed well but was slightly less accurate than GBR.
C. **Support Vector Regression (SVR)** performed the worst among the traditional models, indicating its limitations in this specific dataset and context.
D. Neural Network (NN) performed the worst among all models most likely due to the limited amounts of data

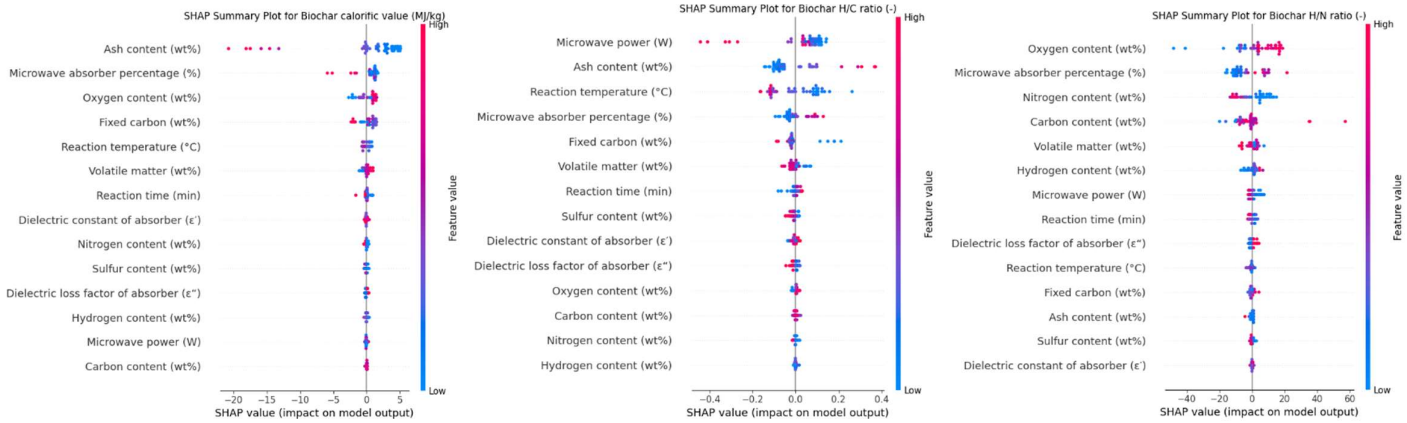
The predictions generated by the neural network did not outperform those produced by traditional machine learning approaches. As a result, the Gradient Boost Regressor continues to demonstrate the best performance.
The reason here might be that Neural networks typically require large amounts of data to perform well and if the dataset is small or lacks diversity, results often underperform.

3.4 SHAP Analysis Results

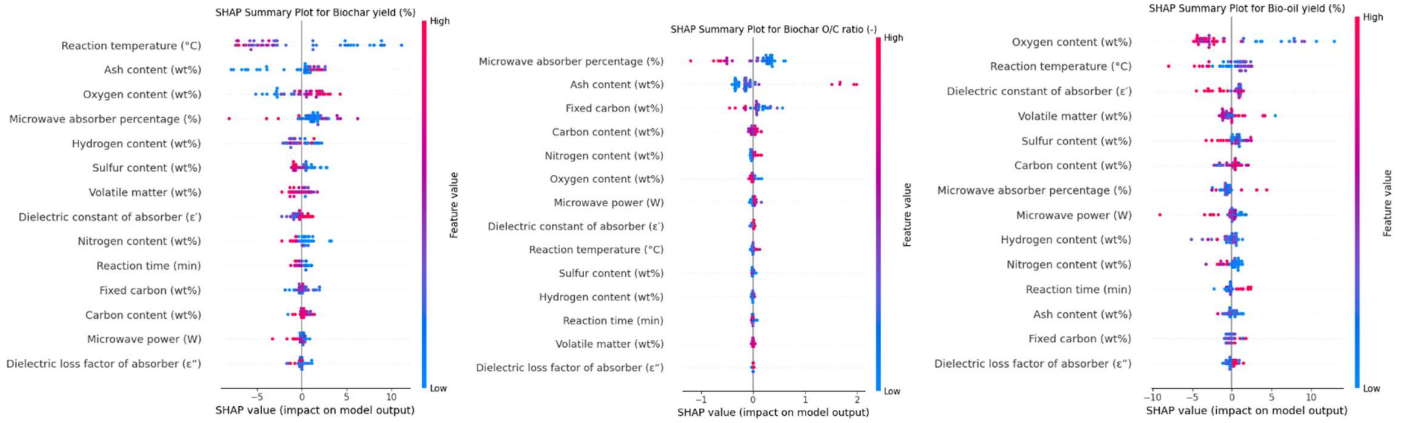
A significant part of this analysis involves SHAP (SHapley Additive exPlanations) analysis, which provides more insight into model interpretability and feature importance. SHAP values help explain the output of machine learning models by quantifying the contribution of each feature to the prediction. This section delves into the SHAP analysis to understand better how each feature affects the target variable across different models.

3.4.1 Introduction to SHAP

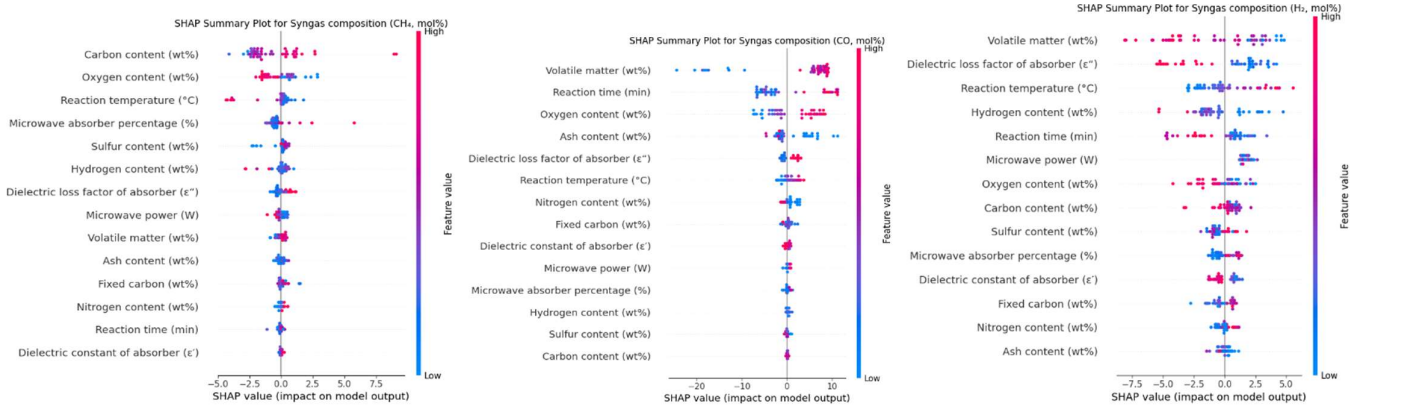
SHAP is a unified approach that explains the output of any machine learning model. It connects optimal credit allocation with local explanations using the classic Shapley values from cooperative game theory. SHAP values represent the impact of each feature on the model's output, enabling a deeper understanding of the model's predictions.



(A) bio-char calorific value (B) biochar H/C ratio (C) biochar H/N ratio

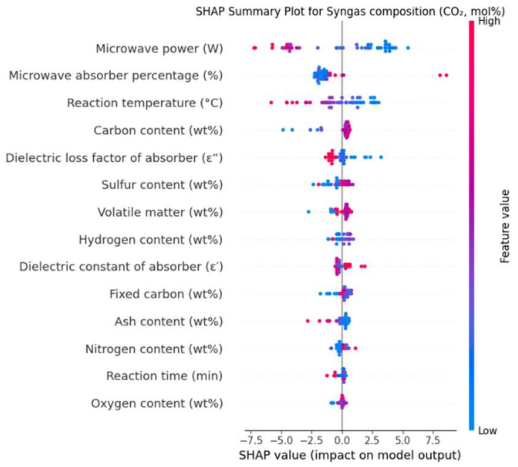


(D) biochar yield (E) biochar O/C ratio (F) Bio oil yield

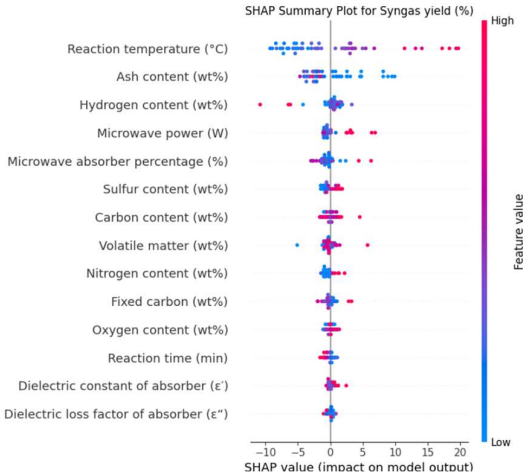


(G) CH4 concentration (H) CO concentration (I) H2 concentration

Figure 3. 10 The SHAP value representing the impacts of input features on each output in biomass microwave pyrolysis: (A) bio-char calorific value, (B) biochar H/C ratio, (C) biochar H/N ratio, (D) biochar yield, (E) biochar O/C ratio, (F) Bio oil yield, (G) CH4 concentration, (H) CO concentration, (I) H2 concentration, (J) CO2 concentration, (K) syngas yield



(J) CO2 concentration



(K) syngas yield

Figure 3. 11 continued.

3.4.2 SHAP Analysis Interpretation

In this section, we will explore an example of SHAP analysis to provide more insight into its interpretability and feature importance. Below is the SHAP summary plot for Bio-oil yield (%), demonstrating how different feature variables influence the target variable.

Example SHAP Summary Plot Interpretation

In this SHAP summary plot for Bio-oil yield (%), we can observe the following:

A. **Feature Variables**:
   a. The y-axis lists all the feature variables tested in the model, such as Oxygen content (wt%), Reaction temperature (°C), Dielectric constant of absorber (ε′), Volatile matter (wt%), Sulfur content (wt%), Carbon content (wt%), Microwave absorber percentage (%), Microwave power (W), Hydrogen content (wt%), Nitrogen content (wt%), Reaction time (min), Ash content (wt%), Fixed carbon (wt%), and Dielectric loss factor of absorber (ε″).
B. **Feature Contribution**:
   a. The x-axis represents the SHAP values, indicating the impact of each feature on the model's output. A positive SHAP value means that the feature contributes to increasing the Bio-oil yield (%), while a negative SHAP value indicates a contribution to decreasing the yield.
C. **Color Coding**:
   a. The colour represents the feature value, ranging from low (blue) to high (red). This coding helps to visualize the relationship between the feature value and its impact on the target variable.

D. **Interpretation of Specific Features**:
    a. **Oxygen Content (wt%)**: High oxygen content values (red dots) generally positively impact the Bio-oil yield, as indicated by the clustering of red dots on the positive side of the SHAP values.
    b. **Reaction Temperature (°C)**: Higher reaction temperatures (red dots) show a mixed impact, but generally, lower temperatures (blue dots) tend to decrease the Bio-oil yield.
    c. **Dielectric Constant of Absorber (ε′)**: This feature shows a complex relationship with the Bio-oil yield, with high and low values having varying impacts.
    d. **Volatile Matter (wt%)**: High volatile matter content (red dots) appears to positively impact the Bio-oil yield, similar to oxygen content.
    e. **Sulfur Content (wt%)**: This feature shows less variation, indicating it might not strongly influence the Bio-oil yield.
E. **General Observations**:
    a. Features at the top of the list, such as Oxygen content (wt%) and Reaction temperature (°C), have more substantial impacts on the model's predictions compared to those lower down the list, like Fixed carbon (wt%) and Dielectric loss factor of absorber (ε″).
    b. This ordering helps identify the most critical features that need attention during optimization or further research.

Insights from SHAP Analysis

The SHAP analysis provides valuable insights into how each feature affects the target variable, Bio-oil yield (%), across different models. This interpretability is crucial for understanding the model's behavior and making informed decisions based on the model's predictions.

A. **Global Interpretation**:
    a. The summary plots show the distribution of SHAP values for each feature, indicating each feature's overall importance and impact on the model's predictions.
B. **Local Interpretation**:
    a. SHAP values also provide local interpretability by explaining individual predictions. This helps us understand why the model made a particular prediction for a specific instance.

The SHAP summary plot for Bio-oil yield (%) is just one example of how SHAP analysis can be utilized to interpret machine learning models. By leveraging SHAP values, we comprehensively understand feature contributions and model behaviour. This analysis enhances our ability to analyse and trust the machine learning models, ensuring that the predictions are accurate and explainable.

4. Conclusion

Microwave pyrolysis offers a viable solution for converting plastic waste into valuable byproducts such as bio-oil, syngas, and biochar. Despite challenges like high energy consumption and economic inefficiency, the technique holds promise for sustainable waste management and resource recovery. Machine learning techniques can significantly enhance the predictability and efficiency of microwave pyrolysis by analyzing and optimizing process parameters.

Our study successfully reproduced and extended the findings of Yang et al., demonstrating the effectiveness of machine learning models in predicting microwave pyrolysis outcomes. The Gradient Boosting Regressor (GBR) performed superior, followed by the Random Forest Regressor (RFR). These models can provide valuable insights for researchers and industry professionals, aiding in optimizing microwave pyrolysis processes.

Future work should focus on optimizing the process and exploring larger, more diverse datasets to improve model accuracy and reliability. Additionally, further research into reducing the energy consumption of microwave pyrolysis and enhancing its economic viability will be crucial for its widespread adoption. Combining advanced machine learning techniques with experimental research can pave the way for more efficient and sustainable plastic waste management solutions.

ALL OUR CODE DONE IN THE PROJECT WORK IS ATTACHED BELOW :-

DATA IMPUTATION

```
pip install scikit-learn
pip install openpyxl
import pandas as pd
import numpy as np
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer

# Load the dataset
file_path = 'C:/Users/mhatr/OneDrive/Desktop/python lab course/Jitesh Project/MGT001437_ProjectC/data.xlsx'  # Replace this with the correct file path
data = pd.read_excel(file_path)

# Set appropriate column names using the second row and remove the first two rows
data.columns = data.iloc[1]
data = data.drop([0, 1])

# Reset the index and remove columns that are completely empty or unnamed
data = data.reset_index(drop=True)
data.columns = data.columns.str.strip()
data = data.loc[:, ~data.columns.str.contains('^Unnamed')]

# Identify columns that should be numeric
numeric_columns = [
    'Carbon content (wt%)', 'Hydrogen content (wt%)', 'Nitrogen content (wt%)',
    'Oxygen content (wt%)', 'Sulfur content (wt%)', 'Volatile matter (wt%)',
    'Fixed carbon (wt%)', 'Ash content (wt%)', 'Reaction temperature (°C)',
    'Microwave power (W)', 'Reaction time (min)', 'Microwave absorber percentage (%)',
    'Dielectric constant of absorber (ε’)', 'Dielectric loss factor of absorber (ε“)',
    'Bio-oil yield (%)', 'Syngas yield (%)', 'Syngas composition (H₂, mol%)',
    'Syngas composition (CH₄, mol%)', 'Syngas composition (CO₂, mol%)',
    'Syngas composition (CO, mol%)', 'Biochar yield (%)', 'Biochar calorific value (MJ/kg)',
    'Biochar H/C ratio (-)', 'Biochar H/N ratio (-)', 'Biochar O/C ratio (-)'
]

# Convert identified columns to numeric, coercing errors
for col in numeric_columns:
    data[col] = pd.to_numeric(data[col], errors='coerce')

# Separate numeric and non-numeric data
numeric_data = data[numeric_columns]
non_numeric_data = data.drop(columns=numeric_columns)

# Debugging: Print the structure of numeric_data
print("Numeric data columns:", numeric_data.columns)
print("Numeric data head:\n", numeric_data.head())
print("Non-numeric data columns:", non_numeric_data.columns)
print("Non-numeric data head:\n", non_numeric_data.head())

# Check if numeric_data is empty
if numeric_data.empty:
    raise ValueError("No numeric columns found in the dataset to impute.")

# Initialize the IterativeImputer
imputer = IterativeImputer(random_state=0)

# Impute missing values in the numeric data
imputed_data = imputer.fit_transform(numeric_data)

# Convert the imputed numpy array back to a DataFrame with appropriate column names
imputed_df = pd.DataFrame(imputed_data, columns=numeric_data.columns)

# Concatenate the imputed numeric data with the non-numeric columns
final_data = pd.concat([non_numeric_data.reset_index(drop=True), imputed_df.reset_index(drop=True)], axis=1)

# Display the first few rows of the final cleaned and imputed dataset
print(final_data.head())

# Save the cleaned and imputed dataset to a new Excel file
final_data.to_excel('cleaned_imputed_data.xlsx', index=False)

```

3D PCA WITH LOADINGS

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from mpl_toolkits.mplot3d import Axes3D

# Load your dataset
df = pd.read_excel('C:/Users/mhatr/OneDrive/Desktop/python lab course/Jitesh Project/cleaned_imputed_data.xls
x')
# Remove all columns that are strings
data = df.select_dtypes(include=[np.number])

# Standardize the features
scaler = StandardScaler()
scaled_data = scaler.fit_transform(data)

# Perform PCA
pca = PCA(n_components=3)
principal_components = pca.fit_transform(scaled_data)
loadings = pca.components_.T * np.sqrt(pca.explained_variance_)

# Create a DataFrame with the principal components
pca_df = pd.DataFrame(data=principal_components, columns=['PC1', 'PC2', 'PC3'])

# Plot the 3D PCA results
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(pca_df['PC1'], pca_df['PC2'], pca_df['PC3'], c='r', marker='o', label='Scores')

# Plot the loadings (arrows)
for i, (x, y, z) in enumerate(loadings):
    ax.quiver(0, 0, 0, x, y, z, color='b', arrow_length_ratio=0.1)
    ax.text(x, y, z, data.columns[i], color='b')

ax.set_title('3D PCA Plot with Loadings')
ax.set_xlabel(f'PC1 ({pca.explained_variance_ratio_[0] * 100:.1f}%)')
ax.set_ylabel(f'PC2 ({pca.explained_variance_ratio_[1] * 100:.1f}%)')
ax.set_zlabel(f'PC3 ({pca.explained_variance_ratio_[2] * 100:.1f}%)')

# Plot 95% confidence ellipse (optional, needs further calculation)
# This is a simplified version and may not match exactly
# from matplotlib.patches import Ellipse
# angle = np.arctan2(pca_df['PC2'], pca_df['PC1'])
# width = 2 * np.sqrt(pca.explained_variance_ratio_[0])
# height = 2 * np.sqrt(pca.explained_variance_ratio_[1])
# ell = Ellipse(xy=(np.mean(pca_df['PC1']), np.mean(pca_df['PC2'])),
#               width=width, height=height,
#               edgecolor='r', fc='None', lw=2, label='95% Confidence Ellipse')
# ax.add_patch(ell)

ax.legend()
plt.show()

```

SPEARMAN HEATMAP CORELATION

```python
import pandas as pd
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt

path = 'cleaned_imputed_data.xlsx'

# Create df
df = pd.read_excel(path)

# Ensure column names are correct
df.columns = df.columns.str.strip()

# gather the column names as str for next step
column_names = df.columns.tolist()

Bio_Oil_Columns = ['Carbon content (wt%)',
 'Hydrogen content (wt%)',
 'Nitrogen content (wt%)',
 'Oxygen content (wt%)',
 'Sulfur content (wt%)',
 'Volatile matter (wt%)',
 'Fixed carbon (wt%)',
 'Ash content (wt%)',
 'Reaction temperature (°C)',
 'Microwave power (W)',
 'Reaction time (min)',
 'Microwave absorber percentage (%)',
 'Dielectric constant of absorber (ε′)',
 'Dielectric loss factor of absorber (ε″)',
 'Bio-oil yield (%)',]

Syn_Gas_Columns = ['Carbon content (wt%)',
 'Hydrogen content (wt%)',
 'Nitrogen content (wt%)',
 'Oxygen content (wt%)',
 'Sulfur content (wt%)',
 'Volatile matter (wt%)',
 'Fixed carbon (wt%)',
 'Ash content (wt%)',
 'Reaction temperature (°C)',
 'Microwave power (W)',
 'Reaction time (min)',
 'Microwave absorber percentage (%)',
 'Dielectric constant of absorber (ε′)',
 'Dielectric loss factor of absorber (ε″)',
 'Syngas yield (%)',
 'Syngas composition ($H_2$, mol%)',
 'Syngas composition ($CH_4$, mol%)',
 'Syngas composition ($CO_2$, mol%)',
 'Syngas composition (CO, mol%)',]

Bio_Char_Columns = ['Carbon content (wt%)',
 'Hydrogen content (wt%)',
 'Nitrogen content (wt%)',
 'Oxygen content (wt%)',
 'Sulfur content (wt%)',
 'Volatile matter (wt%)',
 'Fixed carbon (wt%)',
 'Ash content (wt%)',
 'Reaction temperature (°C)',
 'Microwave power (W)',
 'Reaction time (min)',
 'Microwave absorber percentage (%)',
 'Dielectric constant of absorber (ε′)',
 'Dielectric loss factor of absorber (ε″)',
 'Biochar yield (%)',
 'Biochar calorific value (MJ/kg)',
 'Biochar H/C ratio (-)',
 'Biochar H/N ratio (-)',
 'Biochar O/C ratio (-)']


Bio_Oil_data = df[Bio_Oil_Columns]
Syn_Gas_Data = df[Syn_Gas_Columns]
Bio_Char_Data = df[Bio_Char_Columns]

def create_spearman_heatmap(df, title_name, file_name):

    #put data in correlation matrix eg spearman or pearson
    correlation_matrix = df.corr(method='spearman')
    #generate maskt to make a triangle shaped matrix
    mask = np.triu(np.ones_like(correlation_matrix, dtype=bool))
    #figure size
    plt.figure(figsize=(16, 9))
    #genaerate heatmap from correlation matrix and set colour, initialize the temperature bat etc.
    heatmap = sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f", vmin=-0.5, vmax=0.5, mask=mask, linewidths=0.5, linecolor="White", cbar_kws={"shrink": .
8}) #title
    heatmap.set_title(f'Spearman Heatmap {title_name}', fontdict={'fontsize':20}, pad=12)
    #turn xaxis so make it better readable
    plt.xticks(rotation=45, ha='right')
    #makes better layout
    plt.tight_layout()
    #saves file as file name
    plt.savefig(file_name, dpi=500, bbox_inches='tight')
    plt.show()

create_spearman_heatmap(Bio_Oil_data, 'Bio Oil Data', 'Bio_Oil_Spearman_Heatmap.png')
create_spearman_heatmap(Syn_Gas_Data, 'Syn-Gas Data', 'Syn_Gas_Spearman_Heatmap.png')
create_spearman_heatmap(Bio_Char_Data, 'Bio Char Data', 'Bio_Char_Spearman_Heatmap.png')


```

TERNARY CONTOUR PLOT

```python
import pandas as pd
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt

path = 'cleaned_imputed_data.xlsx'

# Create df
df = pd.read_excel(path)

# Ensure column names are correct
df.columns = df.columns.str.strip()

# gather the column names as str for next step
column_names = df.columns.tolist()

Bio_Oil_Columns = ['Carbon content (wt%)',
 'Hydrogen content (wt%)',
 'Nitrogen content (wt%)',
 'Oxygen content (wt%)',
 'Sulfur content (wt%)',
 'Volatile matter (wt%)',
 'Fixed carbon (wt%)',
 'Ash content (wt%)',
 'Reaction temperature (°C)',
 'Microwave power (W)',
 'Reaction time (min)',
 'Microwave absorber percentage (%)',
 'Dielectric constant of absorber (ε′)',
 'Dielectric loss factor of absorber (ε″)',
 'Bio-oil yield (%)',]

Syn_Gas_Columns = ['Carbon content (wt%)',
 'Hydrogen content (wt%)',
 'Nitrogen content (wt%)',
 'Oxygen content (wt%)',
 'Sulfur content (wt%)',
 'Volatile matter (wt%)',
 'Fixed carbon (wt%)',
 'Ash content (wt%)',
 'Reaction temperature (°C)',
 'Microwave power (W)',
 'Reaction time (min)',
 'Microwave absorber percentage (%)',
 'Dielectric constant of absorber (ε′)',
 'Dielectric loss factor of absorber (ε″)',
 'Syngas yield (%)',
 'Syngas composition ($H_2$, mol%)',
 'Syngas composition ($CH_4$, mol%)',
 'Syngas composition ($CO_2$, mol%)',
 'Syngas composition ($CO$, mol%)',]

Bio_Char_Columns = ['Carbon content (wt%)',
 'Hydrogen content (wt%)',
 'Nitrogen content (wt%)',
 'Oxygen content (wt%)',
 'Sulfur content (wt%)',
 'Volatile matter (wt%)',
 'Fixed carbon (wt%)',
 'Ash content (wt%)',
 'Reaction temperature (°C)',
 'Microwave power (W)',
 'Reaction time (min)',
 'Microwave absorber percentage (%)',
 'Dielectric constant of absorber (ε′)',
 'Dielectric loss factor of absorber (ε″)',
 'Biochar yield (%)',
 'Biochar calorific value (MJ/kg)',
 'Biochar H/C ratio (-)',
 'Biochar H/N ratio (-)',
 'Biochar O/C ratio (-)']


Bio_Oil_data = df[Bio_Oil_Columns]
Syn_Gas_Data = df[Syn_Gas_Columns]
Bio_Char_Data = df[Bio_Char_Columns]

def create_spearman_heatmap(df, title_name, file_name):

    #put data in correlation matrix eg spearman or pearson
    correlation_matrix = df.corr(method='spearman')
    #generate maskt to make a triangle shaped matrix
    mask = np.triu(np.ones_like(correlation_matrix, dtype=bool))
    #figure size
    plt.figure(figsize=(16, 9))
    #genaerate heatmap from correlation matrix and set colour, initialize the temperature bat etc.
    heatmap = sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f", vmin=-0.5, vmax=0.5, mask=mask, linewidths=0.5, linecolor="White", cbar_kws={"shrink": .
8}) #title
    heatmap.set_title(f'Spearman Heatmap {title_name}', fontdict={'fontsize':20}, pad=12)
    #turn xaxis so make it better readable
    plt.xticks(rotation=45, ha='right')
    #makes better layout
    plt.tight_layout()
    #saves file as file name
    plt.savefig(file_name, dpi=500, bbox_inches='tight')
    plt.show()

create_spearman_heatmap(Bio_Oil_data, 'Bio Oil Data', 'Bio_Oil_Spearman_Heatmap.png')
create_spearman_heatmap(Syn_Gas_Data, 'Syn-Gas Data', 'Syn_Gas_Spearman_Heatmap.png')
create_spearman_heatmap(Bio_Char_Data, 'Bio Char Data', 'Bio_Char_Spearman_Heatmap.png')
```

HEATMAP PCA

```python
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

def create_PCA_heatmap(path, title_name, file_name):
    # Read the data from the Excel file
    correlation_matrix = pd.read_excel(path, index_col=0)

    # Generate mask to make a triangle shaped matrix
    mask = np.triu(np.ones_like(correlation_matrix, dtype=bool))

    # Figure size
    plt.figure(figsize=(16, 9))

    # Generate heatmap from correlation matrix and set color, initialize the temperature bar, etc.
    heatmap = sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f", vmin=-0.5, vmax=0.5, mask=mask, linewidths=0.5, linecolor="White", cbar_kws={"shrink": .
8})
    heatmap.set_title(f'PCA {title_name}', fontdict={'fontsize':20}, pad=12)
    #turn xaxis so make it better readable
    plt.xticks(rotation=45, ha='right')
    #makes better layout
    plt.tight_layout()
    #saves file as file name
    plt.savefig(file_name, dpi=500, bbox_inches='tight')
    plt.show()


create_PCA_heatmap('/Users/vincentkellerer/Desktop/PCA/A) Bio oil/bio oil heatmap values.xlsx', 'Bio Oil', 'PCA_Bio_oil.png')
create_PCA_heatmap('/Users/vincentkellerer/Desktop/PCA/B) Syngas/Syngas heatmap values.xlsx', 'Syngas', 'PCA_Syngas.png')
create_PCA_heatmap('/Users/vincentkellerer/Desktop/PCA/C) Bio char/bio char heatmap values.xlsx', 'Bio Char', 'PCA_Bio_char.png')
```

NN PYROLYSIS

```python
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset, random_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error
import shap
import matplotlib.pyplot as plt

# Load the dataset
file_path = '/Users/vincentkellerer/Desktop/PythonSeminar/Python Project/cleaned_imputed_data.csv'
df = pd.read_csv(file_path)

# Assuming the input columns are from column index 2 to 17 (modify as needed)
input_columns = df.columns[2:17]
output_columns = df.columns[17:]

# Convert the dataframe to numpy arrays
X = df[input_columns].values
y = df[output_columns].values

# Feature scaling
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Convert to PyTorch tensors
X_tensor = torch.tensor(X_scaled, dtype=torch.float32)
y_tensor = torch.tensor(y, dtype=torch.float32)

# Prepare the dataset and dataloader
dataset = TensorDataset(X_tensor, y_tensor)
train_size = int(0.8 * len(dataset))
val_size = len(dataset) - train_size
train_dataset, val_dataset = random_split(dataset, [train_size, val_size])
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)

class FinalPyrolysisNN(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_size, n_layers, dropout_rate):
        super(FinalPyrolysisNN, self).__init__()
        layers = []
        layers.append(nn.Linear(input_dim, hidden_size))
        layers.append(nn.ReLU())
        layers.append(nn.Dropout(dropout_rate))
        for _ in range(n_layers - 1):
            layers.append(nn.Linear(hidden_size, hidden_size))
            layers.append(nn.ReLU())
            layers.append(nn.Dropout(dropout_rate))
        layers.append(nn.Linear(hidden_size, output_dim))
        self.model = nn.Sequential(*layers)

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

# Example best hyperparameters from Optuna study (replace with actual best params)
best_params = {
    'hidden_size': 256,
    'n_layers': 3,
    'dropout_rate': 0.3,
    'learning_rate': 0.001
}

input_dim = X.shape[1]
output_dim = y.shape[1]
model = FinalPyrolysisNN(input_dim, output_dim,
                         best_params['hidden_size'],
                         best_params['n_layers'],
                         best_params['dropout_rate'])

optimizer = optim.Adam(model.parameters(), lr=best_params['learning_rate'])
criterion = nn.MSELoss()

# Train the model
num_epochs = 200

for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        outputs = model(X_batch)
        loss = criterion(outputs, y_batch)
```

```python

input_dim = X.shape[1]
output_dim = y.shape[1]
model = FinalPyrolysisNN(input_dim, output_dim,
                         best_params['hidden_size'],
                         best_params['n_layers'],
                         best_params['dropout_rate'])

optimizer = optim.Adam(model.parameters(), lr=best_params['learning_rate'])
criterion = nn.MSELoss()

# Train the model
num_epochs = 200

for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        outputs = model(X_batch)
        loss = criterion(outputs, y_batch)
        loss.backward()
        optimizer.step()
        running_loss += loss.item() * X_batch.size(0)

    epoch_loss = running_loss / len(train_loader.dataset)
    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {epoch_loss:.4f}")

# Validate the model
model.eval()
val_loss = 0.0
with torch.no_grad():
    y_true, y_pred = [], []
    for X_batch, y_batch in val_loader:
        outputs = model(X_batch)
        loss = criterion(outputs, y_batch)
        val_loss += loss.item() * X_batch.size(0)
        y_true.append(y_batch.numpy())
        y_pred.append(outputs.numpy())

    y_true = np.concatenate(y_true)
    y_pred = np.concatenate(y_pred)
    val_loss /= len(val_loader.dataset)
    r2 = r2_score(y_true, y_pred)
    mae = mean_absolute_error(y_true, y_pred)
    rmse = np.sqrt(mean_squared_error(y_true, y_pred))

print(f"Validation Loss: {val_loss:.4f}")
print(f"R-Squared (R^2): {r2:.4f}")
print(f"Mean Absolute Error (MAE): {mae:.4f}")
print(f"Root Mean Squared Error (RMSE): {rmse:.4f}")

# Perform SHAP analysis
explainer = shap.DeepExplainer(model, X_tensor)
shap_values = explainer.shap_values(X_tensor)

# Plot the SHAP summary plot
plt.figure()
shap.summary_plot(shap_values, X_tensor.numpy(), feature_names=input_columns)
plt.savefig('shap_summary_plot.png')

# Plot the SHAP dependence plot for the first feature
plt.figure()
shap.dependence_plot(0, shap_values, X_tensor.numpy(), feature_names=input_columns)
plt.savefig('shap_dependence_plot_feature_0.png')

# Force plot for the first prediction
plt.figure()
shap.force_plot(explainer.expected_value[0], shap_values[0][0], X_tensor.numpy()[0], feature_names=input_columns, matplotlib=Tru
e)
plt.savefig('shap_force_plot_feature_0.png')

# Display the plots
plt.show()

```

```python

# %%
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset, random_split
from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error
from sklearn.preprocessing import StandardScaler
import numpy as np
import optuna
import pandas as pd

# %%
file_path = '/Users/vincentkellerer/Desktop/PythonSeminar/Python Project/cleaned_imputed_data.csv'
# Load the dataset
df = pd.read_csv(file_path)

# Assuming the input columns are from column index 2 to 17 (modify as needed)
input_columns = df.columns[2:17]
output_columns = df.columns[17:]

# Convert the dataframe to numpy arrays
X = df[input_columns].values
y = df[output_columns].values

# Feature scaling
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Convert to PyTorch tensors
X_tensor = torch.tensor(X_scaled, dtype=torch.float32)
y_tensor = torch.tensor(y, dtype=torch.float32)

# Prepare the dataset and dataloader
dataset = TensorDataset(X_tensor, y_tensor)
train_size = int(0.8 * len(dataset))
val_size = len(dataset) - train_size
train_dataset, val_dataset = random_split(dataset, [train_size, val_size])
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)

# Define an improved neural network model
class ImprovedPyrolysisNN(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(ImprovedPyrolysisNN, self).__init__()
        self.layer1 = nn.Linear(input_dim, 256)
        self.bn1 = nn.BatchNorm1d(256)
        self.layer2 = nn.Linear(256, 128)
        self.bn2 = nn.BatchNorm1d(128)
        self.layer3 = nn.Linear(128, 64)
        self.bn3 = nn.BatchNorm1d(64)
        self.layer4 = nn.Linear(64, 32)
        self.bn4 = nn.BatchNorm1d(32)
        self.layer5 = nn.Linear(32, output_dim)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.3)  # Adding dropout to prevent overfitting

    def forward(self, x):
        x = self.relu(self.bn1(self.layer1(x)))
        x = self.dropout(x)
        x = self.relu(self.bn2(self.layer2(x)))
        x = self.dropout(x)
        x = self.relu(self.bn3(self.layer3(x)))
        x = self.dropout(x)
        x = self.relu(self.bn4(self.layer4(x)))
        x = self.layer5(x)
        return x

input_dim = X.shape[1]
output_dim = y.shape[1]
model = ImprovedPyrolysisNN(input_dim, output_dim)

# Define loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Train the model
num_epochs = 200

for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        outputs = model(X_batch)
        loss = criterion(outputs, y_batch)
        loss.backward()
        optimizer.step()
        running_loss += loss.item() * X_batch.size(0)

    epoch_loss = running_loss / len(train_loader.dataset)
    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {epoch_loss:.4f}")



# %%
```

```python

# %%

#optimisation via optuna

def objective(trial):
    # Suggest hyperparameters
    n_layers = trial.suggest_int('n_layers', 2, 5)
    hidden_size = trial.suggest_int('hidden_size', 64, 512)
    dropout_rate = trial.suggest_float('dropout_rate', 0.1, 0.5)
    learning_rate = trial.suggest_float('learning_rate', 1e-5, 1e-3)

    class OptunaPyrolysisNN(nn.Module):
        def __init__(self, input_dim, output_dim, hidden_size, n_layers, dropout_rate):
            super(OptunaPyrolysisNN, self).__init__()
            layers = []
            layers.append(nn.Linear(input_dim, hidden_size))
            layers.append(nn.ReLU())
            layers.append(nn.Dropout(dropout_rate))
            for _ in range(n_layers - 1):
                layers.append(nn.Linear(hidden_size, hidden_size))
                layers.append(nn.ReLU())
                layers.append(nn.Dropout(dropout_rate))
            layers.append(nn.Linear(hidden_size, output_dim))
            self.model = nn.Sequential(*layers)

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

    model = OptunaPyrolysisNN(input_dim, output_dim, hidden_size, n_layers, dropout_rate)
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
    criterion = nn.MSELoss()

    # Training loop
    num_epochs = 50
    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        for X_batch, y_batch in train_loader:
            optimizer.zero_grad()
            outputs = model(X_batch)
            loss = criterion(outputs, y_batch)
            loss.backward()
            optimizer.step()
            running_loss += loss.item() * X_batch.size(0)

    # Validation loss
    model.eval()
    val_loss = 0.0
    with torch.no_grad():
        for X_batch, y_batch in val_loader:
            outputs = model(X_batch)
            loss = criterion(outputs, y_batch)
            val_loss += loss.item() * X_batch.size(0)
    val_loss /= len(val_loader.dataset)
    return val_loss

# Create a study and optimize the objective function
study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=50)

print(f'Best trial: {study.best_trial.value}')
print(f'Best hyperparameters: {study.best_trial.params}')


# %%
best_params = study.best_trial.params

class FinalPyrolysisNN(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_size, n_layers, dropout_rate):
        super(FinalPyrolysisNN, self).__init__()
        layers = []
        layers.append(nn.Linear(input_dim, hidden_size))
        layers.append(nn.ReLU())
        layers.append(nn.Dropout(dropout_rate))
        for _ in range(n_layers - 1):
            layers.append(nn.Linear(hidden_size, hidden_size))
            layers.append(nn.ReLU())
            layers.append(nn.Dropout(dropout_rate))
        layers.append(nn.Linear(hidden_size, output_dim))
        self.model = nn.Sequential(*layers)

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

model = FinalPyrolysisNN(input_dim, output_dim,
                         best_params['hidden_size'],
                         best_params['n_layers'],
                         best_params['dropout_rate'])

optimizer = optim.Adam(model.parameters(), lr=best_params['learning_rate'])
criterion = nn.MSELoss()

# Train the final model
num_epochs = 200
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        outputs = model(X_batch)
        loss = criterion(outputs, y_batch)
        loss.backward()
        optimizer.step()
        running_loss += loss.item() * X_batch.size(0)

    epoch_loss = running_loss / len(train_loader.dataset)
    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {epoch_loss:.4f}")

# Validate the final model
model.eval()
val_loss = 0.0
with torch.no_grad():
    for X_batch, y_batch in val_loader:
        outputs = model(X_batch)
        loss = criterion(outputs, y_batch)
        val_loss += loss.item() * X_batch.size(0)
val_loss /= len(val_loader.dataset)
print(f"Validation Loss: {val_loss:.4f}")

# Make predictions on training data
with torch.no_grad():
    y_train_pred = model(X_tensor).numpy()

# Calculate R^2
r2 = r2_score(y_tensor.numpy(), y_train_pred)
mae = mean_absolute_error(y_tensor.numpy(), y_train_pred)
rmse = np.sqrt(mean_squared_error(y_tensor.numpy(), y_train_pred))
print(f"R-Squared (R^2): {r2:.4f}")
print(f"Mean Absolute Error (MAE): {mae:.4f}")
print(f"Root Mean Squared Error (RMSE): {rmse:.4f}")


```

```
import numpy as np
import pandas as pd
from sklearn.svm import SVR
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error


df = pd.read_excel('/content/cleaned_imputed_data.xlsx')

data.head()
print(data.columns)

target_columns = ['Bio-oil yield (%)',
       'Syngas yield (%)', 'Syngas composition (H₂, mol%)',
       'Syngas composition (CH₄, mol%)', 'Syngas composition (CO₂, mol%)',
       'Syngas composition (CO, mol%)', 'Biochar yield (%)',
       'Biochar calorific value (MJ/kg)', 'Biochar H/C ratio (-)',
       'Biochar H/N ratio (-)', 'Biochar O/C ratio (-)']
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-6-31871e0a451d> in <cell line: 3>()
      1 df = pd.read_excel('/content/cleaned_imputed_data.xlsx')
      2
----> 3 data.head()
      4 print(data.columns)
      5

NameError: name 'data' is not defined
```

```
# Load the California Housing dataset
california = fetch_california_housing()
X = california.data
y = california.target

# Split the dataset 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)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create and train the SVR model
svr = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=0.1)
svr.fit(X_train, y_train)

# Make predictions
y_train_pred = svr.predict(X_train)
y_test_pred = svr.predict(X_test)

# Calculate R²
r2_train = r2_score(y_train, y_train_pred)
r2_test = r2_score(y_test, y_test_pred)

# Calculate RMSE
rmse_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))

# Calculate RRMSE
rrmse_train = rmse_train / np.mean(y_train)
rrmse_test = rmse_test / np.mean(y_test)

# Create a DataFrame to compare the performance metrics
performance_df = pd.DataFrame({
    'Metric': ['R²', 'RMSE', 'RRMSE'],
    'Training': [r2_train, rmse_train, rrmse_train],
    'Testing': [r2_test, rmse_test, rrmse_test]
})

print(performance_df)
```

```
  Metric  Training   Testing
0     R²  0.794794  0.752371
1   RMSE  0.523751  0.569644
2  RRMSE  0.252782  0.277199
```

```
# Step 1: Install necessary libraries (if not already installed)
!pip install pandas openpyxl scikit-learn matplotlib

# Step 2: Upload the dataset
from google.colab import files

# Upload file
uploaded = files.upload()

# Step 3: Read the dataset into a DataFrame
import pandas as pd

# Replace 'your_dataset.xlsx' with the name of your uploaded file


# Display the DataFrame to understand its structure
print("DataFrame Head:")
print(df.head())

# Step 4: Assign variables to X and multiple y variables
# Define your feature columns and target columns
feature_columns = ['Carbon content (wt%)',
       'Hydrogen content (wt%)', 'Nitrogen content (wt%)',
       'Oxygen content (wt%)', 'Sulfur content (wt%)', 'Volatile matter (wt%)',
       'Fixed carbon (wt%)', 'Ash content (wt%)', 'Reaction temperature (°C)',
       'Microwave power (W)', 'Reaction time (min)',
       'Microwave absorber percentage (%)',
       'Dielectric constant of absorber (ε’)',
       'Dielectric loss factor of absorber (ε“)']  # Replace with your actual feature column names
target_columns = ['Syngas yield (%)' ,'Bio-oil yield (%)']  # Replace with your actual target column names

# Assign the feature columns to X and the target columns to y
X = df[feature_columns].values
y = df[target_columns].values

# Display the shapes of X and y to confirm
print("X shape:", X.shape)
print("y shape:", y.shape)

# Split the dataset into training and testing sets
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize the features
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Initialize lists to store results
train_preds = []
test_preds = []
metrics = []

# Train and predict for each target variable
from sklearn.svm import SVR
from sklearn.metrics import r2_score, mean_squared_error
import numpy as np

for i, target in enumerate(target_columns):
    svr = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=0.1)
    svr.fit(X_train, y_train[:, i])

    # Predictions
    y_train_pred = svr.predict(X_train)
    y_test_pred = svr.predict(X_test)

    train_preds.append(y_train_pred)
    test_preds.append(y_test_pred)

    # Evaluate the model
    r2_train = r2_score(y_train[:, i], y_train_pred)
    r2_test = r2_score(y_test[:, i], y_test_pred)
    rmse_train = np.sqrt(mean_squared_error(y_train[:, i], y_train_pred))
    rmse_test = np.sqrt(mean_squared_error(y_test[:, i], y_test_pred))
    rrmse_train = rmse_train / np.mean(y_train[:, i])
    rrmse_test = rmse_test / np.mean(y_test[:, i])

    metrics.append({
        'Target': target,
```

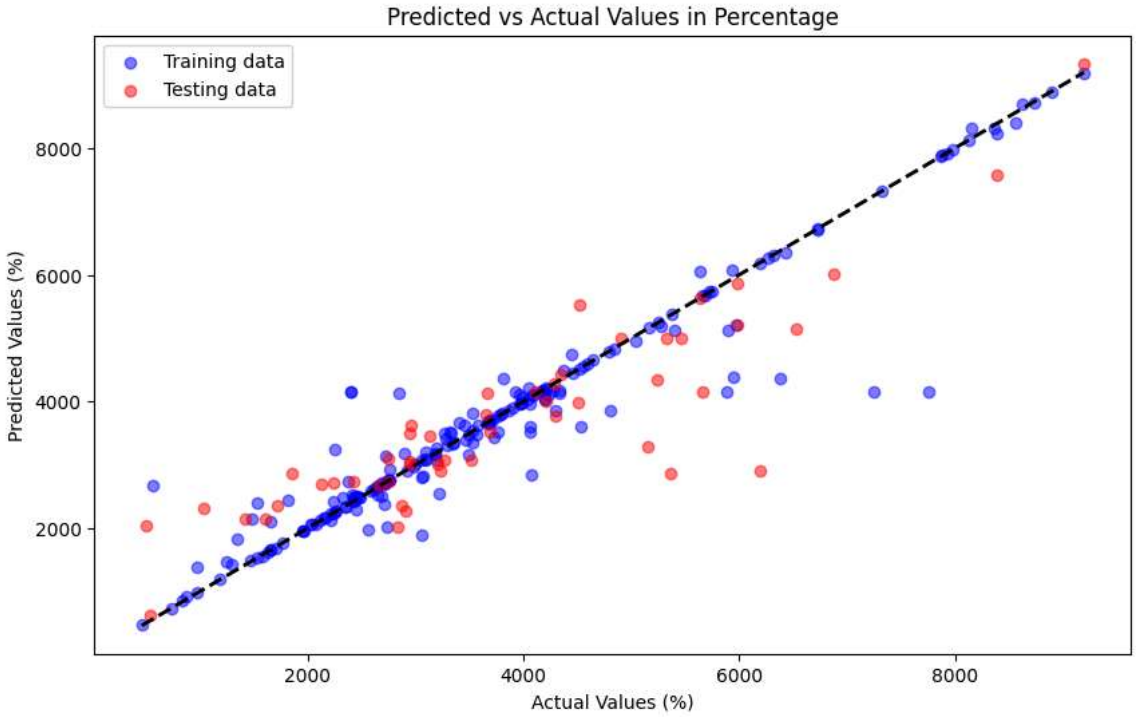
```
            'R² Train': r2_train,
            'R² Test': r2_test,
            'RMSE Train': rmse_train,
            'RMSE Test': rmse_test,
            'RRMSE Train': rrmse_train,
            'RRMSE Test': rrmse_test
        })

    # Create a DataFrame to compare the performance metrics
    performance_df = pd.DataFrame(metrics)
    print(performance_df)

    # Gather all predictions together for visualization
    train_preds = np.array(train_preds).T
    test_preds = np.array(test_preds).T

    # Plot the results
    import matplotlib.pyplot as plt

    for i, target in enumerate(target_columns):
        # Convert actual and predicted values to percentages for plotting
        y_train_percentage = y_train[:, i] * 100
        y_train_pred_percentage = train_preds[:, i] * 100
        y_test_percentage = y_test[:, i] * 100
        y_test_pred_percentage = test_preds[:, i] * 100

        # Create scatter plot to compare predicted vs actual values in percentages
        plt.figure(figsize=(10, 6))
        plt.scatter(y_train_percentage, y_train_pred_percentage, color='blue', alpha=0.5, label='Training data')
        plt.scatter(y_test_percentage, y_test_pred_percentage, color='red', alpha=0.5, label='Testing data')
        plt.plot([y_train_percentage.min(), y_train_percentage.max()], [y_train_percentage.min(), y_train_percentage.max()], 'k--', lw=2)
        plt.xlabel('Actual Values (%)')
        plt.ylabel('Predicted Values (%)')
        plt.title(f'Predicted vs Actual Values for {target} in Percentage')
        plt.legend()
        plt.show()
```

```
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (2.0.3)
Requirement already satisfied: openpyxl in /usr/local/lib/python3.10/dist-packages (3.1.3)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.10/dist-packages (1.2.2)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist-packages (3.7.1)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packages (from pandas) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas) (2023.4)
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-packages (from pandas) (2024.1)
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.10/dist-packages (from pandas) (1.25.2)
Requirement already satisfied: et-xmlfile in /usr/local/lib/python3.10/dist-packages (from openpyxl) (1.1.0)
Requirement already satisfied: scipy>=1.3.2 in /usr/local/lib/python3.10/dist-packages (from scikit-learn) (1.11.4)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from scikit-learn) (1.4.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn) (3.5.0)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (1.2.1)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (4.53.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (1.4.5)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (24.1)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (9.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (3.1.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.2->pandas) (1.16.0)
Choose Files No file chosen  Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.svm import SVR
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error

# Load the California Housing dataset
#california = fetch_california_housing()
#X = california.data
#y = california.target

feature_columns = ['Carbon content (wt%)',
       'Hydrogen content (wt%)', 'Nitrogen content (wt%)',
       'Oxygen content (wt%)', 'Sulfur content (wt%)', 'Volatile matter (wt%)',
       'Fixed carbon (wt%)', 'Ash content (wt%)', 'Reaction temperature (°C)',
       'Microwave power (W)', 'Reaction time (min)',
       'Microwave absorber percentage (%)',
       'Dielectric constant of absorber (ε’)',
       'Dielectric loss factor of absorber (ε“)']  # Replace with your actual feature column names
target_column = 'Syngas yield (%)'  # Replace with your actual target column name
```

```
# Assign the feature columns to X and the target column to y
X = df[feature_columns].values
y = df[target_column].values
# Split the dataset into training and testing sets
# focus on the parameter adjustion
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create and train the SVR model
svr = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=0.1)
svr.fit(X_train, y_train)

# Make predictions
y_train_pred = svr.predict(X_train)
y_test_pred = svr.predict(X_test)

# Calculate R²
r2_train = r2_score(y_train, y_train_pred)
r2_test = r2_score(y_test, y_test_pred)

# Calculate RMSE
rmse_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))

# Calculate RRMSE
rrmse_train = rmse_train / np.mean(y_train)
rrmse_test = rmse_test / np.mean(y_test)

# Create a DataFrame to compare the performance metrics
performance_df = pd.DataFrame({
    'Metric': ['R²', 'RMSE', 'RRMSE'],
    'Training': [r2_train, rmse_train, rrmse_train],
    'Testing': [r2_test, rmse_test, rrmse_test]
})

print(performance_df)

# Convert actual and predicted values to percentages for plotting
y_train_percentage = y_train * 100
y_train_pred_percentage = y_train_pred * 100
y_test_percentage = y_test * 100
y_test_pred_percentage = y_test_pred * 100

# Create scatter plot to compare predicted vs actual values in percentages
plt.figure(figsize=(10, 6))
plt.scatter(y_train_percentage, y_train_pred_percentage, color='blue', alpha=0.5, label='Training data')
plt.scatter(y_test_percentage, y_test_pred_percentage, color='red', alpha=0.5, label='Testing data')
plt.plot([y.min() * 100, y.max() * 100], [y.min() * 100, y.max() * 100], 'k--', lw=2)
plt.xlabel('Actual Values (%)')
plt.ylabel('Predicted Values (%)')
plt.title('Predicted vs Actual Values in Percentage')
plt.legend()
plt.show()
```

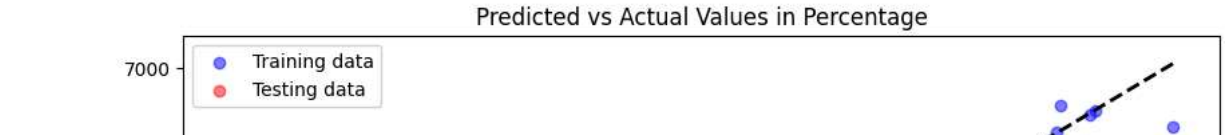
	Metric	Training	Testing
0	R^2	0.915195	0.767129
1	RMSE	5.404144	8.872308
2	RRMSE	0.144605	0.229255

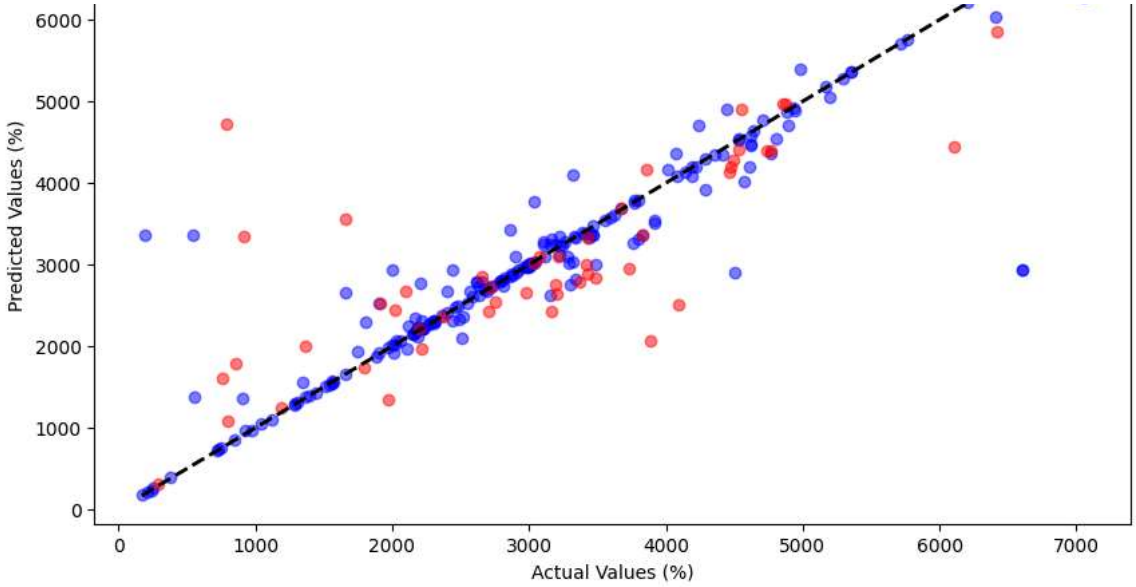


```
# Step 1: Install necessary libraries (if not already installed)
!pip install pandas openpyxl scikit-learn matplotlib

# Step 2: Upload the dataset
from google.colab import files

# Upload file
data = pd.read_excel('/content/cleaned_imputed_data.xlsx')

# Step 3: Read the dataset into a DataFrame
import pandas as pd

# Replace 'your_dataset.xlsx' with the name of your uploaded file
df = pd.read_excel('/content/cleaned_imputed_data.xlsx')

# Display the DataFrame to understand its structure
print("DataFrame Head:")
print(df.head())

# Step 4: Assign variables to X and y
# Define your feature columns and target column
feature_columns = ['Carbon content (wt%)',
       'Hydrogen content (wt%)', 'Nitrogen content (wt%)',
       'Oxygen content (wt%)', 'Sulfur content (wt%)', 'Volatile matter (wt%)',
       'Fixed carbon (wt%)', 'Ash content (wt%)', 'Reaction temperature (°C)',
       'Microwave power (W)', 'Reaction time (min)',
       'Microwave absorber percentage (%)',
       'Dielectric constant of absorber (ε’)',
       'Dielectric loss factor of absorber (ε“)']  # Replace with your actual feature column names
target_column = 'Bio-oil yield (%)'  # Replace with your actual target column name

# Assign the feature columns to X and the target column to y
X = df[feature_columns].values
y = df[target_column].values

# Display the shapes of X and y to confirm
print("X shape:", X.shape)
print("y shape:", y.shape)

# Step 5: Split the dataset into training and testing sets
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 6: Standardize the features
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Step 7: Create and train the SVR model
from sklearn.svm import SVR

svr = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=0.1)
svr.fit(X_train, y_train)

# Step 8: Make predictions
y_train_pred = svr.predict(X_train)
y_test_pred = svr.predict(X_test)

# Step 9: Evaluate the model
from sklearn.metrics import r2_score, mean_squared_error

# Calculate R²
r2_train = r2_score(y_train, y_train_pred)
r2_test = r2_score(y_test, y_test_pred)

# Calculate RMSE
rmse_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))

# Calculate RRMSE
rrmse_train = rmse_train / np.mean(y_train)
rrmse_test = rmse_test / np.mean(y_test)

# Create a DataFrame to compare the performance metrics
performance_df = pd.DataFrame({
    'Metric': ['R²', 'RMSE', 'RRMSE'],
    'Training': [r2_train, rmse_train, rrmse_train],
    'Testing': [r2_test, rmse_test, rrmse_test]
})
```

```
print(performance_df)

# Step 10: Visualize the predictions vs actual values in percentage
import matplotlib.pyplot as plt

# Convert actual and predicted values to percentages for plotting
y_train_percentage = y_train * 100
y_train_pred_percentage = y_train_pred * 100
y_test_percentage = y_test * 100
y_test_pred_percentage = y_test_pred * 100

# Create scatter plot to compare predicted vs actual values in percentages
plt.figure(figsize=(10, 6))
plt.scatter(y_train_percentage, y_train_pred_percentage, color='blue', alpha=0.5, label='Training data')
plt.scatter(y_test_percentage, y_test_pred_percentage, color='red', alpha=0.5, label='Testing data')
plt.plot([y.min() * 100, y.max() * 100], [y.min() * 100, y.max() * 100], 'k--', lw=2)
plt.xlabel('Actual Values (%)')
plt.ylabel('Predicted Values (%)')
plt.title('Predicted vs Actual Values in Percentage')
plt.legend()
plt.show()
```

```
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (2.0.3)
Requirement already satisfied: openpyxl in /usr/local/lib/python3.10/dist-packages (3.1.3)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.10/dist-packages (1.2.2)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist-packages (3.7.1)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packages (from pandas) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas) (2023.4)
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-packages (from pandas) (2024.1)
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.10/dist-packages (from pandas) (1.25.2)
Requirement already satisfied: et-xmlfile in /usr/local/lib/python3.10/dist-packages (from openpyxl) (1.1.0)
Requirement already satisfied: scipy>=1.3.2 in /usr/local/lib/python3.10/dist-packages (from scikit-learn) (1.11.4)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from scikit-learn) (1.4.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn) (3.5.0)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (1.2.1)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (4.53.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (1.4.5)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (24.1)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (9.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (3.1.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.2->pandas) (1.16.0)
DataFrame Head:
                  Reference (DOI) Biomass type  Carbon content (wt%)  \
0  10.1016/j.apenergy.2020.114855       Sludge                 33.14
1  10.1016/j.apenergy.2020.114855       Sludge                 33.14
2  10.1016/j.apenergy.2020.114855       Sludge                 33.14
3  10.1016/j.apenergy.2020.114855       Sludge                 33.14
4  10.1016/j.apenergy.2020.114855       Sludge                 33.14

   Hydrogen content (wt%)  Nitrogen content (wt%)  Oxygen content (wt%)  \
0                    5.58                    4.85                 55.75
1                    5.58                    4.85                 55.75
2                    5.58                    4.85                 55.75
3                    5.58                    4.85                 55.75
4                    5.58                    4.85                 55.75

   Sulfur content (wt%)  Volatile matter (wt%)  Fixed carbon (wt%)  \
0                  0.68                  51.09                1.94
1                  0.68                  51.09                1.94
2                  0.68                  51.09                1.94
3                  0.68                  51.09                1.94
4                  0.68                  51.09                1.94

   Ash content (wt%)  ...  Syngas yield (%)  Syngas composition (H₂, mol%)  \
0              46.97  ...             37.87                           4.70
1              46.97  ...             44.53                          12.20
2              46.97  ...             47.94                          20.44
3              46.97  ...             52.76                          27.94
4              46.97  ...             37.16                          15.71

   Syngas composition (CH₄, mol%)  Syngas composition (CO₂, mol%)  \
0                            5.58                            6.76
1                            8.08                            5.44
2                           11.76                            5.14
3                           13.67                            5.00
4                            5.38                            6.73

   Syngas composition (CO, mol%)  Biochar yield (%)  \
0                          14.41              48.08
1                          19.26              40.70
2                          21.17              38.86
3                          25.00              37.73
4                          17.95              49.21

   Biochar calorific value (MJ/kg)  Biochar H/C ratio (-)  \
0                         4.590000                   1.34
1                         3.050000                   1.26
2                         1.000000                   1.12
3                         0.651554                   1.19
4                         3.750000                   1.89

   Biochar H/N ratio (-)  Biochar O/C ratio (-)
0                  13.35                   1.94
1                  13.84                   2.22
2                  11.09                   2.70
3                   9.65                   3.35
4                  16.69                   2.34

[5 rows x 27 columns]
X shape: (248, 14)
y shape: (248,)
  Metric  Training   Testing
0     R²  0.847499  0.575006
1   RMSE  5.434442  9.092239
2  RRMSE  0.180284  0.298321
```

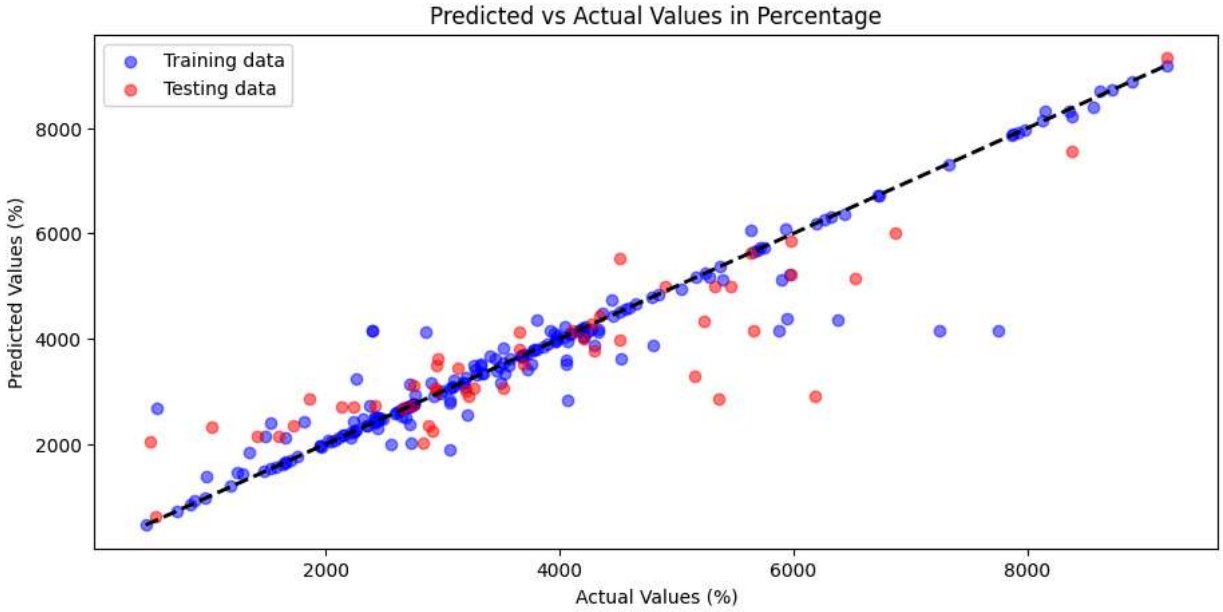



Predicted Values (%)
Actual Values (%)
0
1000
2000
3000
4000
5000
6000
7000

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.svm import SVR
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error

# Assuming `df` is already defined as your DataFrame
# Replace the following line with the actual code to load your data if not already done
# df = pd.read_csv('your_data.csv')

feature_columns = ['Carbon content (wt%)',
                   'Hydrogen content (wt%)', 'Nitrogen content (wt%)',
                   'Oxygen content (wt%)', 'Sulfur content (wt%)', 'Volatile matter (wt%)',
                   'Fixed carbon (wt%)', 'Ash content (wt%)', 'Reaction temperature (°C)',
                   'Microwave power (W)', 'Reaction time (min)',
                   'Microwave absorber percentage (%)',
                   'Dielectric constant of absorber (ε’)',
                   'Dielectric loss factor of absorber (ε“)']  # Replace with your actual feature column names
target_column = 'Syngas yield (%)'  # Replace with your actual target column name

# Assign the feature columns to X and the target column to y
X = df[feature_columns].values
y = df[target_column].values

# Split the dataset 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)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create and train the SVR model
svr = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=0.1)
svr.fit(X_train, y_train)

# Make predictions
y_train_pred = svr.predict(X_train)
y_test_pred = svr.predict(X_test)

# Calculate R²
r2_train = r2_score(y_train, y_train_pred)
r2_test = r2_score(y_test, y_test_pred)

# Calculate RMSE
rmse_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))

# Calculate RRMSE
rrmse_train = rmse_train / np.mean(y_train)
rrmse_test = rmse_test / np.mean(y_test)

# Create a DataFrame to compare the performance metrics
performance_df = pd.DataFrame({
    'Metric': ['R²', 'RMSE', 'RRMSE'],
    'Training': [r2_train, rmse_train, rrmse_train],
    'Testing': [r2_test, rmse_test, rrmse_test]
})

print(performance_df)

# Convert actual and predicted values to percentages for plotting
y_train_percentage = y_train * 100
y_train_pred_percentage = y_train_pred * 100
y_test_percentage = y_test * 100
y_test_pred_percentage = y_test_pred * 100

# Create scatter plot to compare predicted vs actual values in percentages
plt.figure(figsize=(12, 8))
plt.scatter(y_train_percentage, y_train_pred_percentage, color='blue', alpha=0.5, label='Training data')
plt.scatter(y_test_percentage, y_test_pred_percentage, color='red', alpha=0.5, label='Testing data')
plt.plot([min(y) * 100, max(y) * 100], [min(y) * 100, max(y) * 100], 'k--', lw=2)
plt.xlabel('Actual Values (%)')
plt.ylabel('Predicted Values (%)')
plt.title('Predicted vs Actual Values in Percentage')
plt.legend()

# Add table with performance metrics
table_data = [['R²', f'{r2_train:.3f}', f'{r2_test:.3f}'],
              ['RMSE', f'{rmse_train:.3f}', f'{rmse_test:.3f}'],
```

```
            ['RRMSE', f'{rrmse_train:.3f}', f'{rrmse_test:.3f}']]
table = plt.table(cellText=table_data,
                  colLabels=['Metric', 'Training', 'Testing'],
                  cellLoc='center',
                  loc='bottom',
                  bbox=[0.0, -0.3, 1, 0.15])  # Adjust bbox to control position and size

# Adjust layout to make room for the table
plt.subplots_adjust(left=0.2, bottom=0.4)

# Reduce font size for the table
table.auto_set_font_size(False)
table.set_fontsize(10)
table.scale(1, 1.5)  # Scale the table size

plt.show()
```

```
   Metric  Training   Testing
0      R²  0.915195  0.767129
1    RMSE  5.404144  8.872308
2   RRMSE  0.144605  0.229255
```

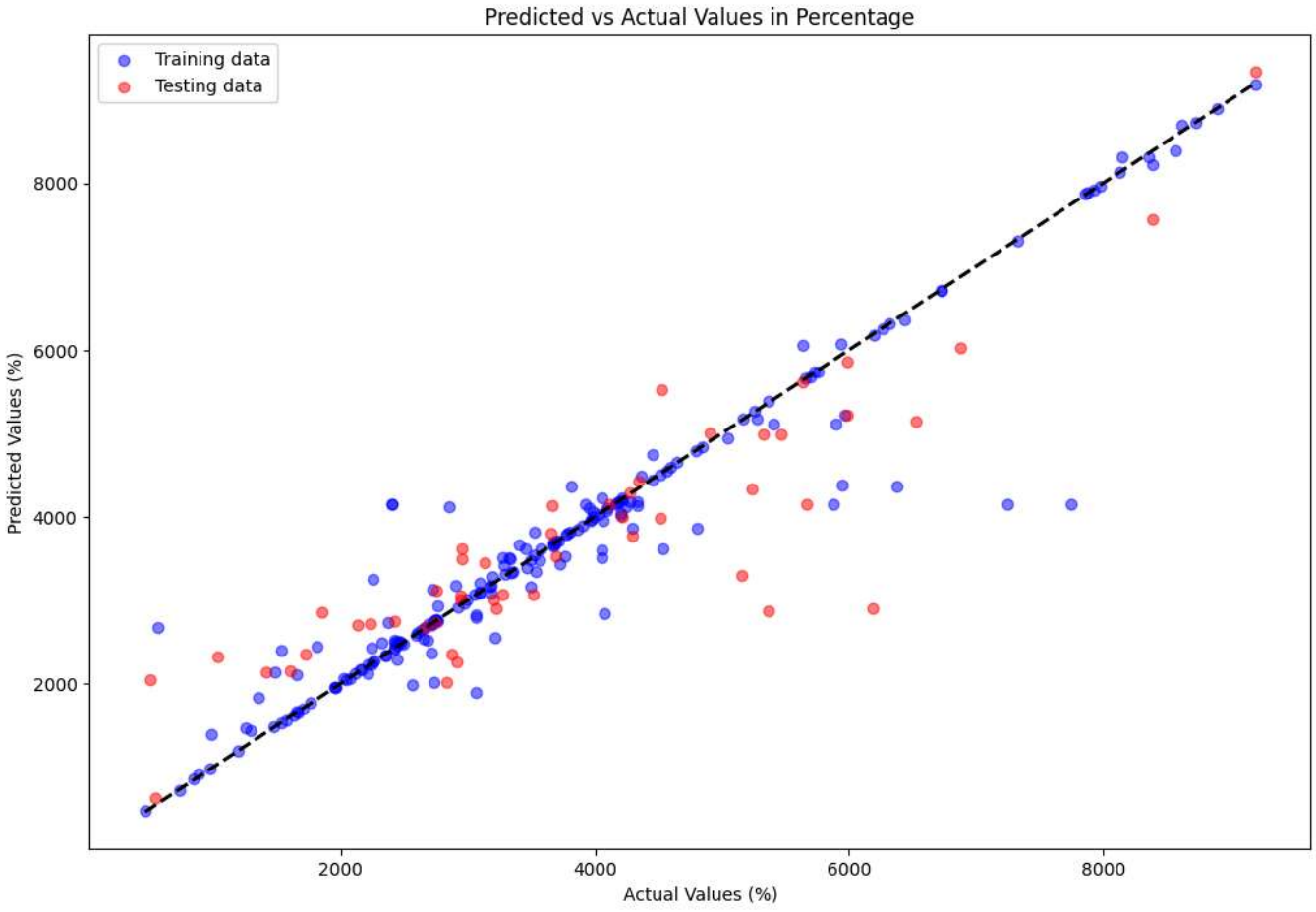


Metric	Training	Testing
R²	0.915	0.767
RMSE	5.404	8.872
RRMSE	0.145	0.229

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.svm import SVR
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error

# Assuming `df` is already defined as your DataFrame
# Replace the following line with the actual code to load your data if not already done
# df = pd.read_csv('your_data.csv')

feature_columns = ['Carbon content (wt%)',
                   'Hydrogen content (wt%)', 'Nitrogen content (wt%)',
                   'Oxygen content (wt%)', 'Sulfur content (wt%)', 'Volatile matter (wt%)',
                   'Fixed carbon (wt%)', 'Ash content (wt%)', 'Reaction temperature (°C)',
                   'Microwave power (W)', 'Reaction time (min)',
                   'Microwave absorber percentage (%)',
                   'Dielectric constant of absorber (ε′)',
                   'Dielectric loss factor of absorber (ε‟)']  # Replace with your actual feature column names
target_column = 'Syngas yield (%)'  # Replace with your actual target column name

# Assign the feature columns to X and the target column to y
X = df[feature_columns].values
y = df[target_column].values

# Split the dataset 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)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create and train the SVR model
svr = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=0.1)
svr.fit(X_train, y_train)

# Make predictions
y_train_pred = svr.predict(X_train)
y_test_pred = svr.predict(X_test)

# Calculate R²
r2_train = r2_score(y_train, y_train_pred)
r2_test = r2_score(y_test, y_test_pred)

# Calculate RMSE
rmse_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))

# Calculate RRMSE
rrmse_train = rmse_train / np.mean(y_train)
rrmse_test = rmse_test / np.mean(y_test)

# Create a DataFrame to compare the performance metrics
performance_df = pd.DataFrame({
    'Metric': ['R²', 'RMSE', 'RRMSE'],
    'Training': [r2_train, rmse_train, rrmse_train],
    'Testing': [r2_test, rmse_test, rrmse_test]
})

print(performance_df)

# Convert actual and predicted values to percentages for plotting
y_train_percentage = y_train * 100
y_train_pred_percentage = y_train_pred * 100
y_test_percentage = y_test * 100
y_test_pred_percentage = y_test_pred * 100

# Create scatter plot to compare predicted vs actual values in percentages
fig, ax = plt.subplots(figsize=(12, 8))
ax.scatter(y_train_percentage, y_train_pred_percentage, color='blue', alpha=0.5, label='Training data')
ax.scatter(y_test_percentage, y_test_pred_percentage, color='red', alpha=0.5, label='Testing data')
ax.plot([min(y) * 100, max(y) * 100], [min(y) * 100, max(y) * 100], 'k--', lw=2)
ax.set_xlabel('Actual Values (%)')
ax.set_ylabel('Predicted Values (%)')
ax.set_title('Predicted vs Actual Values in Percentage')
ax.legend()

# Add table with performance metrics
table_data = [['R²', f'{r2_train:.3f}', f'{r2_test:.3f}'],
              ['RMSE', f'{rmse_train:.3f}', f'{rmse_test:.3f}'],
```

```
            ['RRMSE', f'{rrmse_train:.3f}', f'{rrmse_test:.3f}']]
table = ax.table(cellText=table_data,
                 colLabels=['Metric', 'Training', 'Testing'],
                 cellLoc='center',
                 loc='lower right',
                 bbox=[0.8, -0.3, 0.2, 0.2])  # Adjust bbox to control position and size

# Adjust layout to make room for the table
plt.subplots_adjust(left=0.1, right=0.9, top=0.9, bottom=0.1)

# Reduce font size for the table
table.auto_set_font_size(False)
table.set_fontsize(10)
table.scale(1, 1.5)  # Scale the table size

plt.show()
```

```
   Metric  Training   Testing
0      R²  0.915195  0.767129
1    RMSE  5.404144  8.872308
2   RRMSE  0.144605  0.229255
```



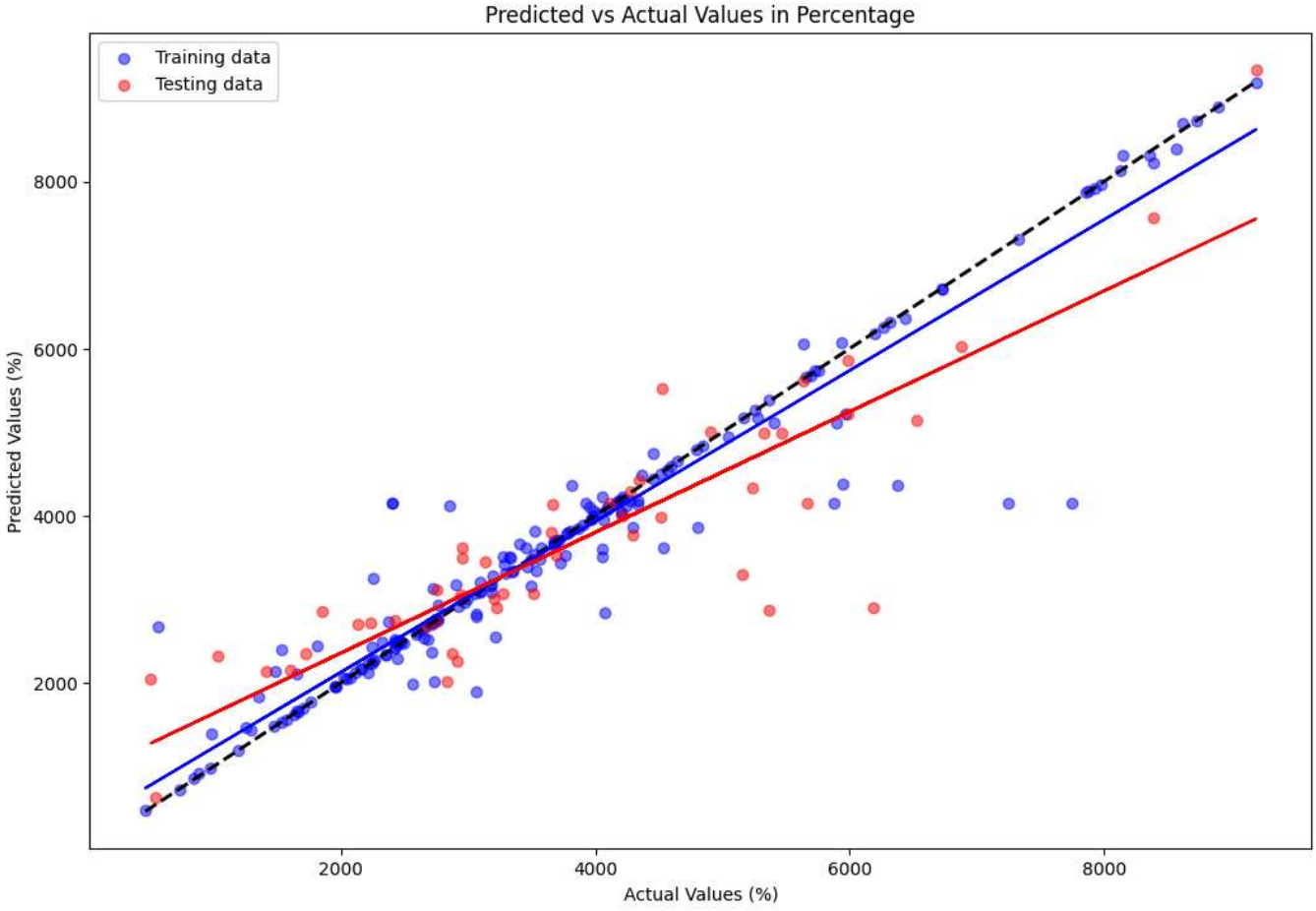
Metric	Training	Testing
R²	0.915	0.767
RMSE	5.404	8.872
RRMSE	0.145	0.229

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.svm import SVR
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error

# Assuming `df` is already defined as your DataFrame
# Replace the following line with the actual code to load your data if not already done
# df = pd.read_csv('your_data.csv')

feature_columns = ['Carbon content (wt%)',
```

```
                      'Hydrogen content (wt%)', 'Nitrogen content (wt%)',
                      'Oxygen content (wt%)', 'Sulfur content (wt%)', 'Volatile matter (wt%)',
                      'Fixed carbon (wt%)', 'Ash content (wt%)', 'Reaction temperature (°C)',
                      'Microwave power (W)', 'Reaction time (min)',
                      'Microwave absorber percentage (%)',
                      'Dielectric constant of absorber (ε′)',
                      'Dielectric loss factor of absorber (ε‟)']  # Replace with your actual feature column names
target_column = 'Syngas yield (%)'  # Replace with your actual target column name

# Assign the feature columns to X and the target column to y
X = df[feature_columns].values
y = df[target_column].values

# Split the dataset 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)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create and train the SVR model
svr = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=0.1)
svr.fit(X_train, y_train)

# Make predictions
y_train_pred = svr.predict(X_train)
y_test_pred = svr.predict(X_test)

# Calculate R²
r2_train = r2_score(y_train, y_train_pred)
r2_test = r2_score(y_test, y_test_pred)

# Calculate RMSE
rmse_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))

# Calculate RRMSE
rrmse_train = rmse_train / np.mean(y_train)
rrmse_test = rmse_test / np.mean(y_test)

# Create a DataFrame to compare the performance metrics
performance_df = pd.DataFrame({
    'Metric': ['R²', 'RMSE', 'RRMSE'],
    'Training': [r2_train, rmse_train, rrmse_train],
    'Testing': [r2_test, rmse_test, rrmse_test]
})

print(performance_df)

# Convert actual and predicted values to percentages for plotting
y_train_percentage = y_train * 100
y_train_pred_percentage = y_train_pred * 100
y_test_percentage = y_test * 100
y_test_pred_percentage = y_test_pred * 100

# Create scatter plot to compare predicted vs actual values in percentages
fig, ax = plt.subplots(figsize=(12, 8))
ax.scatter(y_train_percentage, y_train_pred_percentage, color='blue', alpha=0.5, label='Training data')
ax.scatter(y_test_percentage, y_test_pred_percentage, color='red', alpha=0.5, label='Testing data')
ax.plot([min(y) * 100, max(y) * 100], [min(y) * 100, max(y) * 100], 'k--', lw=2)
ax.set_xlabel('Actual Values (%)')
ax.set_ylabel('Predicted Values (%)')
ax.set_title('Predicted vs Actual Values in Percentage')
ax.legend()

# Add table with performance metrics
table_data = [['R²', f'{r2_train:.3f}', f'{r2_test:.3f}'],
              ['RMSE', f'{rmse_train:.3f}', f'{rmse_test:.3f}'],
              ['RRMSE', f'{rrmse_train:.3f}', f'{rrmse_test:.3f}']]
table = ax.table(cellText=table_data,
                 colLabels=['Metric', 'Training', 'Testing'],
                 cellLoc='center',
                 loc='lower right',
                 bbox=[0.8, -0.3, 0.2, 0.2])  # Adjust bbox to control position and size

# Add trend lines
# Training data trend line
train_polyfit = np.polyfit(y_train_percentage, y_train_pred_percentage, 1)
train_polyval = np.polyval(train_polyfit, y_train_percentage)
ax.plot(y_train_percentage, train_polyval, color='blue', label='Training trend line')

# Testing data trend line
```

```
# Testing data trend line
test_polyfit = np.polyfit(y_test_percentage, y_test_pred_percentage, 1)
test_polyval = np.polyval(test_polyfit, y_test_percentage)
ax.plot(y_test_percentage, test_polyval, color='red', label='Testing trend line')

# Adjust layout to make room for the table
plt.subplots_adjust(left=0.1, right=0.9, top=0.9, bottom=0.1)

# Reduce font size for the table
table.auto_set_font_size(False)
table.set_fontsize(10)
table.scale(1, 1.5)  # Scale the table size

plt.show()
```

```
  Metric  Training   Testing
0     R²  0.915195  0.767129
1   RMSE  5.404144  8.872308
2  RRMSE  0.144605  0.229255
```

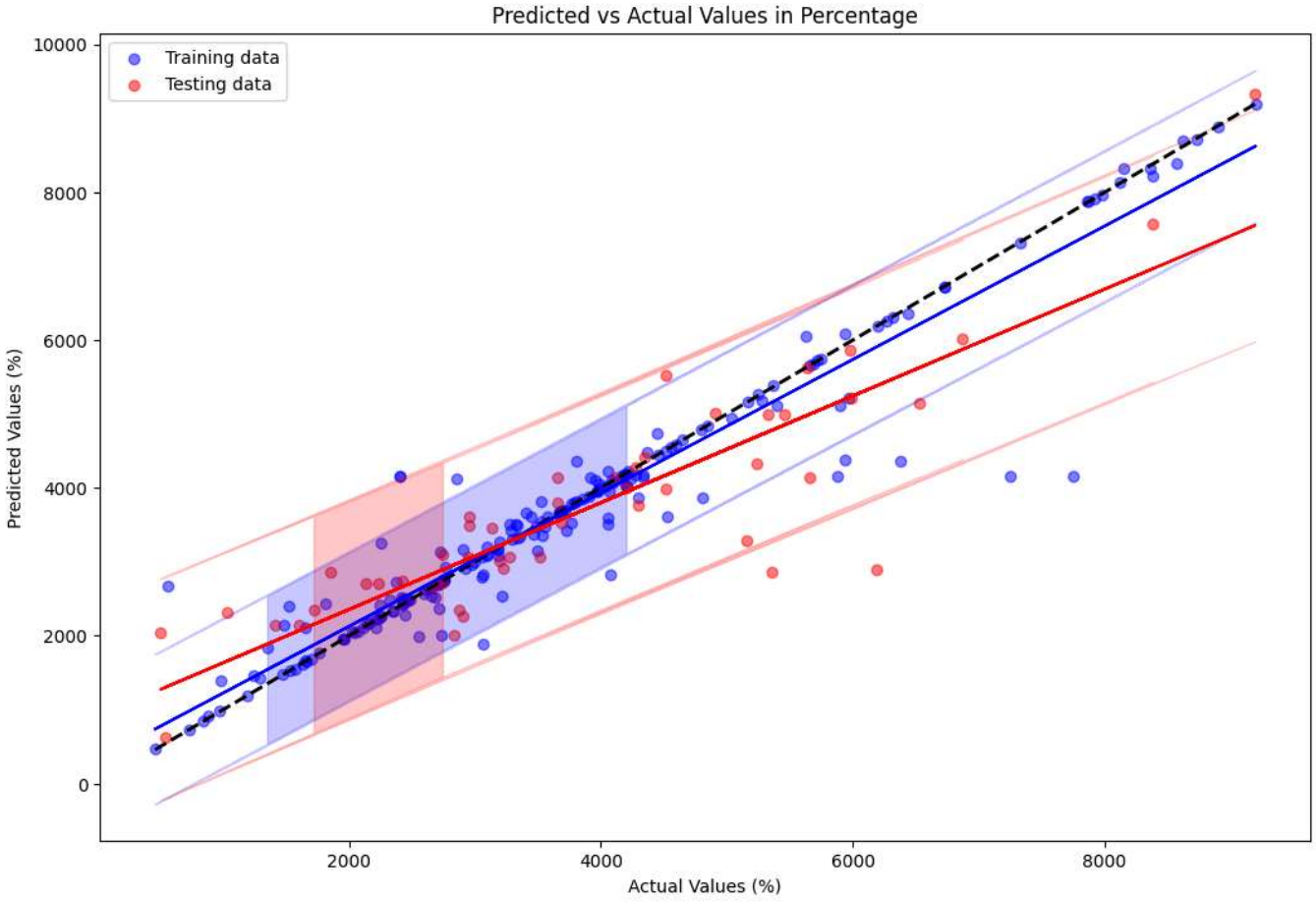


Metric	Training	Testing
R²	0.915	0.767
RMSE	5.404	8.872
RRMSE	0.145	0.229

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.svm import SVR
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error
from scipy.stats import t

# Assuming `df` is already defined as your DataFrame
# Replace the following line with the actual code to load your data if not already done
# df = pd.read_csv('your_data.csv')

feature_columns = ['Carbon content (wt%)',
                   'Hydrogen content (wt%)', 'Nitrogen content (wt%)',
                   'Oxygen content (wt%)', 'Sulfur content (wt%)', 'Volatile matter (wt%)',
                   'Fixed carbon (wt%)', 'Ash content (wt%)', 'Reaction temperature (°C)',
                   'Microwave power (W)', 'Reaction time (min)',
                   'Microwave absorber percentage (%)',
                   'Dielectric constant of absorber (ε′)',
                   'Dielectric loss factor of absorber (ε‟)']  # Replace with your actual feature column names
target_column = 'Syngas yield (%)'  # Replace with your actual target column name

# Assign the feature columns to X and the target column to y
X = df[feature_columns].values
y = df[target_column].values

# Split the dataset 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)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create and train the SVR model
svr = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=0.1)
svr.fit(X_train, y_train)

# Make predictions
y_train_pred = svr.predict(X_train)
y_test_pred = svr.predict(X_test)

# Calculate R²
r2_train = r2_score(y_train, y_train_pred)
r2_test = r2_score(y_test, y_test_pred)

# Calculate RMSE
rmse_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))

# Calculate RRMSE
rrmse_train = rmse_train / np.mean(y_train)
rrmse_test = rmse_test / np.mean(y_test)

# Create a DataFrame to compare the performance metrics
performance_df = pd.DataFrame({
    'Metric': ['R²', 'RMSE', 'RRMSE'],
    'Training': [r2_train, rmse_train, rrmse_train],
    'Testing': [r2_test, rmse_test, rrmse_test]
})

print(performance_df)

# Convert actual and predicted values to percentages for plotting
y_train_percentage = y_train * 100
y_train_pred_percentage = y_train_pred * 100
y_test_percentage = y_test * 100
y_test_pred_percentage = y_test_pred * 100

# Create scatter plot to compare predicted vs actual values in percentages
fig, ax = plt.subplots(figsize=(12, 8))
ax.scatter(y_train_percentage, y_train_pred_percentage, color='blue', alpha=0.5, label='Training data')
ax.scatter(y_test_percentage, y_test_pred_percentage, color='red', alpha=0.5, label='Testing data')
ax.plot([min(y) * 100, max(y) * 100], [min(y) * 100, max(y) * 100], 'k--', lw=2)
ax.set_xlabel('Actual Values (%)')
ax.set_ylabel('Predicted Values (%)')
ax.set_title('Predicted vs Actual Values in Percentage')
ax.legend()

# Add table with performance metrics
table_data = [['R²', f'{r2_train:.3f}', f'{r2_test:.3f}'],
```

```
                ['RMSE', f'{rmse_train:.3f}', f'{rmse_test:.3f}'],
                ['RRMSE', f'{rrmse_train:.3f}', f'{rrmse_test:.3f}']]
    table = ax.table(cellText=table_data,
                     colLabels=['Metric', 'Training', 'Testing'],
                     cellLoc='center',
                     loc='lower right',
                     bbox=[0.8, -0.3, 0.2, 0.2])  # Adjust bbox to control position and size

    # Function to calculate confidence intervals
    def confidence_interval(x, y, polyfit, alpha=0.05):
        p = np.poly1d(polyfit)
        yhat = p(x)
        n = len(y)
        residual = y - yhat
        std_error = (np.sum(residual**2) / (n - 2))**0.5
        t_val = t.ppf(1 - alpha/2, n - 2)
        ci = t_val * std_error * (1 + (x - np.mean(x))**2 / np.sum((x - np.mean(x))**2))**0.5
        return yhat - ci, yhat + ci

    # Training data trend line and confidence interval
    train_polyfit = np.polyfit(y_train_percentage, y_train_pred_percentage, 1)
    train_polyval = np.polyval(train_polyfit, y_train_percentage)
    train_ci_lower, train_ci_upper = confidence_interval(y_train_percentage, y_train_pred_percentage, train_polyfit)
    ax.plot(y_train_percentage, train_polyval, color='blue', label='Training trend line')
    ax.fill_between(y_train_percentage, train_ci_lower, train_ci_upper, color='blue', alpha=0.2)

    # Testing data trend line and confidence interval
    test_polyfit = np.polyfit(y_test_percentage, y_test_pred_percentage, 1)
    test_polyval = np.polyval(test_polyfit, y_test_percentage)
    test_ci_lower, test_ci_upper = confidence_interval(y_test_percentage, y_test_pred_percentage, test_polyfit)
    ax.plot(y_test_percentage, test_polyval, color='red', label='Testing trend line')
    ax.fill_between(y_test_percentage, test_ci_lower, test_ci_upper, color='red', alpha=0.2)

    # Adjust layout to make room for the table
    plt.subplots_adjust(left=0.1, right=0.9, top=0.9, bottom=0.1)

    # Reduce font size for the table
    table.auto_set_font_size(False)
    table.set_fontsize(10)
    table.scale(1, 1.5)  # Scale the table size

    plt.show()
```

```
   Metric  Training   Testing
0      R²  0.915195  0.767129
1    RMSE  5.404144  8.872308
2  RRMSE  0.144605  0.229255
```

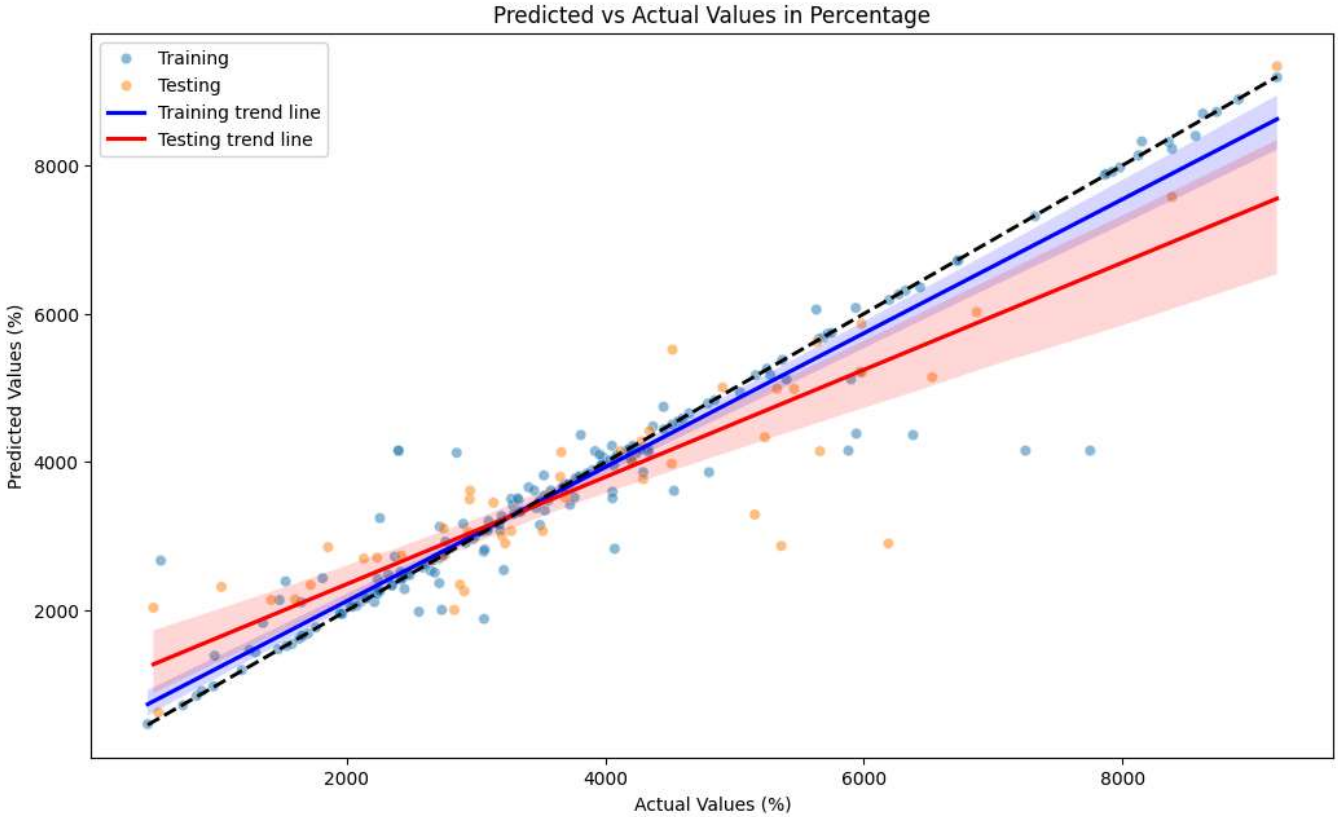


Metric	Training	Testing
R²	0.915	0.767
RMSE	5.404	8.872
RRMSE	0.145	0.229

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.svm import SVR
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error

# Assuming `df` is already defined as your DataFrame
# Replace the following line with the actual code to load your data if not already done
# df = pd.read_csv('your_data.csv')

feature_columns = ['Carbon content (wt%)',
                   'Hydrogen content (wt%)', 'Nitrogen content (wt%)',
                   'Oxygen content (wt%)', 'Sulfur content (wt%)', 'Volatile matter (wt%)',
                   'Fixed carbon (wt%)', 'Ash content (wt%)', 'Reaction temperature (°C)',
                   'Microwave power (W)', 'Reaction time (min)',
                   'Microwave absorber percentage (%)',
                   'Dielectric constant of absorber (ε’)',
                   'Dielectric loss factor of absorber (ε“)']  # Replace with your actual feature column names
target_column = 'Syngas yield (%)'  # Replace with your actual target column name

# Assign the feature columns to X and the target column to y
X = df[feature_columns].values
y = df[target_column].values

# Split the dataset 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)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create and train the SVR model
svr = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=0.1)
svr.fit(X_train, y_train)

# Make predictions
y_train_pred = svr.predict(X_train)
y_test_pred = svr.predict(X_test)

# Calculate R²
r2_train = r2_score(y_train, y_train_pred)
r2_test = r2_score(y_test, y_test_pred)

# Calculate RMSE
rmse_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))

# Calculate RRMSE
rrmse_train = rmse_train / np.mean(y_train)
rrmse_test = rmse_test / np.mean(y_test)

# Create a DataFrame to compare the performance metrics
performance_df = pd.DataFrame({
    'Metric': ['R²', 'RMSE', 'RRMSE'],
    'Training': [r2_train, rmse_train, rrmse_train],
    'Testing': [r2_test, rmse_test, rrmse_test]
})

print(performance_df)

# Convert actual and predicted values to percentages for plotting
y_train_percentage = y_train * 100
y_train_pred_percentage = y_train_pred * 100
y_test_percentage = y_test * 100
y_test_pred_percentage = y_test_pred * 100

# Combine data into a DataFrame for plotting with seaborn
df_plot = pd.DataFrame({
    'Actual': np.concatenate((y_train_percentage, y_test_percentage)),
    'Predicted': np.concatenate((y_train_pred_percentage, y_test_pred_percentage)),
    'Dataset': ['Training'] * len(y_train_percentage) + ['Testing'] * len(y_test_percentage)
})

# Create scatter plot with regression line
plt.figure(figsize=(12, 8))
sns.scatterplot(data=df_plot, x='Actual', y='Predicted', hue='Dataset', alpha=0.5)
sns.regplot(data=df_plot[df_plot['Dataset'] == 'Training'], x='Actual', y='Predicted', scatter=False, color='blue', label='Training tre
sns.regplot(data=df_plot[df_plot['Dataset'] == 'Testing'], x='Actual', y='Predicted', scatter=False, color='red', label='Testing trend
```

```
# Add diagonal line
plt.plot([min(y) * 100, max(y) * 100], [min(y) * 100, max(y) * 100], 'k--', lw=2)

# Add labels and title
plt.xlabel('Actual Values (%)')
plt.ylabel('Predicted Values (%)')
plt.title('Predicted vs Actual Values in Percentage')
plt.legend()

# Add table with performance metrics
table_data = [['R²', f'{r2_train:.3f}', f'{r2_test:.3f}'],
              ['RMSE', f'{rmse_train:.3f}', f'{rmse_test:.3f}'],
              ['RRMSE', f'{rrmse_train:.3f}', f'{rrmse_test:.3f}']]
table = plt.table(cellText=table_data,
                  colLabels=['Metric', 'Training', 'Testing'],
                  cellLoc='center',
                  loc='bottom',
                  bbox=[0.8, -0.3, 0.2, 0.2])  # Adjust bbox to control position and size

# Adjust layout to make room for the table
plt.subplots_adjust(left=0.1, right=0.9, top=0.9, bottom=0.2)

# Reduce font size for the table
table.auto_set_font_size(False)
table.set_fontsize(10)
table.scale(1, 1.5)  # Scale the table size

plt.show()
```

```
   Metric  Training   Testing
0      R²  0.915195  0.767129
1    RMSE  5.404144  8.872308
2   RRMSE  0.144605  0.229255
```



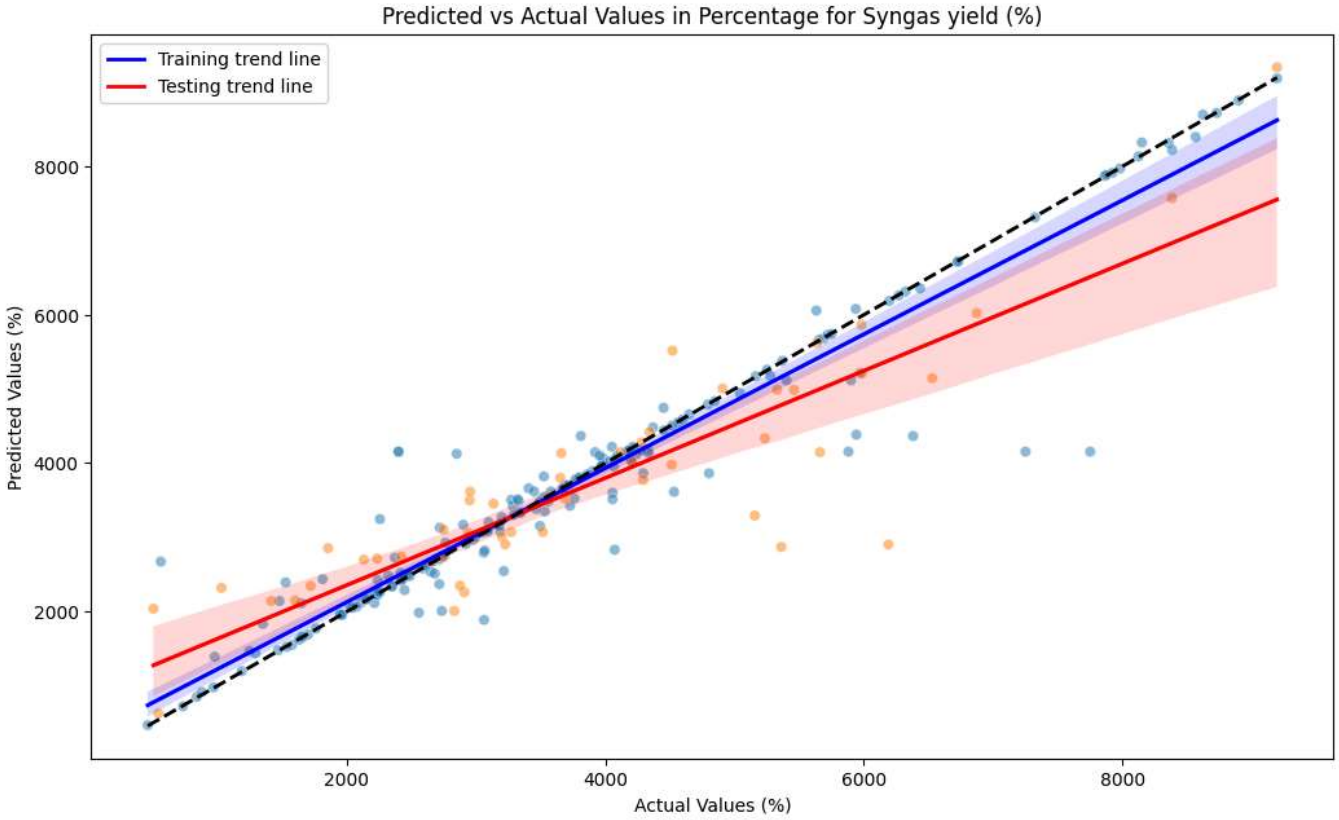
Metric	Training	Testing
R²	0.915	0.767
RMSE	5.404	8.872
RRMSE	0.145	0.229

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.svm import SVR
```

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error

# Assuming `df` is already defined as your DataFrame
# Replace the following line with the actual code to load your data if not already done
# df = pd.read_csv('your_data.csv')

feature_columns = ['Carbon content (wt%)',
                   'Hydrogen content (wt%)', 'Nitrogen content (wt%)',
                   'Oxygen content (wt%)', 'Sulfur content (wt%)', 'Volatile matter (wt%)',
                   'Fixed carbon (wt%)', 'Ash content (wt%)', 'Reaction temperature (°C)',
                   'Microwave power (W)', 'Reaction time (min)',
                   'Microwave absorber percentage (%)',
                   'Dielectric constant of absorber (ε′)',
                   'Dielectric loss factor of absorber (ε‟)']  # Replace with your actual feature column names
target_column = 'Syngas yield (%)'  # Replace with your actual target column name

# Assign the feature columns to X and the target column to y
X = df[feature_columns].values
y = df[target_column].values

# Split the dataset 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)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create and train the SVR model
svr = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=0.1)
svr.fit(X_train, y_train)

# Make predictions
y_train_pred = svr.predict(X_train)
y_test_pred = svr.predict(X_test)

# Calculate R²
r2_train = r2_score(y_train, y_train_pred)
r2_test = r2_score(y_test, y_test_pred)

# Calculate RMSE
rmse_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))

# Calculate RRMSE
rrmse_train = rmse_train / np.mean(y_train)
rrmse_test = rmse_test / np.mean(y_test)

# Create a DataFrame to compare the performance metrics
performance_df = pd.DataFrame({
    'Metric': ['R²', 'RMSE', 'RRMSE'],
    'Training': [r2_train, rmse_train, rrmse_train],
    'Testing': [r2_test, rmse_test, rrmse_test]
})

print(performance_df)

# Convert actual and predicted values to percentages for plotting
y_train_percentage = y_train * 100
y_train_pred_percentage = y_train_pred * 100
y_test_percentage = y_test * 100
y_test_pred_percentage = y_test_pred * 100

# Combine data into a DataFrame for plotting with seaborn
df_plot = pd.DataFrame({
    'Actual': np.concatenate((y_train_percentage, y_test_percentage)),
    'Predicted': np.concatenate((y_train_pred_percentage, y_test_pred_percentage)),
    'Dataset': ['Training'] * len(y_train_percentage) + ['Testing'] * len(y_test_percentage)
})

# Create scatter plot with regression line
plt.figure(figsize=(12, 8))
sns.scatterplot(data=df_plot, x='Actual', y='Predicted', hue='Dataset', alpha=0.5)

# Training data trend line and confidence interval
sns.regplot(data=df_plot[df_plot['Dataset'] == 'Training'], x='Actual', y='Predicted', scatter=False, color='blue', label='Training trend

# Testing data trend line and confidence interval
sns.regplot(data=df_plot[df_plot['Dataset'] == 'Testing'], x='Actual', y='Predicted', scatter=False, color='red', label='Testing trend li

# Add diagonal line
```

```
plt.plot([min(y) * 100, max(y) * 100], [min(y) * 100, max(y) * 100], 'k--', lw=2)

# Add labels and title with target column
plt.xlabel('Actual Values (%)')
plt.ylabel('Predicted Values (%)')
plt.title(f'Predicted vs Actual Values in Percentage for {target_column}')
plt.legend()

# Create custom legend
custom_lines = [plt.Line2D([0], [0], color='blue', lw=2),
                plt.Line2D([0], [0], color='red', lw=2)]
plt.legend(custom_lines, ['Training trend line', 'Testing trend line'])

# Add table with performance metrics
table_data = [['R²', f'{r2_train:.3f}', f'{r2_test:.3f}'],
              ['RMSE', f'{rmse_train:.3f}', f'{rmse_test:.3f}'],
              ['RRMSE', f'{rrmse_train:.3f}', f'{rrmse_test:.3f}']]
table = plt.table(cellText=table_data,
                  colLabels=['Metric', 'Training', 'Testing'],
                  cellLoc='center',
                  loc='bottom',
                  bbox=[0.8, -0.3, 0.2, 0.2])  # Adjust bbox to control position and size

# Adjust layout to make room for the table
plt.subplots_adjust(left=0.1, right=0.9, top=0.9, bottom=0.2)

# Reduce font size for the table
table.auto_set_font_size(False)
table.set_fontsize(10)
table.scale(1, 1.5)  # Scale the table size

plt.show()
```

```
   Metric  Training   Testing
0      R²  0.915195  0.767129
1    RMSE  5.404144  8.872308
2   RRMSE  0.144605  0.229255
```

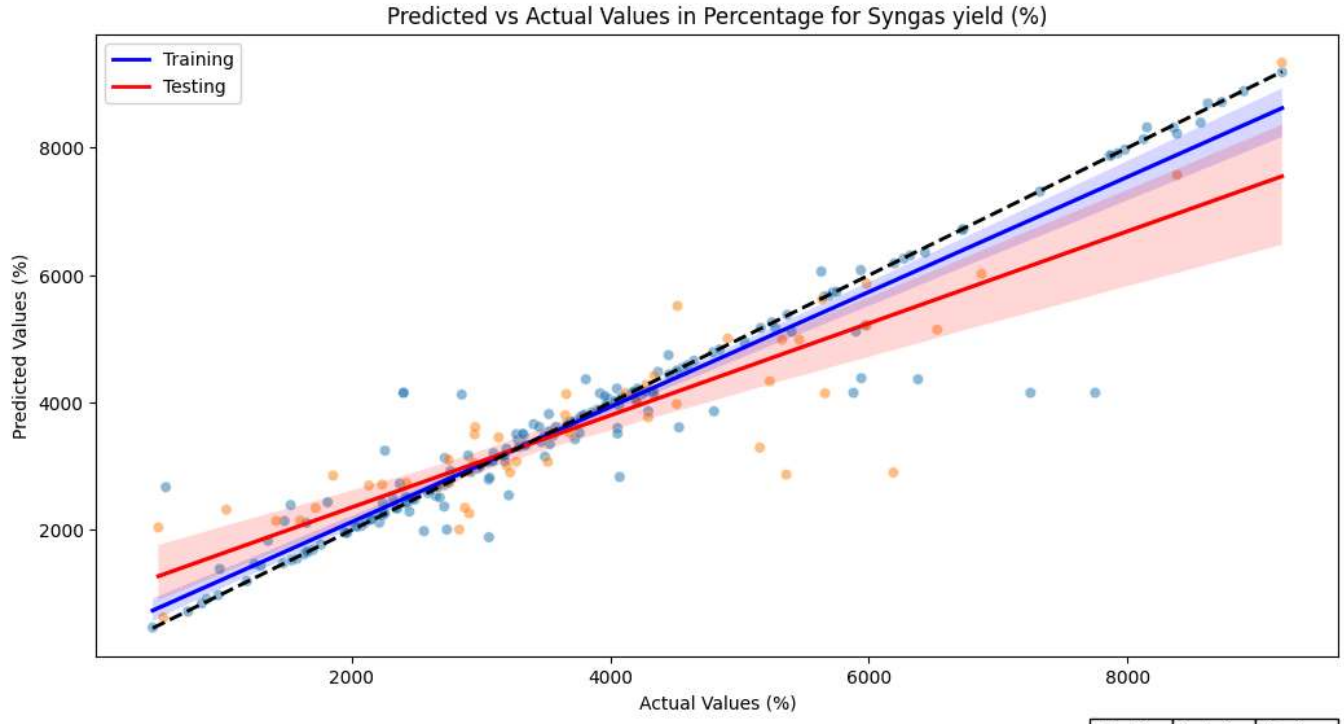


Metric	Training	Testing
R²	0.915	0.767
RMSE	5.404	8.872
RRMSE	0.145	0.229

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.svm import SVR
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error

# Assuming `df` is already defined as your DataFrame
# Replace the following line with the actual code to load your data if not already done
# df = pd.read_csv('your_data.csv')

feature_columns = ['Carbon content (wt%)',
                   'Hydrogen content (wt%)', 'Nitrogen content (wt%)',
                   'Oxygen content (wt%)', 'Sulfur content (wt%)', 'Volatile matter (wt%)',
                   'Fixed carbon (wt%)', 'Ash content (wt%)', 'Reaction temperature (°C)',
                   'Microwave power (W)', 'Reaction time (min)',
                   'Microwave absorber percentage (%)',
                   'Dielectric constant of absorber (ε’)',
                   'Dielectric loss factor of absorber (ε“)']  # Replace with your actual feature column names
target_column = 'Syngas yield (%)'  # Replace with your actual target column name

# Assign the feature columns to X and the target column to y
X = df[feature_columns].values
y = df[target_column].values

# Split the dataset 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)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create and train the SVR model
svr = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=0.1)
svr.fit(X_train, y_train)

# Make predictions
y_train_pred = svr.predict(X_train)
y_test_pred = svr.predict(X_test)

# Calculate R²
r2_train = r2_score(y_train, y_train_pred)
r2_test = r2_score(y_test, y_test_pred)

# Calculate RMSE
rmse_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))

# Calculate RRMSE
rrmse_train = rmse_train / np.mean(y_train)
rrmse_test = rmse_test / np.mean(y_test)

# Create a DataFrame to compare the performance metrics
performance_df = pd.DataFrame({
    'Metric': ['R²', 'RMSE', 'RRMSE'],
    'Training': [r2_train, rmse_train, rrmse_train],
    'Testing': [r2_test, rmse_test, rrmse_test]
})

print(performance_df)

# Convert actual and predicted values to percentages for plotting
y_train_percentage = y_train * 100
y_train_pred_percentage = y_train_pred * 100
y_test_percentage = y_test * 100
y_test_pred_percentage = y_test_pred * 100

# Combine data into a DataFrame for plotting with seaborn
df_plot = pd.DataFrame({
    'Actual': np.concatenate((y_train_percentage, y_test_percentage)),
    'Predicted': np.concatenate((y_train_pred_percentage, y_test_pred_percentage)),
    'Dataset': ['Training'] * len(y_train_percentage) + ['Testing'] * len(y_test_percentage)
})

# Create scatter plot with regression line
plt.figure(figsize=(12, 8))
sns.scatterplot(data=df_plot, x='Actual', y='Predicted', hue='Dataset', alpha=0.5)

# Training data trend line and confidence interval
```

```
sns.regplot(data=df_plot[df_plot['Dataset'] == 'Training'], x='Actual', y='Predicted', scatter=False, color='blue', label='Training tren

# Testing data trend line and confidence interval
sns.regplot(data=df_plot[df_plot['Dataset'] == 'Testing'], x='Actual', y='Predicted', scatter=False, color='red', label='Testing trend l

# Add diagonal line
plt.plot([min(y) * 100, max(y) * 100], [min(y) * 100, max(y) * 100], 'k--', lw=2)

# Add labels and title with target column
plt.xlabel('Actual Values (%)')
plt.ylabel('Predicted Values (%)')
plt.title(f'Predicted vs Actual Values in Percentage for {target_column}')

# Create custom legend with both training and testing labels
custom_lines = [plt.Line2D([0], [0], color='blue', lw=2),
                plt.Line2D([0], [0], color='red', lw=2)]
plt.legend(custom_lines, ['Training', 'Testing'])

# Add table with performance metrics
table_data = [['R²', f'{r2_train:.3f}', f'{r2_test:.3f}'],
              ['RMSE', f'{rmse_train:.3f}', f'{rmse_test:.3f}'],
              ['RRMSE', f'{rrmse_train:.3f}', f'{rrmse_test:.3f}']]
table = plt.table(cellText=table_data,
                  colLabels=['Metric', 'Training', 'Testing'],
                  cellLoc='center',
                  loc='bottom',
                  bbox=[0.8, -0.3, 0.2, 0.2])  # Adjust bbox to control position and size

# Adjust layout to make room for the table
plt.subplots_adjust(left=0.1, right=0.9, top=0.9, bottom=0.3)

# Reduce font size for the table
table.auto_set_font_size(False)
table.set_fontsize(10)
table.scale(1, 1.5)  # Scale the table size

plt.show()
```

```
   Metric  Training   Testing
0      R²  0.915195  0.767129
1    RMSE  5.404144  8.872308
2   RRMSE  0.144605  0.229255
```



Metric	Training	Testing
R²	0.915	0.767
RMSE	5.404	8.872
RRMSE	0.145	0.229

Now, put this into a function

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.svm import SVR
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error

def plot_predicted_vs_actual(df, feature_columns, target_column):
    # Assign the feature columns to X and the target column to y
    X = df[feature_columns].values
    y = df[target_column].values

    # Split the dataset 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)

    # Standardize the features
    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)

    # Create and train the SVR model
    svr = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=0.1)
    svr.fit(X_train, y_train)

    # Make predictions
    y_train_pred = svr.predict(X_train)
    y_test_pred = svr.predict(X_test)

    # Calculate R²
    r2_train = r2_score(y_train, y_train_pred)
    r2_test = r2_score(y_test, y_test_pred)

    # Calculate RMSE
    rmse_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
    rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))

    # Calculate RRMSE
    rrmse_train = rmse_train / np.mean(y_train)
    rrmse_test = rmse_test / np.mean(y_test)

    # Convert actual and predicted values to percentages for plotting
    y_train_percentage = y_train * 100
    y_train_pred_percentage = y_train_pred * 100
    y_test_percentage = y_test * 100
    y_test_pred_percentage = y_test_pred * 100

    # Combine data into a DataFrame for plotting with seaborn
    df_plot = pd.DataFrame({
        'Actual': np.concatenate((y_train_percentage, y_test_percentage)),
        'Predicted': np.concatenate((y_train_pred_percentage, y_test_pred_percentage)),
        'Dataset': ['Training'] * len(y_train_percentage) + ['Testing'] * len(y_test_percentage)
    })

    # Create scatter plot with regression line
    plt.figure(figsize=(12, 8))
    sns.scatterplot(data=df_plot, x='Actual', y='Predicted', hue='Dataset', alpha=0.5)

    # Training data trend line and confidence interval
    sns.regplot(data=df_plot[df_plot['Dataset'] == 'Training'], x='Actual', y='Predicted', scatter=False, color='blue', label='Training

    # Testing data trend line and confidence interval
    sns.regplot(data=df_plot[df_plot['Dataset'] == 'Testing'], x='Actual', y='Predicted', scatter=False, color='red', label='Testing tre

    # Add diagonal line
    plt.plot([min(y) * 100, max(y) * 100], [min(y) * 100, max(y) * 100], 'k--', lw=2)

    # Add labels and title with target column
    plt.xlabel('Actual Values (%)')
    plt.ylabel('Predicted Values (%)')
    plt.title(f'Predicted vs Actual Values in Percentage for {target_column}')

    # Create custom legend with both training and testing labels
    custom_lines = [plt.Line2D([0], [0], color='blue', lw=2),
                    plt.Line2D([0], [0], color='red', lw=2)]
    plt.legend(custom_lines, ['Training', 'Testing'])

    # Add table with performance metrics
    table_data = [['R²', f'{r2_train:.3f}', f'{r2_test:.3f}'],
                  ['RMSE', f'{rmse_train:.3f}', f'{rmse_test:.3f}'],
                  ['RRMSE', f'{rrmse_train:.3f}', f'{rrmse_test:.3f}']]
```

```
        table = plt.table(cellText=table_data,
                          colLabels=['Metric', 'Training', 'Testing'],
                          cellLoc='center',
                          loc='bottom',
                          bbox=[0.8, -0.3, 0.2, 0.2])  # Adjust bbox to control position and size

        # Adjust layout to make room for the table
        plt.subplots_adjust(left=0.1, right=0.9, top=0.9, bottom=0.3)

        # Reduce font size for the table
        table.auto_set_font_size(False)
        table.set_fontsize(10)
        table.scale(1, 1.5)  # Scale the table size

        plt.show()

    # Example usage:
    # Replace with your actual DataFrame and columns
    # df = pd.read_csv('your_data.csv')
    # feature_columns = ['feature1', 'feature2', ...]
    # target_column = 'target_column'
    # plot_predicted_vs_actual(df, feature_columns, target_column)
```

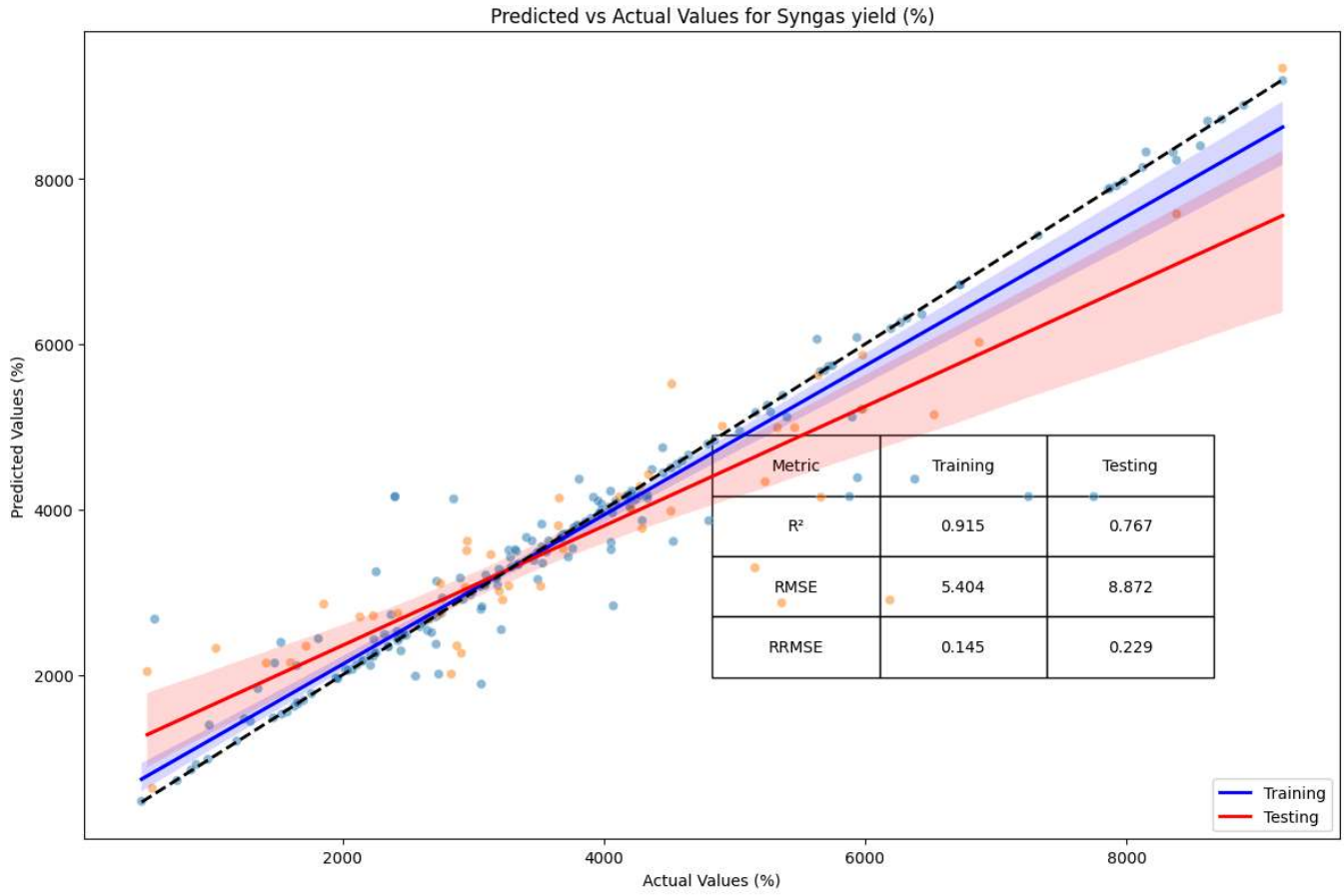
here to testify to print the graph together

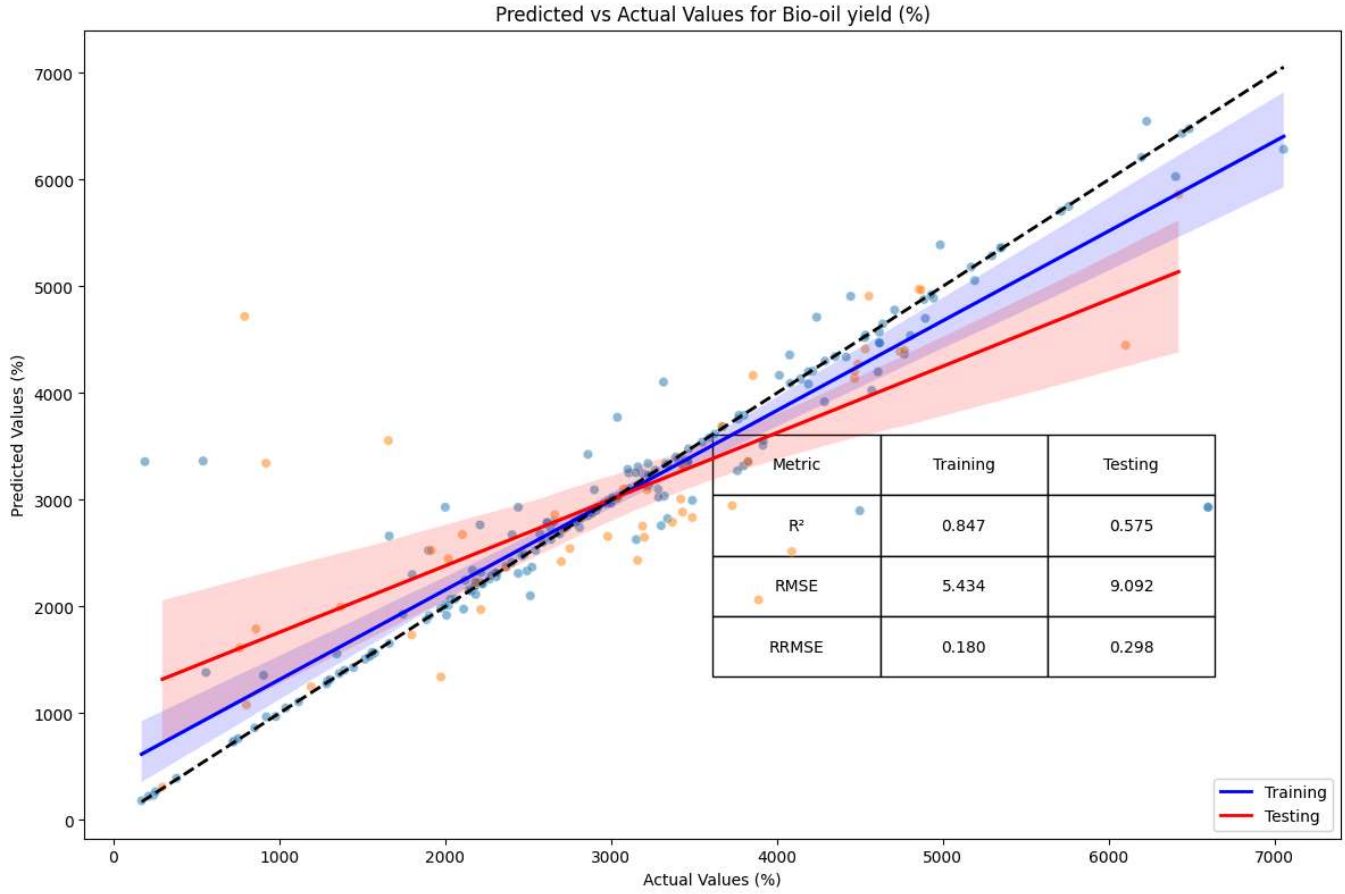
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.svm import SVR
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error

def plot_predicted_vs_actual_multi(df, feature_columns, target_columns):
    num_cols = len(target_columns)
    fig, axs = plt.subplots(num_cols, 1, figsize=(12, 8*num_cols))

    for idx, target_column in enumerate(target_columns):
        # Assign the feature columns to X and the target column to y
        X = df[feature_columns].values
        y = df[target_column].values

        # Split the dataset 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)

        # Standardize the features
        scaler = StandardScaler()
        X_train = scaler.fit_transform(X_train)
        X_test = scaler.transform(X_test)

        # Create and train the SVR model
        svr = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=0.1)
        svr.fit(X_train, y_train)

        # Make predictions
        y_train_pred = svr.predict(X_train)
        y_test_pred = svr.predict(X_test)

        # Calculate R²
        r2_train = r2_score(y_train, y_train_pred)
        r2_test = r2_score(y_test, y_test_pred)

        # Calculate RMSE
        rmse_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
        rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))

        # Calculate RRMSE
        rrmse_train = rmse_train / np.mean(y_train)
        rrmse_test = rmse_test / np.mean(y_test)

        # Convert actual and predicted values to percentages for plotting
        y_train_percentage = y_train * 100
        y_train_pred_percentage = y_train_pred * 100
        y_test_percentage = y_test * 100
        y_test_pred_percentage = y_test_pred * 100

        # Combine data into a DataFrame for plotting with seaborn
        df_plot = pd.DataFrame({
            'Actual': np.concatenate((y_train_percentage, y_test_percentage))
```

```
                'Actual': np.concatenate((y_train_percentage, y_test_percentage)),
                'Predicted': np.concatenate((y_train_pred_percentage, y_test_pred_percentage)),
                'Dataset': ['Training'] * len(y_train_percentage) + ['Testing'] * len(y_test_percentage)
            })

            # Plot on the respective subplot
            ax = axs[idx]
            sns.scatterplot(data=df_plot, x='Actual', y='Predicted', hue='Dataset', alpha=0.5, ax=ax)

            # Training data trend line and confidence interval
            sns.regplot(data=df_plot[df_plot['Dataset'] == 'Training'], x='Actual', y='Predicted', scatter=False, color='blue', ax=ax)

            # Testing data trend line and confidence interval
            sns.regplot(data=df_plot[df_plot['Dataset'] == 'Testing'], x='Actual', y='Predicted', scatter=False, color='red', ax=ax)

            # Add diagonal line
            ax.plot([min(y) * 100, max(y) * 100], [min(y) * 100, max(y) * 100], 'k--', lw=2)

            # Add labels and title with target column
            ax.set_xlabel('Actual Values (%)')
            ax.set_ylabel('Predicted Values (%)')
            ax.set_title(f'Predicted vs Actual Values for {target_column}')

            # Create custom legend with both training and testing labels
            custom_lines = [plt.Line2D([0], [0], color='blue', lw=2),
                            plt.Line2D([0], [0], color='red', lw=2)]
            ax.legend(custom_lines, ['Training', 'Testing'])

            # Add table with performance metrics
            table_data = [['R²', f'{r2_train:.3f}', f'{r2_test:.3f}'],
                          ['RMSE', f'{rmse_train:.3f}', f'{rmse_test:.3f}'],
                          ['RRMSE', f'{rrmse_train:.3f}', f'{rrmse_test:.3f}']]
            table = ax.table(cellText=table_data,
                             colLabels=['Metric', 'Training', 'Testing'],
                             cellLoc='center',
                             loc='bottom',
                             bbox=[0.8, -0.3, 0.2, 0.2])  # Adjust bbox to control position and size

            # Adjust layout to make room for the table
            fig.subplots_adjust(left=0.1, right=0.9, top=0.9, bottom=0.3)

            # Reduce font size for the table
            table.auto_set_font_size(False)
            table.set_fontsize(10)
            table.scale(1, 1.5)  # Scale the table size

        plt.tight_layout()
        plt.show()

    # Example usage:
    # Replace with your actual DataFrame and columns
    # df = pd.read_csv('your_data.csv')
    # feature_columns = ['feature1', 'feature2', ...]
    # target_columns = ['target_column1', 'target_column2', ...]
    # plot_predicted_vs_actual_multi(df, feature_columns, target_columns)
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.svm import SVR
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error

def plot_predicted_vs_actual_multi(df, feature_columns, target_columns):
    num_cols = len(target_columns)
    fig, axs = plt.subplots(num_cols, 1, figsize=(12, 8*num_cols))

    for idx, target_column in enumerate(target_columns):
        # Assign the feature columns to X and the target column to y
        X = df[feature_columns].values
        y = df[target_column].values

        # Split the dataset 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)

        # Standardize the features
        scaler = StandardScaler()
        X_train = scaler.fit_transform(X_train)
        X_test = scaler.transform(X_test)

        # Create and train the SVR model
        svr = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=0.1)
        svr.fit(X_train, y_train)

        # Make predictions
        y_train_pred = svr.predict(X_train)
        y_test_pred = svr.predict(X_test)

        # Calculate R²
        r2_train = r2_score(y_train, y_train_pred)
        r2_test = r2_score(y_test, y_test_pred)

        # Calculate RMSE
        rmse_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
        rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))

        # Calculate RRMSE
        rrmse_train = rmse_train / np.mean(y_train)
        rrmse_test = rmse_test / np.mean(y_test)

        # Convert actual and predicted values to percentages for plotting
        y_train_percentage = y_train * 100
        y_train_pred_percentage = y_train_pred * 100
        y_test_percentage = y_test * 100
        y_test_pred_percentage = y_test_pred * 100

        # Combine data into a DataFrame for plotting with seaborn
        df_plot = pd.DataFrame({
            'Actual': np.concatenate((y_train_percentage, y_test_percentage)),
            'Predicted': np.concatenate((y_train_pred_percentage, y_test_pred_percentage)),
            'Dataset': ['Training'] * len(y_train_percentage) + ['Testing'] * len(y_test_percentage)
        })

        # Plot on the respective subplot
        ax = axs[idx]
        sns.scatterplot(data=df_plot, x='Actual', y='Predicted', hue='Dataset', alpha=0.5, ax=ax)

        # Training data trend line and confidence interval
        sns.regplot(data=df_plot[df_plot['Dataset'] == 'Training'], x='Actual', y='Predicted', scatter=False, color='blue', ax=ax)

        # Testing data trend line and confidence interval
        sns.regplot(data=df_plot[df_plot['Dataset'] == 'Testing'], x='Actual', y='Predicted', scatter=False, color='red', ax=ax)

        # Add diagonal line
        ax.plot([min(y) * 100, max(y) * 100], [min(y) * 100, max(y) * 100], 'k--', lw=2)

        # Add labels and title with target column
        ax.set_xlabel('Actual Values (%)')
        ax.set_ylabel('Predicted Values (%)')
        ax.set_title(f'Predicted vs Actual Values for {target_column}')

        # Create custom legend with both training and testing labels
        custom_lines = [plt.Line2D([0], [0], color='blue', lw=2),
                        plt.Line2D([0], [0], color='red', lw=2)]
        ax.legend(custom_lines, ['Training', 'Testing'], loc='lower right')
```

```
            # Add table with performance metrics
            table_data = [['R²', f'{r2_train:.3f}', f'{r2_test:.3f}'],
                          ['RMSE', f'{rmse_train:.3f}', f'{rmse_test:.3f}'],
                          ['RRMSE', f'{rrmse_train:.3f}', f'{rrmse_test:.3f}']]
            table = ax.table(cellText=table_data,
                             colLabels=['Metric', 'Training', 'Testing'],
                             cellLoc='center',
                             loc='center right',  # Adjusted to center right
                             bbox=[0.5, 0.2, 0.4, 0.3])  # Adjust bbox to control position and size

            # Adjust table font size and style
            table.auto_set_font_size(False)
            table.set_fontsize(10)

        plt.tight_layout()
        plt.show()

# Example usage:
# Assuming `df`, `feature_columns`, and `target_columns` are defined appropriately
plot_predicted_vs_actual_multi(df, feature_columns, target_columns)
```


Predicted vs Actual Values for Syngas yield (%)
Predicted Values (%)
Actual Values (%)
Metric | Training | Testing
R² | 0.915 | 0.767
RMSE | 5.404 | 8.872
RRMSE | 0.145 | 0.229
Training
Testing


Predicted vs Actual Values for Bio-oil yield (%)
Predicted Values (%)
Actual Values (%)
Metric | Training | Testing
R² | 0.847 | 0.575
RMSE | 5.434 | 9.092
RRMSE | 0.180 | 0.298
Training
Testing

Here is the final result, as a function

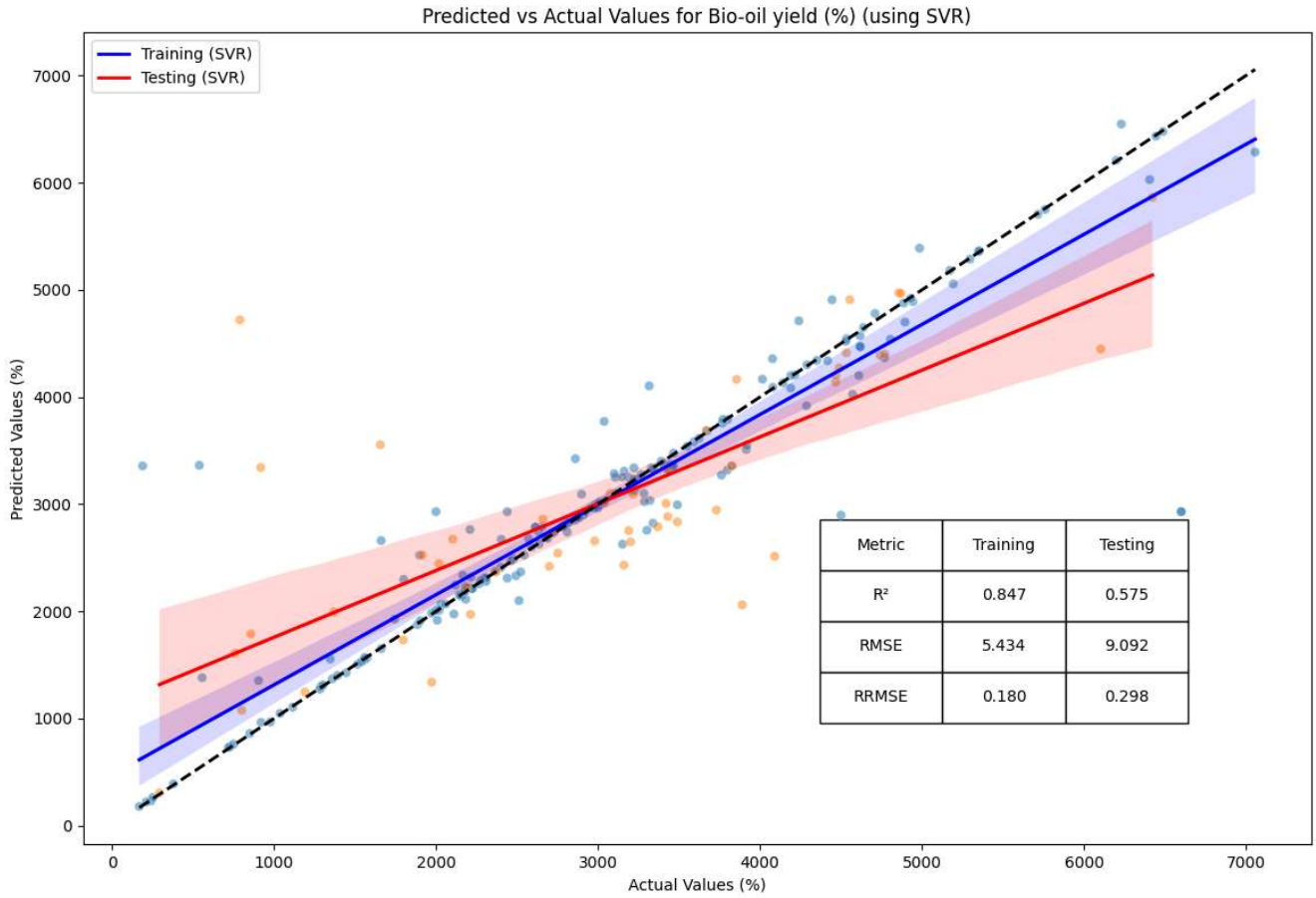
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.svm import SVR
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error

def plot_predicted_vs_actual_multi(df, feature_columns, target_columns):
    num_cols = len(target_columns)
    fig, axs = plt.subplots(num_cols, 1, figsize=(12, 8*num_cols))

    for idx, target_column in enumerate(target_columns):
        # Assign the feature columns to X and the target column to y
        X = df[feature_columns].values
        y = df[target_column].values

        # Split the dataset 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)

        # Standardize the features
        scaler = StandardScaler()
        X_train = scaler.fit_transform(X_train)
        X_test = scaler.transform(X_test)

        # Create and train the SVR model
        svr = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=0.1)
        svr.fit(X_train, y_train)

        # Make predictions
        y_train_pred = svr.predict(X_train)
        y_test_pred = svr.predict(X_test)

        # Calculate R²
        r2_train = r2_score(y_train, y_train_pred)
        r2_test = r2_score(y_test, y_test_pred)

        # Calculate RMSE
        rmse_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
        rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))

        # Calculate RRMSE
        rrmse_train = rmse_train / np.mean(y_train)
        rrmse_test = rmse_test / np.mean(y_test)

        # Convert actual and predicted values to percentages for plotting
        y_train_percentage = y_train * 100
        y_train_pred_percentage = y_train_pred * 100
        y_test_percentage = y_test * 100
        y_test_pred_percentage = y_test_pred * 100

        # Combine data into a DataFrame for plotting with seaborn
        df_plot = pd.DataFrame({
            'Actual': np.concatenate((y_train_percentage, y_test_percentage)),
            'Predicted': np.concatenate((y_train_pred_percentage, y_test_pred_percentage)),
            'Dataset': ['Training'] * len(y_train_percentage) + ['Testing'] * len(y_test_percentage)
        })

        # Plot on the respective subplot
        ax = axs[idx]
        sns.scatterplot(data=df_plot, x='Actual', y='Predicted', hue='Dataset', alpha=0.5, ax=ax)

        # Training data trend line and confidence interval
        sns.regplot(data=df_plot[df_plot['Dataset'] == 'Training'], x='Actual', y='Predicted', scatter=False, color='blue', ax=ax)

        # Testing data trend line and confidence interval
        sns.regplot(data=df_plot[df_plot['Dataset'] == 'Testing'], x='Actual', y='Predicted', scatter=False, color='red', ax=ax)

        # Add diagonal line
        ax.plot([min(y) * 100, max(y) * 100], [min(y) * 100, max(y) * 100], 'k--', lw=2)

        # Add labels and title with target column
        ax.set_xlabel('Actual Values (%)')
        ax.set_ylabel('Predicted Values (%)')
        ax.set_title(f'Predicted vs Actual Values for {target_column}')

        # Create custom legend with both training and testing labels
        custom_lines = [plt.Line2D([0], [0], color='blue', lw=2),
                        plt.Line2D([0], [0], color='red', lw=2)]
        ax.legend(custom_lines, ['Training', 'Testing'], loc='upper left')
```

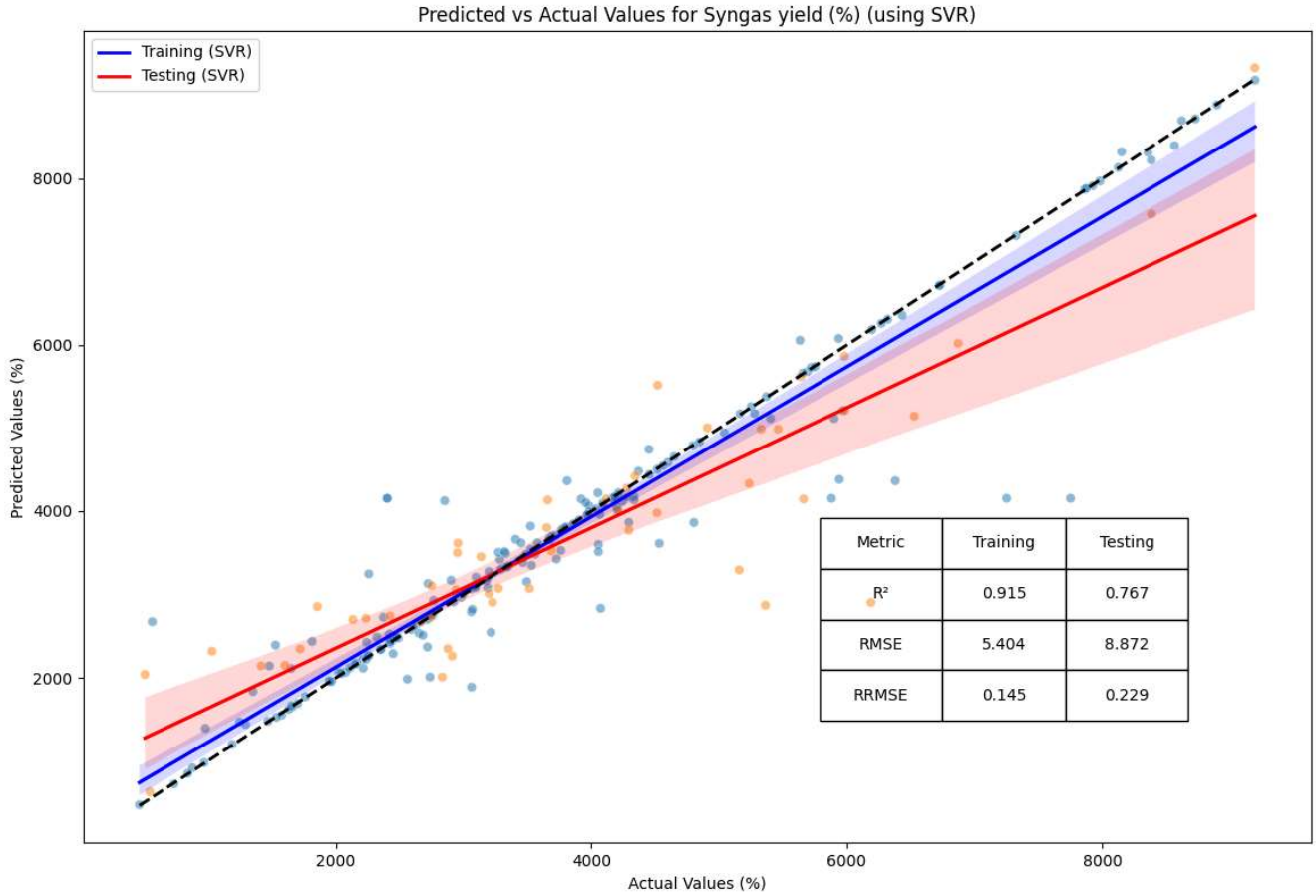
```
import numpy as np
import pandas as pd
from sklearn.svm import SVR
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error


df = pd.read_excel('/content/cleaned_imputed_data.xlsx')

df.head()
print(df.columns)

feature_columns = [
    'Carbon content (wt%)',
    'Hydrogen content (wt%)',
    'Nitrogen content (wt%)',
    'Oxygen content (wt%)',
    'Sulfur content (wt%)',
    'Volatile matter (wt%)',
    'Fixed carbon (wt%)',
    'Ash content (wt%)',
    'Reaction temperature (°C)',
    'Microwave power (W)',
    'Reaction time (min)',
    'Microwave absorber percentage (%)',
    'Dielectric constant of absorber (ε’)',
    'Dielectric loss factor of absorber (ε“)'
]
target_columns = ['Bio-oil yield (%)',
       'Syngas yield (%)', 'Syngas composition (H₂, mol%)',
       'Syngas composition (CH₄, mol%)', 'Syngas composition (CO₂, mol%)',
       'Syngas composition (CO, mol%)', 'Biochar yield (%)',
       'Biochar calorific value (MJ/kg)', 'Biochar H/C ratio (-)',
       'Biochar H/N ratio (-)', 'Biochar O/C ratio (-)']
```

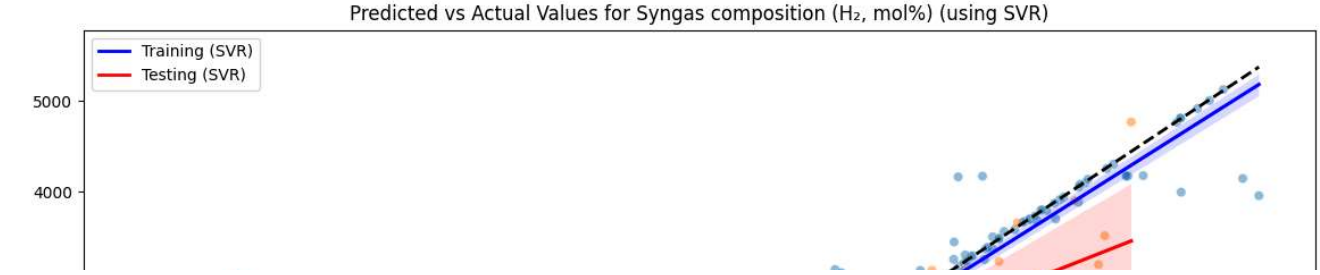
```
Index(['Reference (DOI)', 'Biomass type', 'Carbon content (wt%)',
       'Hydrogen content (wt%)', 'Nitrogen content (wt%)',
       'Oxygen content (wt%)', 'Sulfur content (wt%)', 'Volatile matter (wt%)',
       'Fixed carbon (wt%)', 'Ash content (wt%)', 'Reaction temperature (°C)',
       'Microwave power (W)', 'Reaction time (min)',
       'Microwave absorber percentage (%)',
       'Dielectric constant of absorber (ε’)',
       'Dielectric loss factor of absorber (ε“)', 'Bio-oil yield (%)',
       'Syngas yield (%)', 'Syngas composition (H₂, mol%)',
       'Syngas composition (CH₄, mol%)', 'Syngas composition (CO₂, mol%)',
       'Syngas composition (CO, mol%)', 'Biochar yield (%)',
       'Biochar calorific value (MJ/kg)', 'Biochar H/C ratio (-)',
       'Biochar H/N ratio (-)', 'Biochar O/C ratio (-)'],
      dtype='object')
```

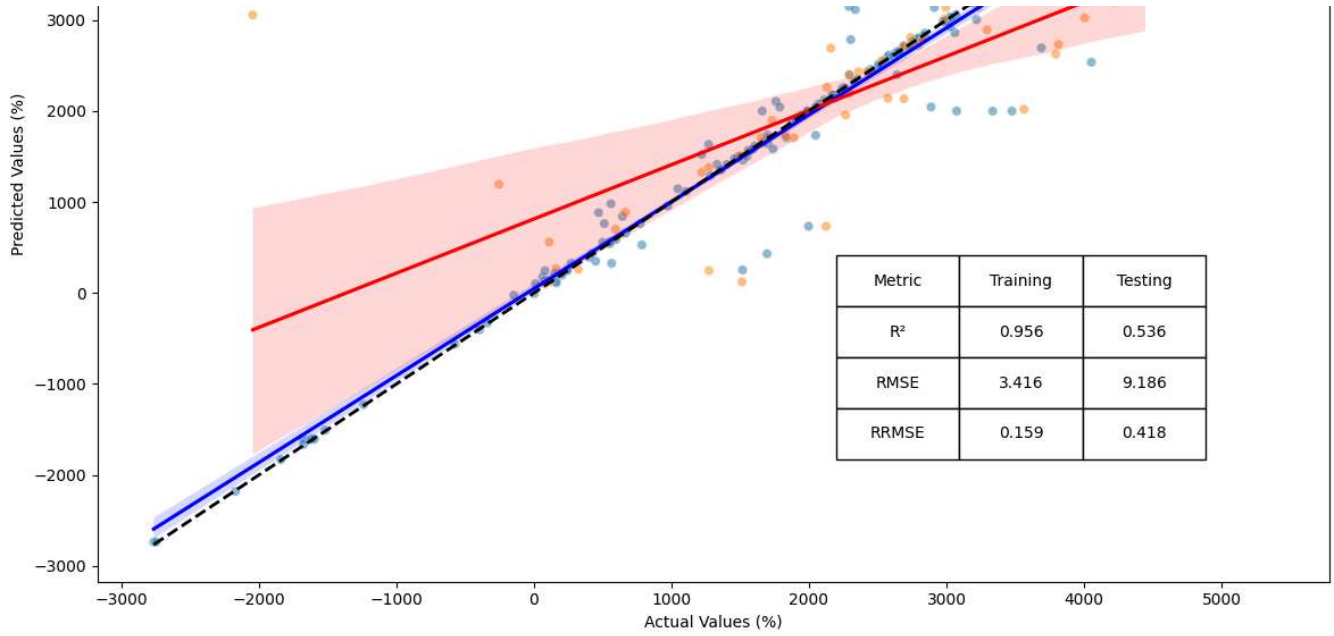
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.svm import SVR
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error

def plot_predicted_vs_actual_multi(df, feature_columns, target_columns, output_file=None):
    num_cols = len(target_columns)
    fig, axs = plt.subplots(num_cols, 1, figsize=(12, 8*num_cols))

    for idx, target_column in enumerate(target_columns):
        # Assign the feature columns to X and the target column to y
        X = df[feature_columns].values
        y = df[target_column].values

        # Split the dataset 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)

        # Standardize the features
        scaler = StandardScaler()
        X_train = scaler.fit_transform(X_train)
        X_test = scaler.transform(X_test)

        # Create and train the SVR model
        svr = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=0.1)
        svr.fit(X_train, y_train)

        # Make predictions
        y_train_pred = svr.predict(X_train)
        y_test_pred = svr.predict(X_test)

        # Calculate R²
        r2_train = r2_score(y_train, y_train_pred)
        r2_test = r2_score(y_test, y_test_pred)

        # Calculate RMSE
        rmse_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
        rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))

        # Calculate RRMSE
        rrmse_train = rmse_train / np.mean(y_train)
        rrmse_test = rmse_test / np.mean(y_test)

        # Convert actual and predicted values to percentages for plotting
        y_train_percentage = y_train * 100
        y_train_pred_percentage = y_train_pred * 100
        y_test_percentage = y_test * 100
        y_test_pred_percentage = y_test_pred * 100

        # Combine data into a DataFrame for plotting with seaborn
        df_plot = pd.DataFrame({
            'Actual': np.concatenate((y_train_percentage, y_test_percentage)),
            'Predicted': np.concatenate((y_train_pred_percentage, y_test_pred_percentage)),
            'Dataset': ['Training'] * len(y_train_percentage) + ['Testing'] * len(y_test_percentage)
        })

        # Plot on the respective subplot
        ax = axs[idx]
        sns.scatterplot(data=df_plot, x='Actual', y='Predicted', hue='Dataset', alpha=0.5, ax=ax)

        # Training data trend line and confidence interval
        sns.regplot(data=df_plot[df_plot['Dataset'] == 'Training'], x='Actual', y='Predicted', scatter=False, color='blue', ax=ax)

        # Testing data trend line and confidence interval
        sns.regplot(data=df_plot[df_plot['Dataset'] == 'Testing'], x='Actual', y='Predicted', scatter=False, color='red', ax=ax)

        # Add diagonal line
        ax.plot([min(y) * 100, max(y) * 100], [min(y) * 100, max(y) * 100], 'k--', lw=2)

        # Add labels and title with target column
        ax.set_xlabel('Actual Values (%)')
        ax.set_ylabel('Predicted Values (%)')
        ax.set_title(f'Predicted vs Actual Values for {target_column} (using SVR)')

        # Create custom legend with both training and testing labels
        custom_lines = [plt.Line2D([0], [0], color='blue', lw=2),
                        plt.Line2D([0], [0], color='red', lw=2)]
        ax.legend(custom_lines, ['Training (SVR)', 'Testing (SVR)'], loc='upper left')
```

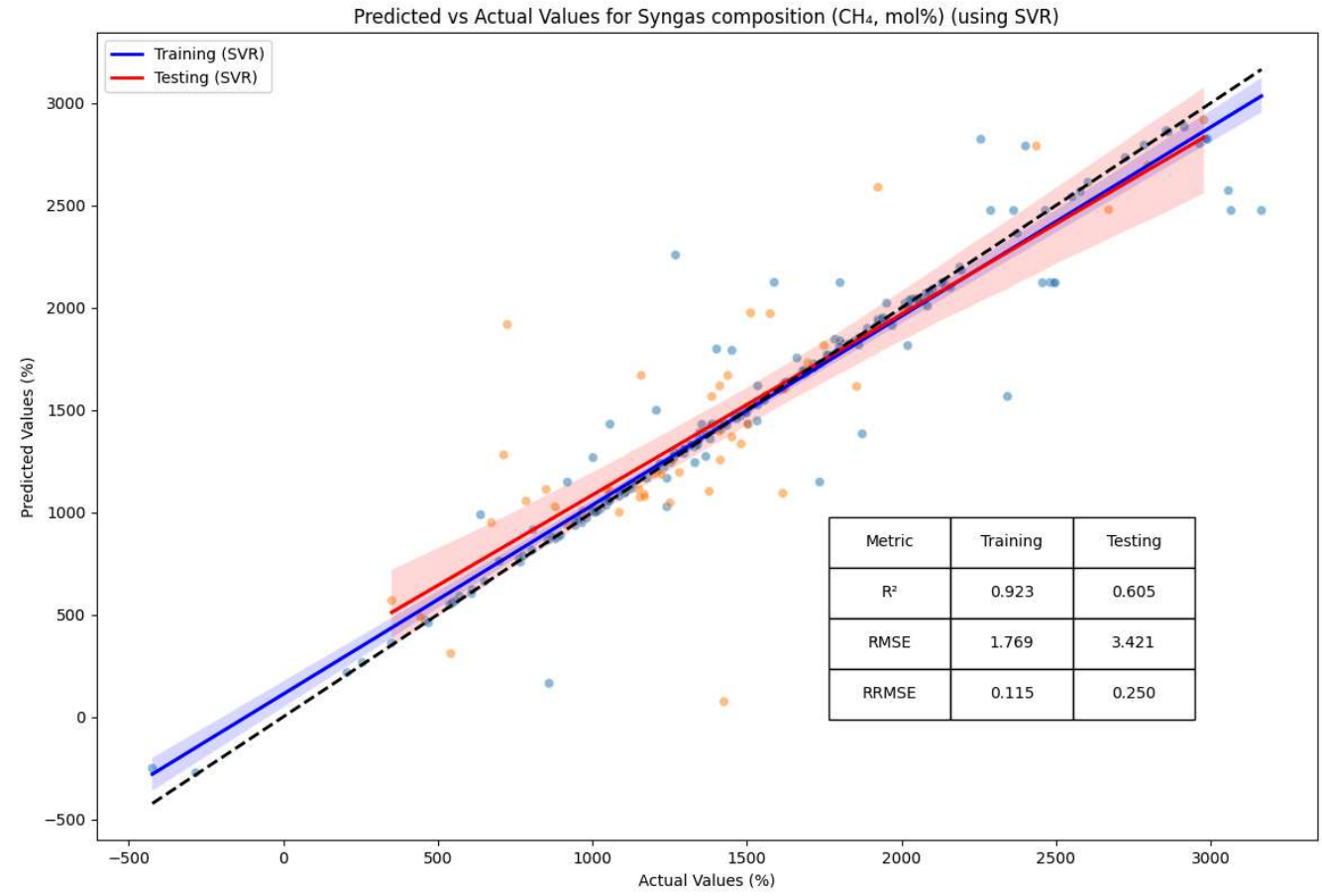
```
            # Add table with performance metrics
            table_data = [['R²', f'{r2_train:.3f}', f'{r2_test:.3f}'],
                          ['RMSE', f'{rmse_train:.3f}', f'{rmse_test:.3f}'],
                          ['RRMSE', f'{rrmse_train:.3f}', f'{rrmse_test:.3f}']]
            table = ax.table(cellText=table_data,
                             colLabels=['Metric', 'Training', 'Testing'],
                             cellLoc='center',
                             loc='bottom right',  # Adjusted to bottom right
                             bbox=[0.6, 0.15, 0.3, 0.25])  # Adjust bbox to control position and size

            # Adjust table font size and style
            table.auto_set_font_size(False)
            table.set_fontsize(10)

        plt.tight_layout()

        # Save the figure to a file if output_file is provided
        if output_file:
            plt.savefig(output_file)

        plt.show()

    # Example usage:
    # Assuming `df`, `feature_columns`, and `target_columns` are defined appropriately
    # Also assuming `output_file` is the file path where you want to save the plot
    feature_columns = [
        'Carbon content (wt%)',
        'Hydrogen content (wt%)',
        'Nitrogen content (wt%)',
        'Oxygen content (wt%)',
        'Sulfur content (wt%)',
        'Volatile matter (wt%)',
        'Fixed carbon (wt%)',
        'Ash content (wt%)',
        'Reaction temperature (°C)',
        'Microwave power (W)',
        'Reaction time (min)',
        'Microwave absorber percentage (%)',
        'Dielectric constant of absorber (ε’)',
        'Dielectric loss factor of absorber (ε“)'
    ]

    target_columns = [
        'Bio-oil yield (%)',
        'Syngas yield (%)',
        'Syngas composition (H₂, mol%)',
        'Syngas composition (CH₄, mol%)',
        'Syngas composition (CO₂, mol%)',
        'Syngas composition (CO, mol%)',
        'Biochar yield (%)',
        'Biochar calorific value (MJ/kg)',
        'Biochar H/C ratio (-)',
        'Biochar H/N ratio (-)',
        'Biochar O/C ratio (-)'
    ]

    output_file = 'SVR_predicted_vs_actual_multi.png'
    plot_predicted_vs_actual_multi(df, feature_columns, target_columns, output_file)
```

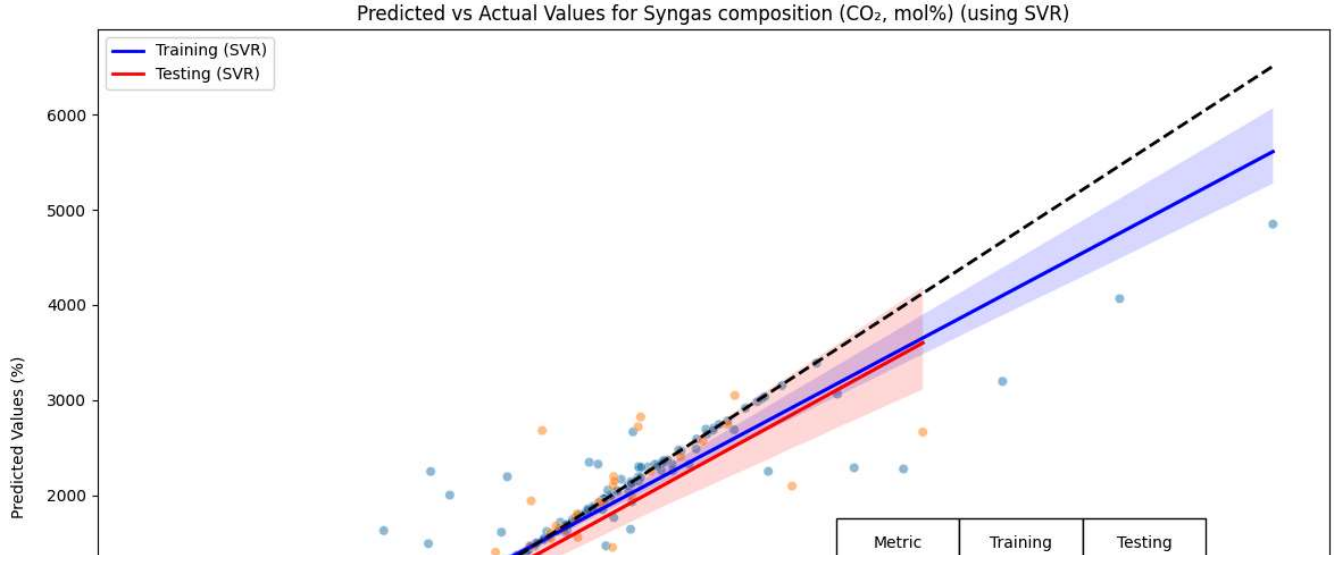

Predicted vs Actual Values for Bio-oil yield (%) (using SVR)
Training (SVR)
Testing (SVR)
Predicted Values (%)
Actual Values (%)
0
1000
2000
3000
4000
5000
6000
7000
Metric | Training | Testing
R² | 0.847 | 0.575
RMSE | 5.434 | 9.092
RRMSE | 0.180 | 0.298

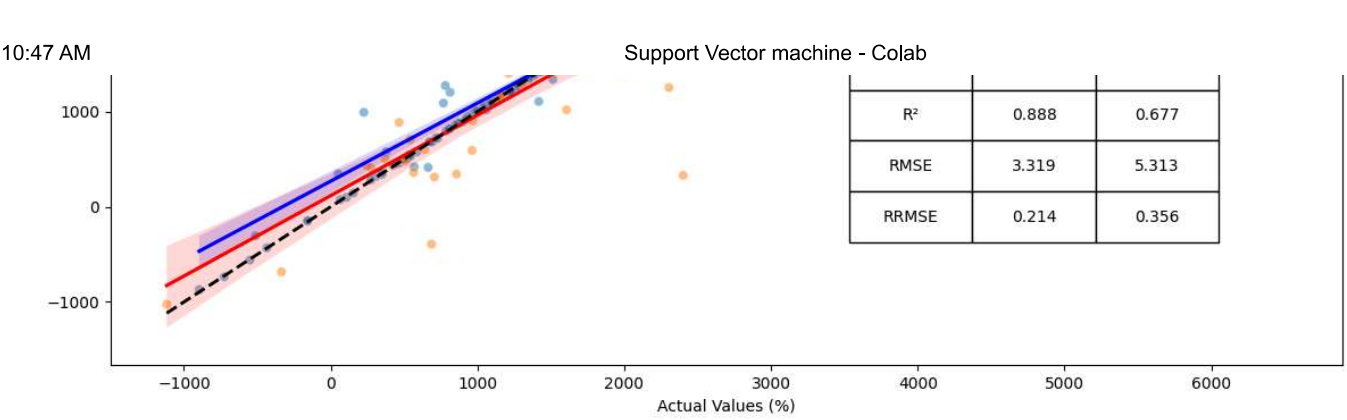

Predicted vs Actual Values for Syngas yield (%) (using SVR)
Training (SVR)
Testing (SVR)
Predicted Values (%)
Actual Values (%)
2000
4000
6000
8000
Metric | Training | Testing
R² | 0.915 | 0.767
RMSE | 5.404 | 8.872
RRMSE | 0.145 | 0.229


Predicted vs Actual Values for Syngas composition (H₂, mol%) (using SVR)
Training (SVR)
Testing (SVR)
5000
4000

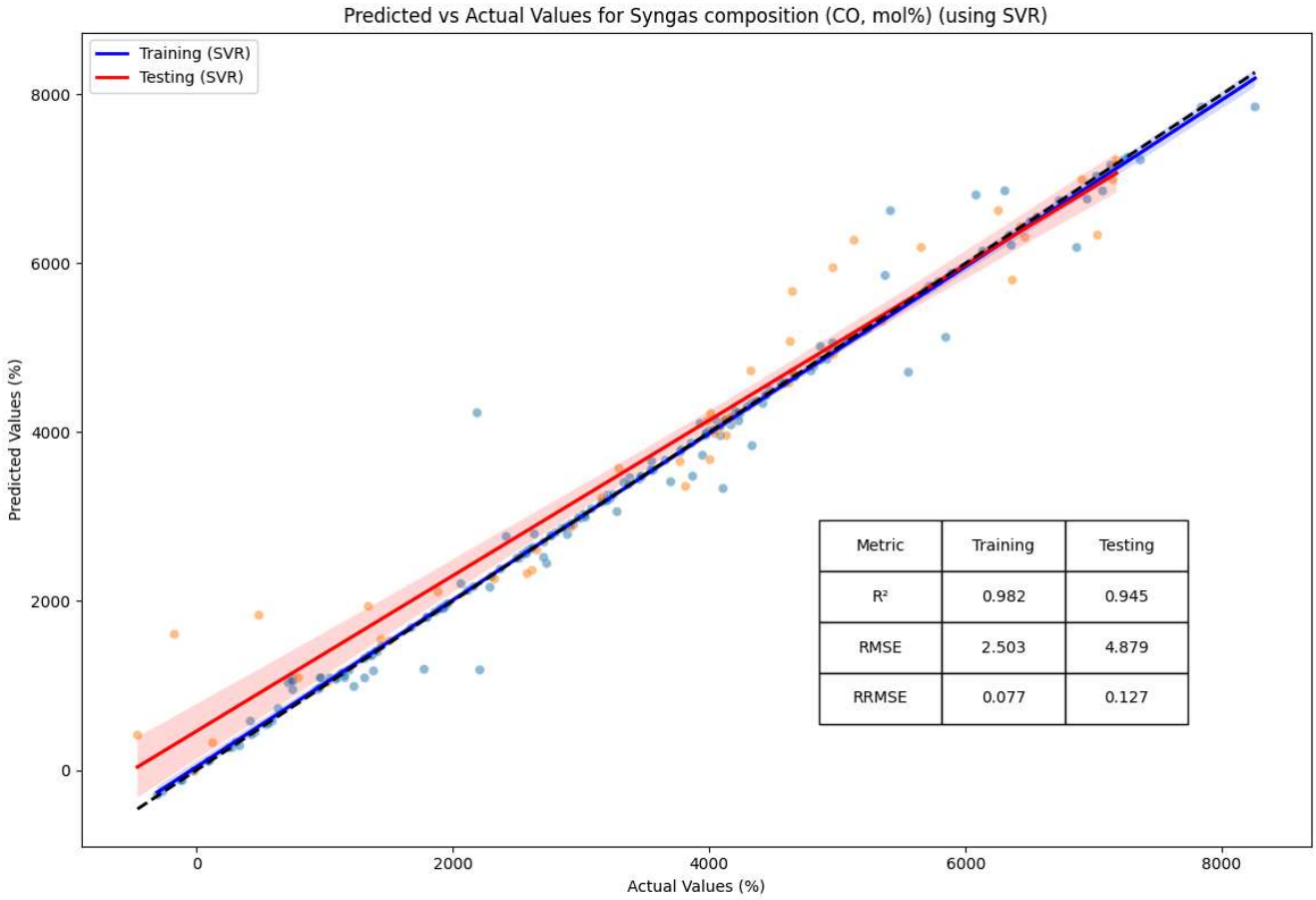

Predicted Values (%)
Actual Values (%)
Metric | Training | Testing
R² | 0.956 | 0.536
RMSE | 3.416 | 9.186
RRMSE | 0.159 | 0.418


Predicted vs Actual Values for Syngas composition (CH₄, mol%) (using SVR)
Training (SVR)
Testing (SVR)
Predicted Values (%)
Actual Values (%)
Metric | Training | Testing
R² | 0.923 | 0.605
RMSE | 1.769 | 3.421
RRMSE | 0.115 | 0.250

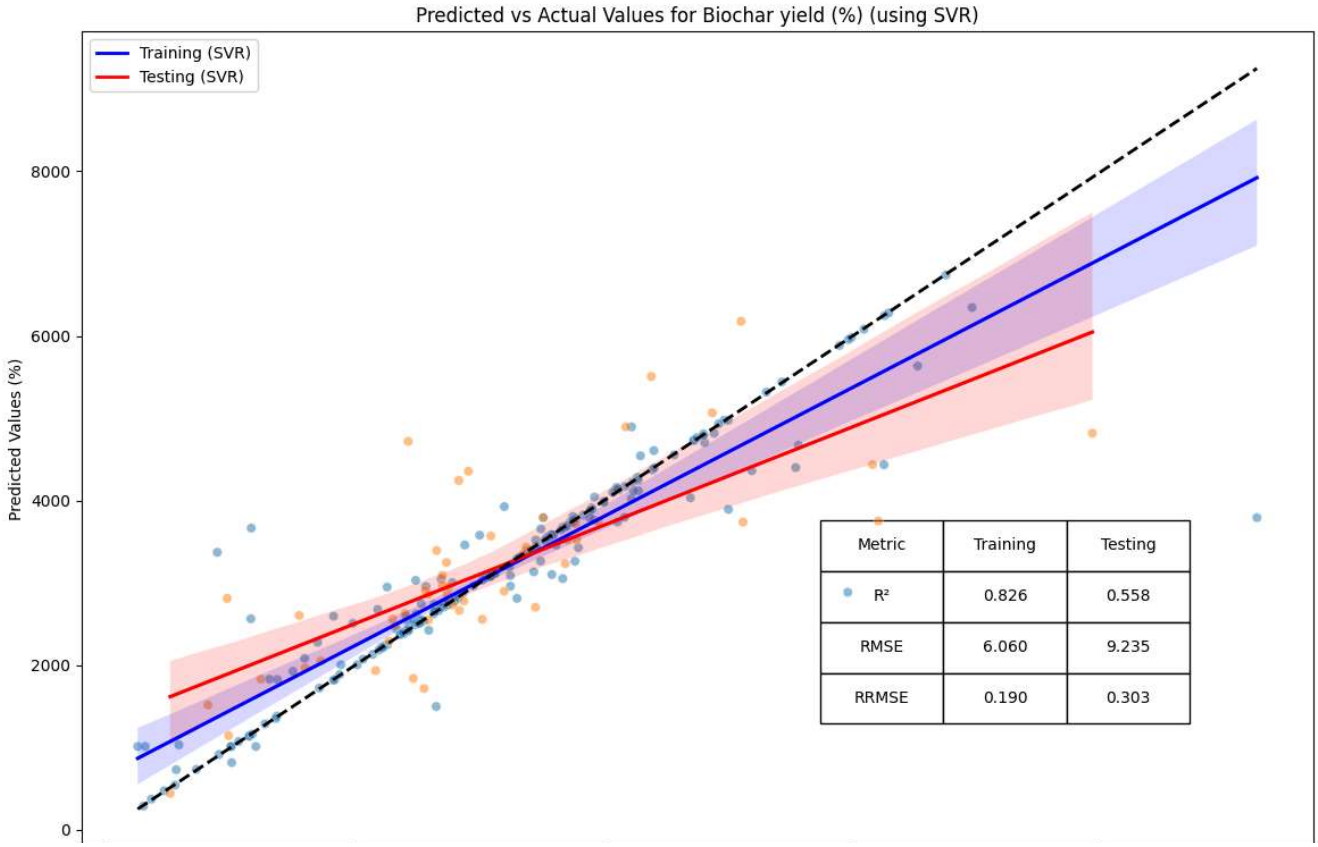

Predicted vs Actual Values for Syngas composition (CO₂, mol%) (using SVR)
Training (SVR)
Testing (SVR)
Predicted Values (%)
Metric | Training | Testing



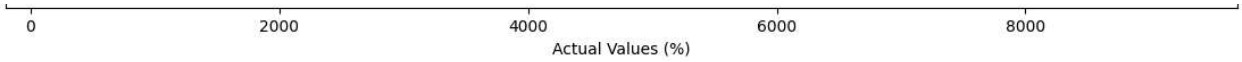
R²	0.888	0.677
RMSE	3.319	5.313
RRMSE	0.214	0.356

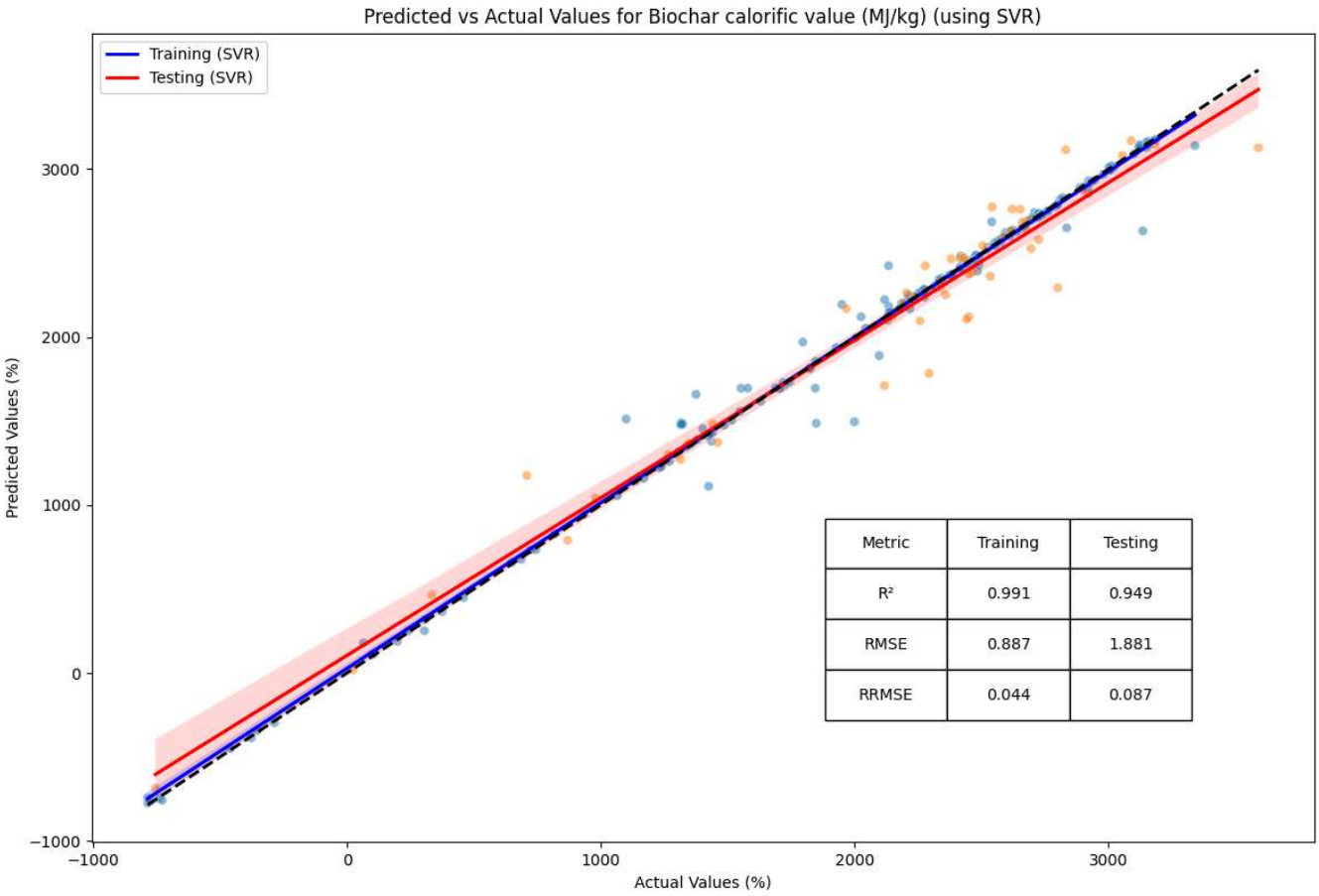


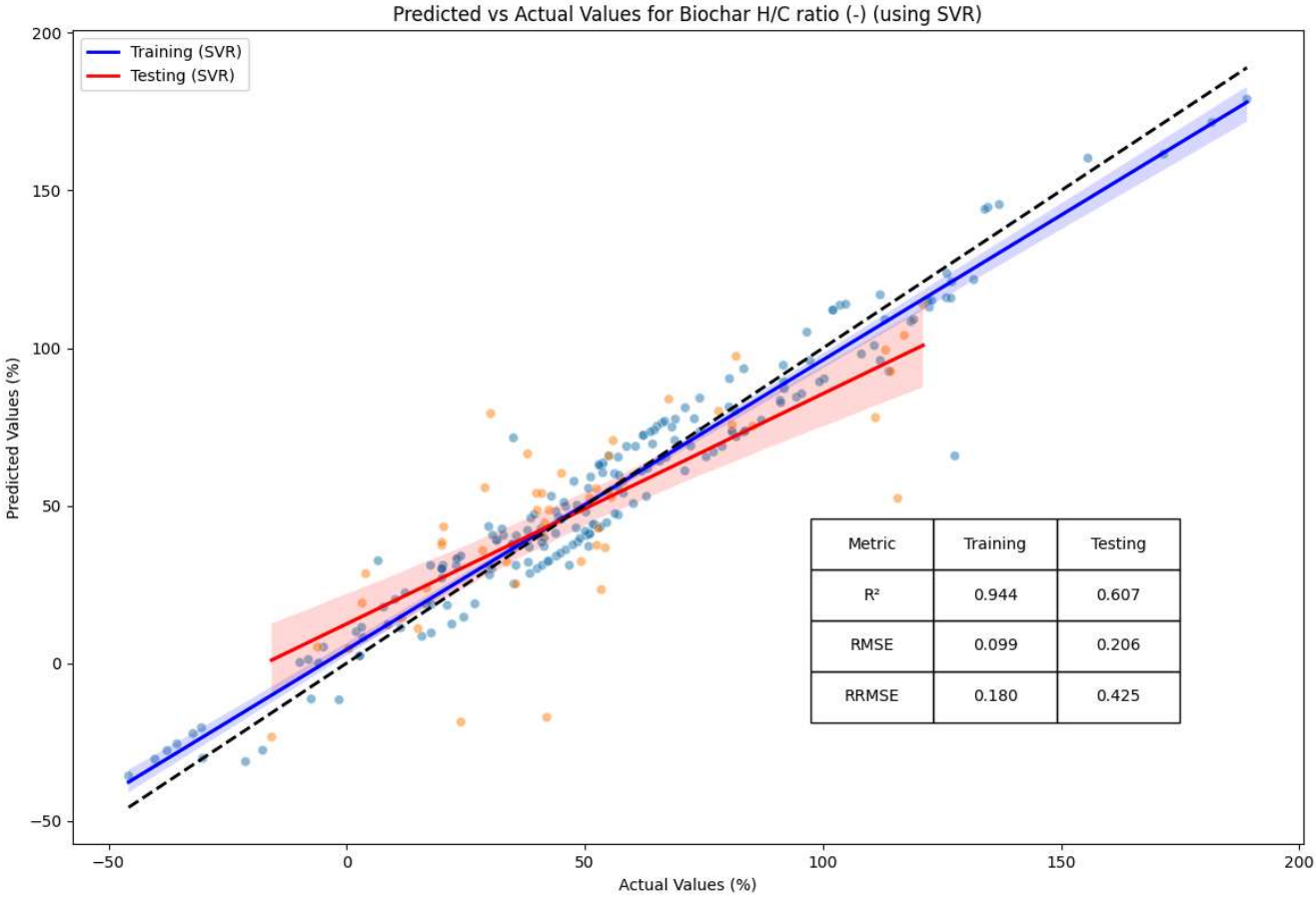
Metric	Training	Testing
R²	0.982	0.945
RMSE	2.503	4.879
RRMSE	0.077	0.127

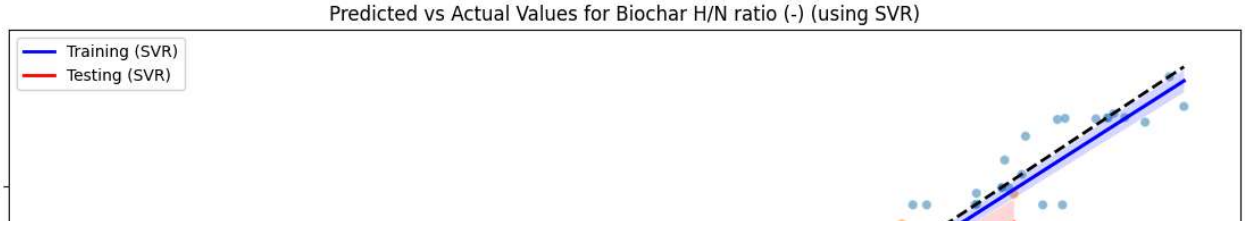


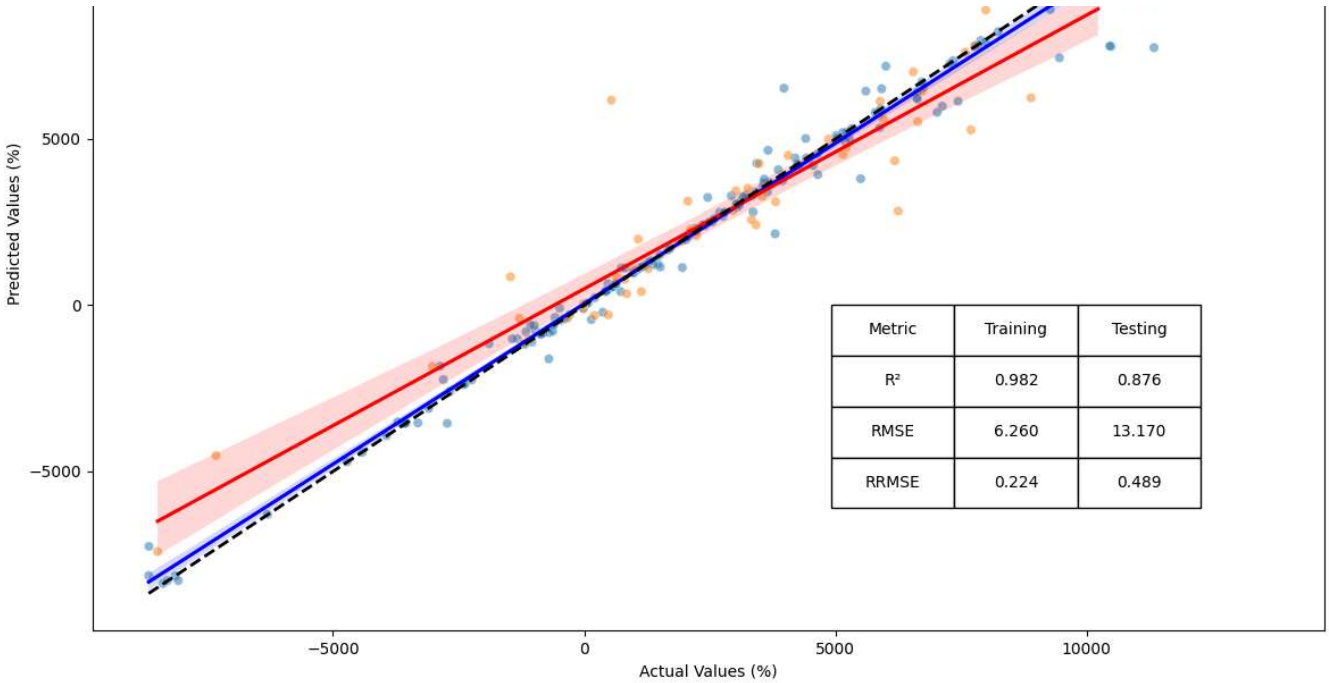
Metric	Training	Testing
R²	0.826	0.558
RMSE	6.060	9.235
RRMSE	0.190	0.303


0
2000
4000
6000
8000
Actual Values (%)


Predicted vs Actual Values for Biochar calorific value (MJ/kg) (using SVR)
Training (SVR)
Testing (SVR)
Predicted Values (%)
Actual Values (%)
3000
2000
1000
0
−1000
−1000
0
1000
2000
3000
Metric | Training | Testing
R² | 0.991 | 0.949
RMSE | 0.887 | 1.881
RRMSE | 0.044 | 0.087

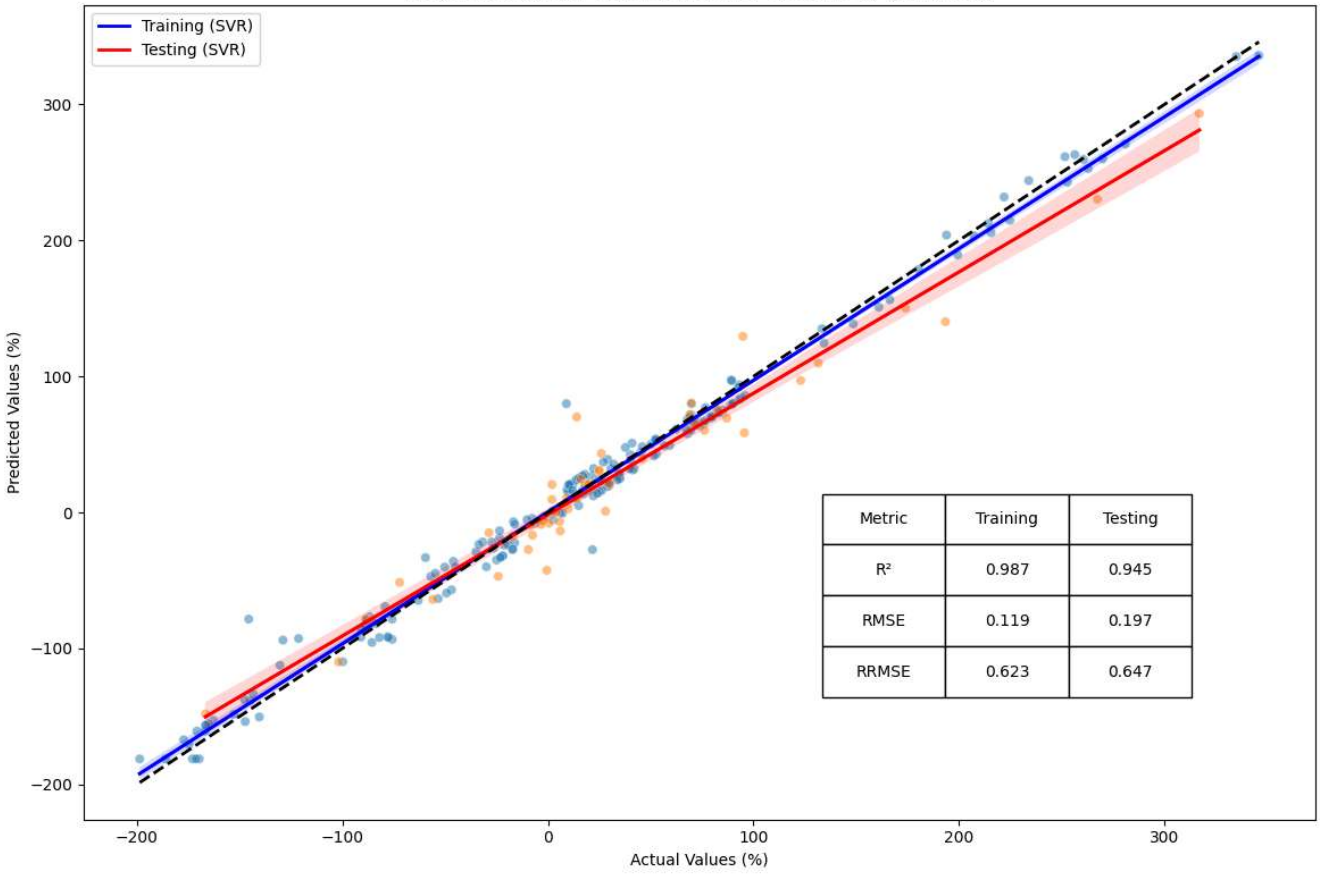

Predicted vs Actual Values for Biochar H/C ratio (-) (using SVR)
Training (SVR)
Testing (SVR)
Predicted Values (%)
Actual Values (%)
200
150
100
50
0
−50
−50
0
50
100
150
200
Metric | Training | Testing
R² | 0.944 | 0.607
RMSE | 0.099 | 0.206
RRMSE | 0.180 | 0.425


Predicted vs Actual Values for Biochar H/N ratio (-) (using SVR)
Training (SVR)
Testing (SVR)
10000



Metric	Training	Testing
R^2	0.982	0.876
RMSE	6.260	13.170
RRMSE	0.224	0.489

Predicted vs Actual Values for Biochar O/C ratio (-) (using SVR)



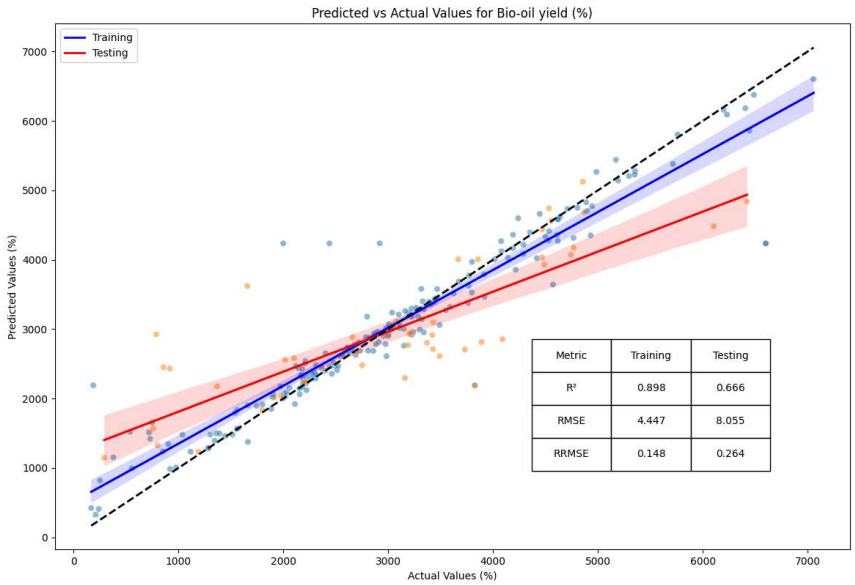
Metric	Training	Testing
R^2	0.987	0.945
RMSE	0.119	0.197
RRMSE	0.623	0.647

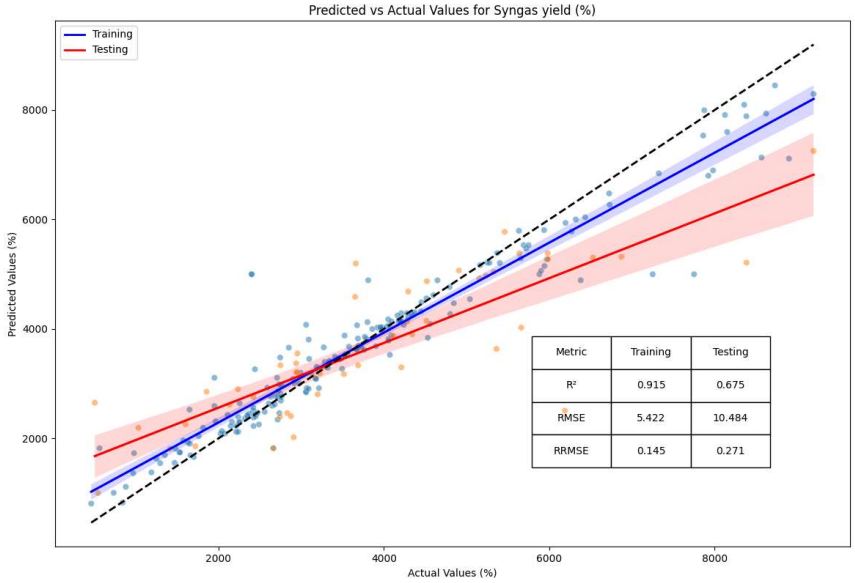
```
import numpy as np
import pandas as pd
from sklearn.svm import SVR
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error


df = pd.read_excel('/content/cleaned_imputed_data.xlsx')

df.head()
print(df.columns)

feature_columns = [
    'Carbon content (wt%)',
    'Hydrogen content (wt%)',
    'Nitrogen content (wt%)',
    'Oxygen content (wt%)',
    'Sulfur content (wt%)',
    'Volatile matter (wt%)',
    'Fixed carbon (wt%)',
    'Ash content (wt%)',
    'Reaction temperature (°C)',
    'Microwave power (W)',
    'Reaction time (min)',
    'Microwave absorber percentage (%)',
    'Dielectric constant of absorber (ε’)',
    'Dielectric loss factor of absorber (ε“)'
]
target_columns = ['Bio-oil yield (%)',
       'Syngas yield (%)', 'Syngas composition (H₂, mol%)',
       'Syngas composition (CH₄, mol%)', 'Syngas composition (CO₂, mol%)',
       'Syngas composition (CO, mol%)', 'Biochar yield (%)',
       'Biochar calorific value (MJ/kg)', 'Biochar H/C ratio (-)',
       'Biochar H/N ratio (-)', 'Biochar O/C ratio (-)']
```

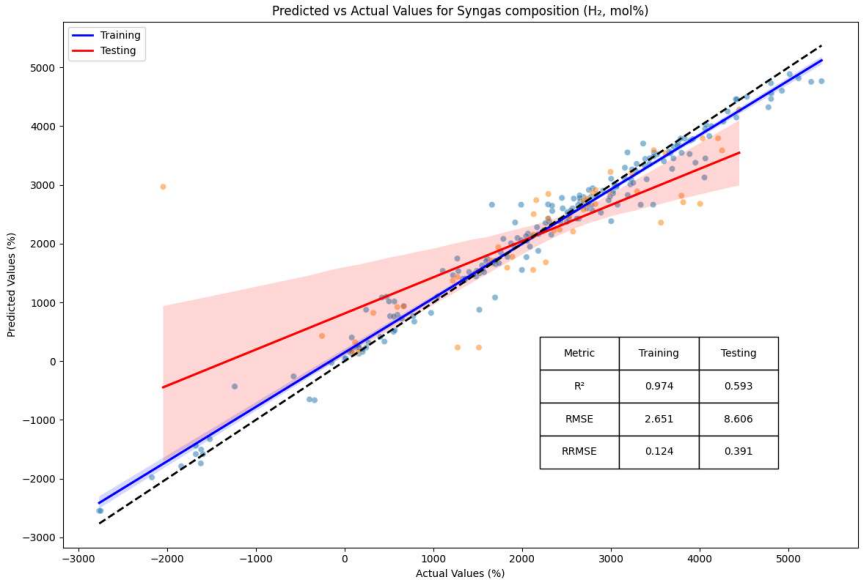
```
Index(['Reference (DOI)', 'Biomass type', 'Carbon content (wt%)',
       'Hydrogen content (wt%)', 'Nitrogen content (wt%)',
       'Oxygen content (wt%)', 'Sulfur content (wt%)', 'Volatile matter (wt%)',
       'Fixed carbon (wt%)', 'Ash content (wt%)', 'Reaction temperature (°C)',
       'Microwave power (W)', 'Reaction time (min)',
       'Microwave absorber percentage (%)',
       'Dielectric constant of absorber (ε’)',
       'Dielectric loss factor of absorber (ε“)', 'Bio-oil yield (%)',
       'Syngas yield (%)', 'Syngas composition (H₂, mol%)',
       'Syngas composition (CH₄, mol%)', 'Syngas composition (CO₂, mol%)',
       'Syngas composition (CO, mol%)', 'Biochar yield (%)',
       'Biochar calorific value (MJ/kg)', 'Biochar H/C ratio (-)',
       'Biochar H/N ratio (-)', 'Biochar O/C ratio (-)'],
      dtype='object')
```

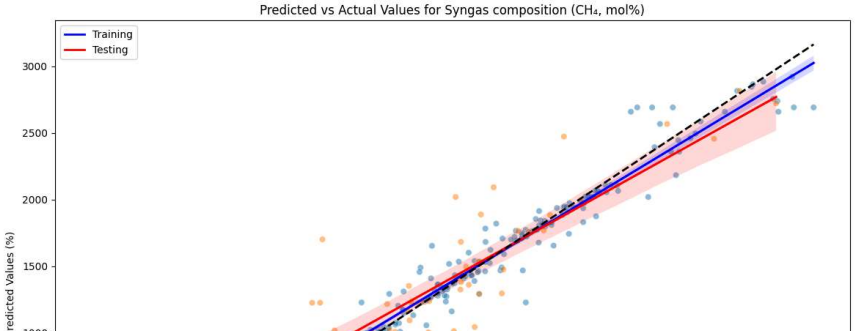
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error

def plot_predicted_vs_actual_multi(df, feature_columns, target_columns, output_file=None):
    num_cols = len(target_columns)
    fig, axs = plt.subplots(num_cols, 1, figsize=(12, 8*num_cols))

    for idx, target_column in enumerate(target_columns):
        # Assign the feature columns to X and the target column to y
        X = df[feature_columns].values
        y = df[target_column].values

        # Split the dataset 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)

        # Standardize the features
        scaler = StandardScaler()
        X_train = scaler.fit_transform(X_train)
        X_test = scaler.transform(X_test)

        # Create and train the Random Forest model
        rf = RandomForestRegressor(n_estimators=100, random_state=42)
        rf.fit(X_train, y_train)

        # Make predictions
        y_train_pred = rf.predict(X_train)
        y_test_pred = rf.predict(X_test)

        # Calculate R²
        r2_train = r2_score(y_train, y_train_pred)
        r2_test = r2_score(y_test, y_test_pred)

        # Calculate RMSE
        rmse_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
        rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))

        # Calculate RRMSE
        rrmse_train = rmse_train / np.mean(y_train)
        rrmse_test = rmse_test / np.mean(y_test)

        # Convert actual and predicted values to percentages for plotting
        y_train_percentage = y_train * 100
        y_train_pred_percentage = y_train_pred * 100
        y_test_percentage = y_test * 100
        y_test_pred_percentage = y_test_pred * 100

        # Combine data into a DataFrame for plotting with seaborn
        df_plot = pd.DataFrame({
            'Actual': np.concatenate((y_train_percentage, y_test_percentage)),
            'Predicted': np.concatenate((y_train_pred_percentage, y_test_pred_percentage)),
            'Dataset': ['Training'] * len(y_train_percentage) + ['Testing'] * len(y_test_percentage)
        })

        # Plot on the respective subplot
        ax = axs[idx]
        sns.scatterplot(data=df_plot, x='Actual', y='Predicted', hue='Dataset', alpha=0.5, ax=ax)

        # Training data trend line and confidence interval
        sns.regplot(data=df_plot[df_plot['Dataset'] == 'Training'], x='Actual', y='Predicted', scatter=False, color='blue', ax=ax)

        # Testing data trend line and confidence interval
        sns.regplot(data=df_plot[df_plot['Dataset'] == 'Testing'], x='Actual', y='Predicted', scatter=False, color='red', ax=ax)

        # Add diagonal line
        ax.plot([min(y) * 100, max(y) * 100], [min(y) * 100, max(y) * 100], 'k--', lw=2)

        # Add labels and title with target column
        ax.set_xlabel('Actual Values (%)')
        ax.set_ylabel('Predicted Values (%)')
        ax.set_title(f'Predicted vs Actual Values for {target_column}')

        # Create custom legend with both training and testing labels
        custom_lines = [plt.Line2D([0], [0], color='blue', lw=2),
                        plt.Line2D([0], [0], color='red', lw=2)]
        ax.legend(custom_lines, ['Training', 'Testing'], loc='upper left')
```

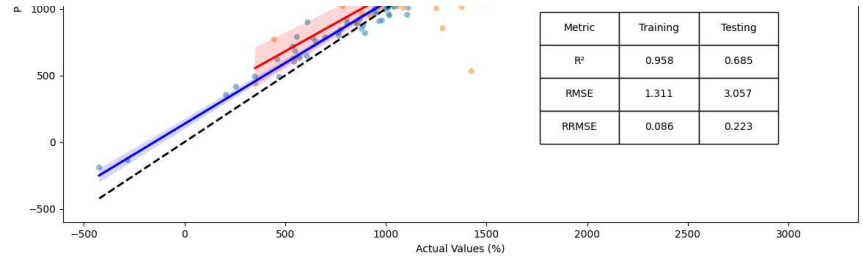
```
            # Add table with performance metrics
            table_data = [['R²', f'{r2_train:.3f}', f'{r2_test:.3f}'],
                          ['RMSE', f'{rmse_train:.3f}', f'{rmse_test:.3f}'],
                          ['RRMSE', f'{rrmse_train:.3f}', f'{rrmse_test:.3f}']]
            table = ax.table(cellText=table_data,
                             colLabels=['Metric', 'Training', 'Testing'],
                             cellLoc='center',
                             loc='bottom right',  # Adjusted to bottom right
                             bbox=[0.6, 0.15, 0.3, 0.25])  # Adjust bbox to control position and size

            # Adjust table font size and style
            table.auto_set_font_size(False)
            table.set_fontsize(10)

        plt.tight_layout()

        # Save the figure to a file if output_file is provided
        if output_file:
            plt.savefig(output_file)

        plt.show()

    # Example usage:
    # Assuming `df`, `feature_columns`, and `target_columns` are defined appropriately
    # Also assuming `output_file` is the file path where you want to save the plot
    output_file = 'predicted_vs_actual_multi_rf.png'
    plot_predicted_vs_actual_multi(df, feature_columns, target_columns, output_file)
```


Predicted vs Actual Values for Bio-oil yield (%)
Training
Testing
Predicted Values (%)
Actual Values (%)
Metric | Training | Testing
R² | 0.898 | 0.666
RMSE | 4.447 | 8.055
RRMSE | 0.148 | 0.264


Predicted vs Actual Values for Syngas yield (%)
Training
Testing
Predicted Values (%)
Actual Values (%)
Metric | Training | Testing
R² | 0.915 | 0.675
RMSE | 5.422 | 10.484
RRMSE | 0.145 | 0.271


Predicted vs Actual Values for Syngas composition (H₂, mol%)
Training
Testing
Predicted Values (%)
Actual Values (%)
Metric | Training | Testing
R² | 0.974 | 0.593
RMSE | 2.651 | 8.606
RRMSE | 0.124 | 0.391

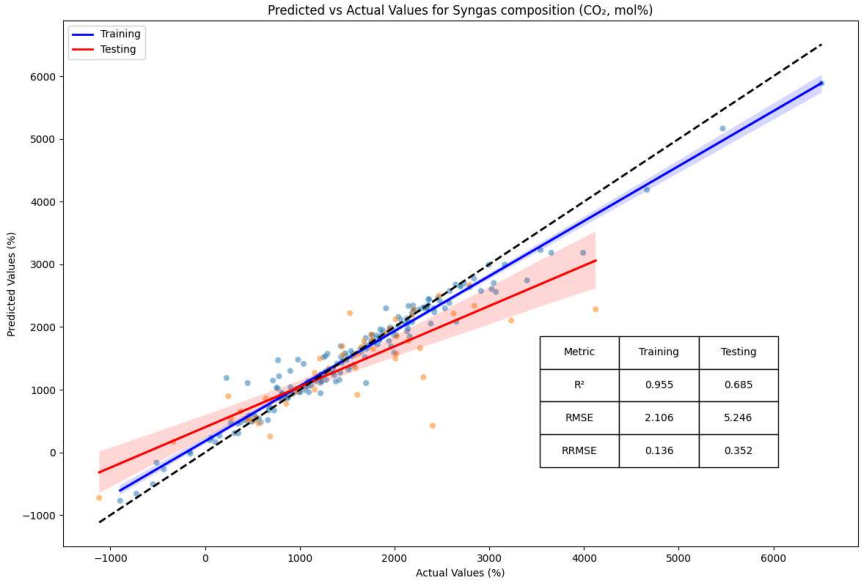

Predicted vs Actual Values for Syngas composition (CH₄, mol%)
Training
Testing



Metric	Training	Testing
R^2	0.958	0.685
RMSE	1.311	3.057
RRMSE	0.086	0.223

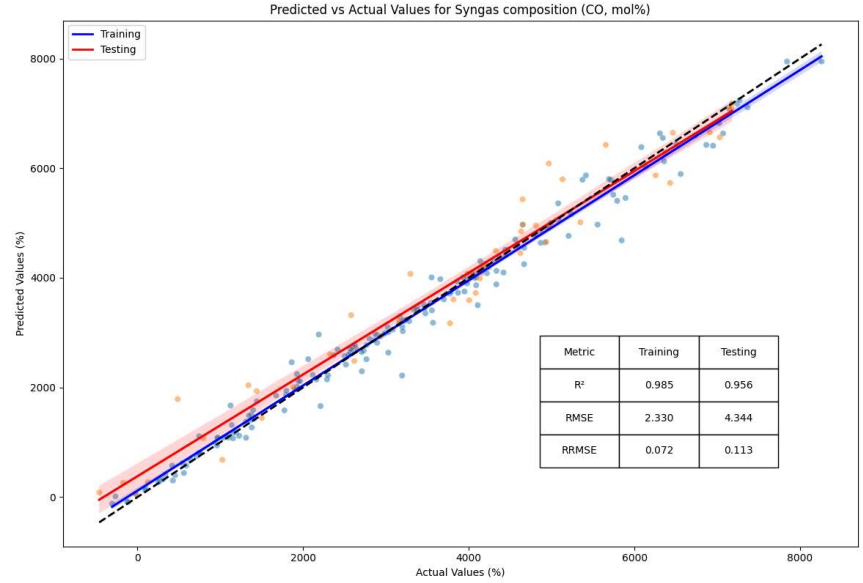
Actual Values (%)

Predicted vs Actual Values for Syngas composition (CO_2, mol%)



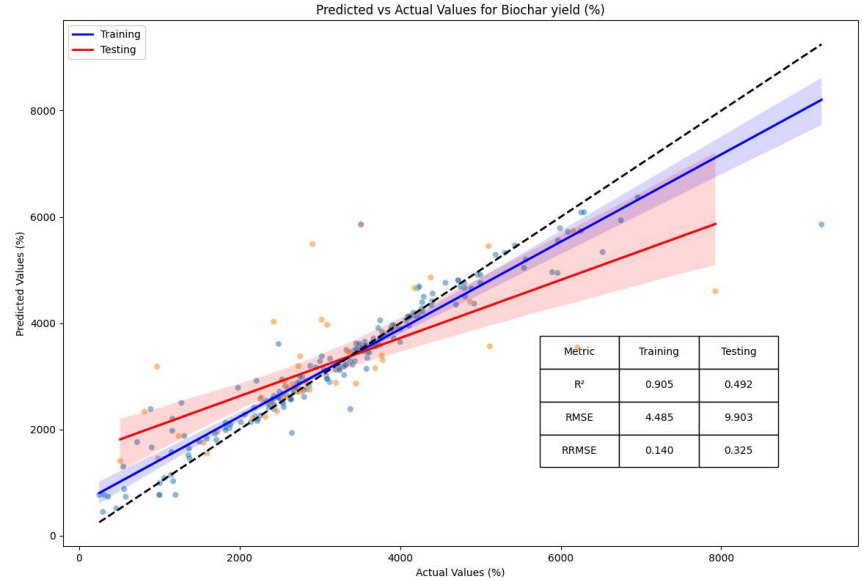
Metric	Training	Testing
R^2	0.955	0.685
RMSE	2.106	5.246
RRMSE	0.136	0.352

Predicted vs Actual Values for Syngas composition (CO, mol%)



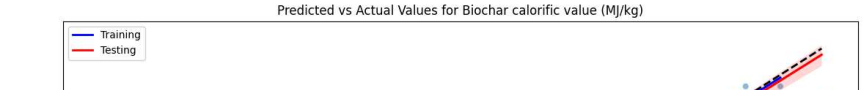
Metric	Training	Testing
R^2	0.985	0.956
RMSE	2.330	4.344
RRMSE	0.072	0.113

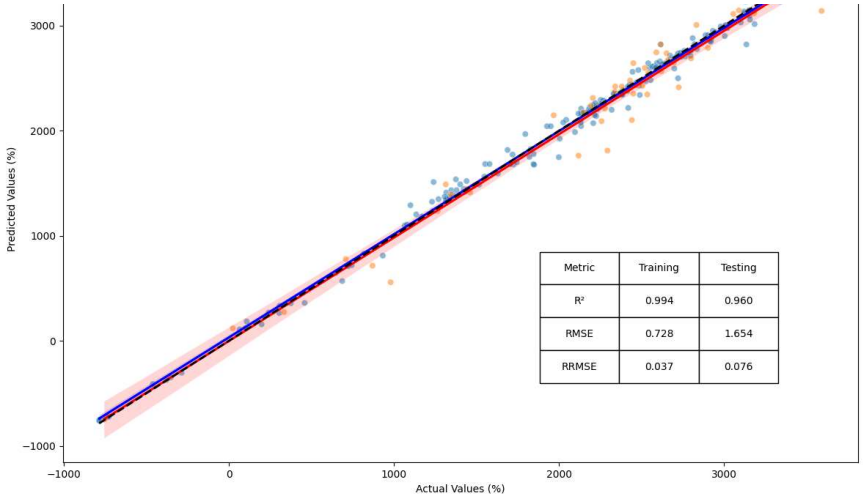
Predicted vs Actual Values for Biochar yield (%)



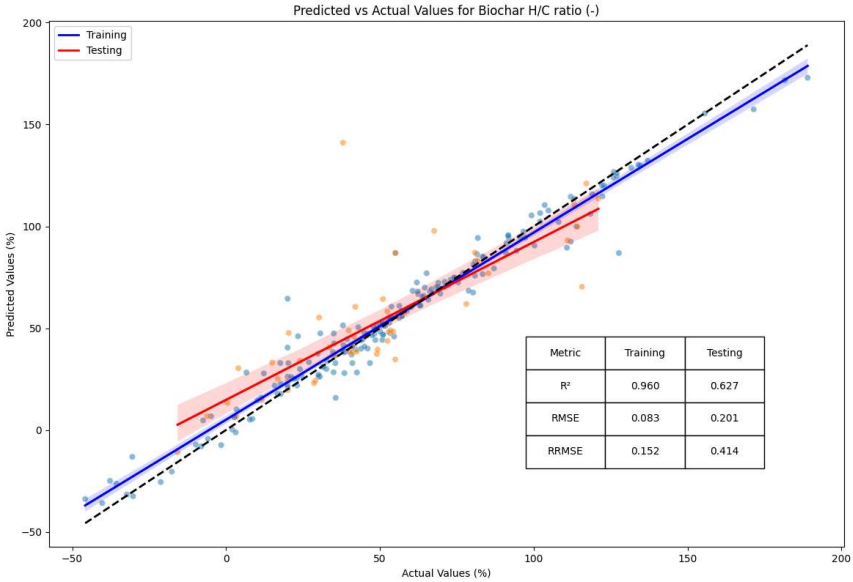
Metric	Training	Testing
R^2	0.905	0.492
RMSE	4.485	9.903
RRMSE	0.140	0.325

Predicted vs Actual Values for Biochar calorific value (MJ/kg)

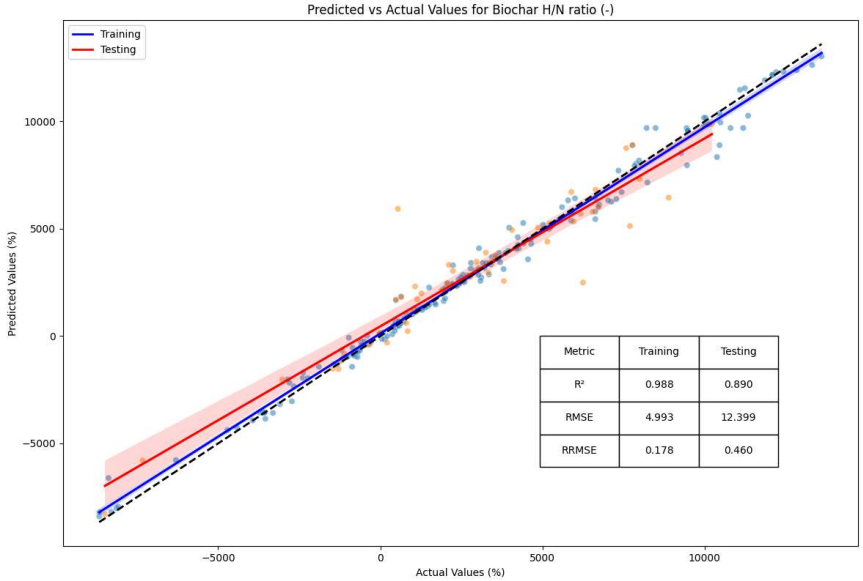




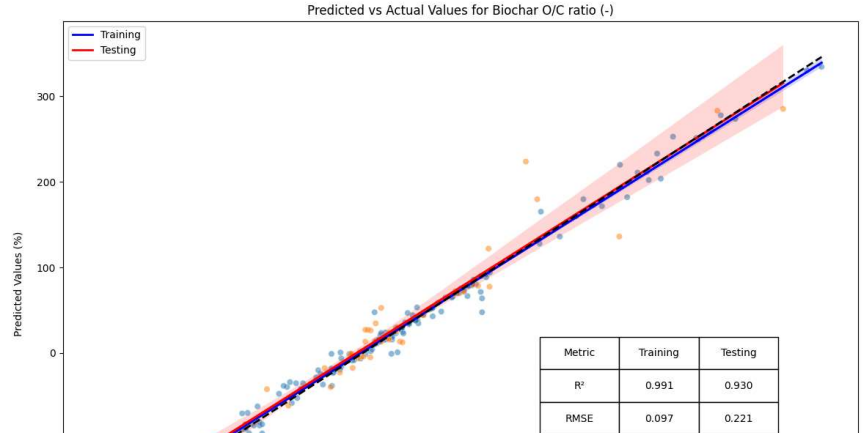
Predicted vs Actual Values for Biochar H/C ratio (-)

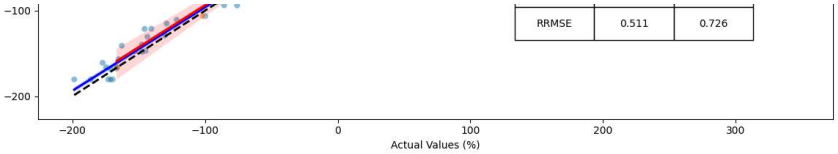


Predicted vs Actual Values for Biochar H/N ratio (-)



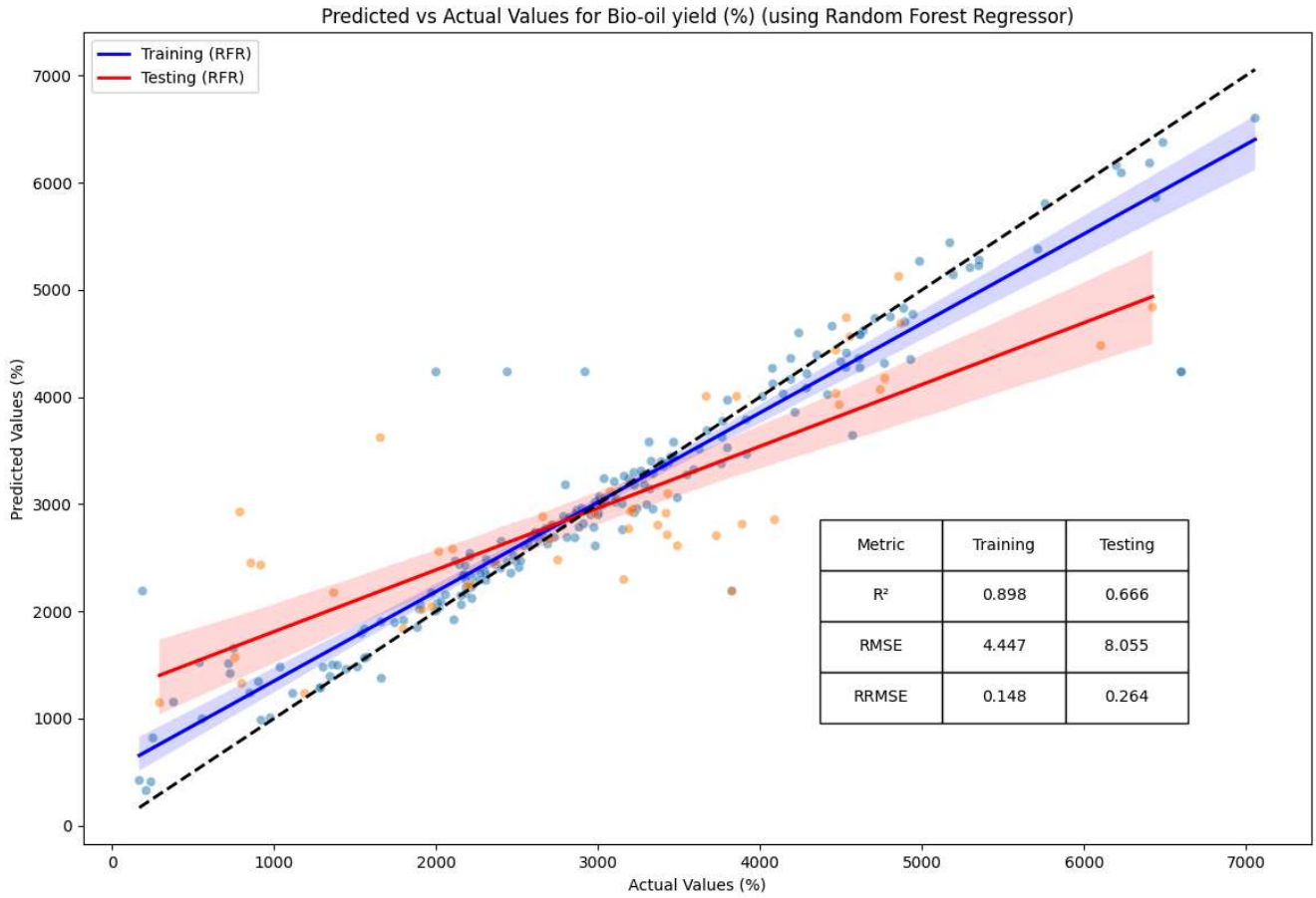
Predicted vs Actual Values for Biochar O/C ratio (-)

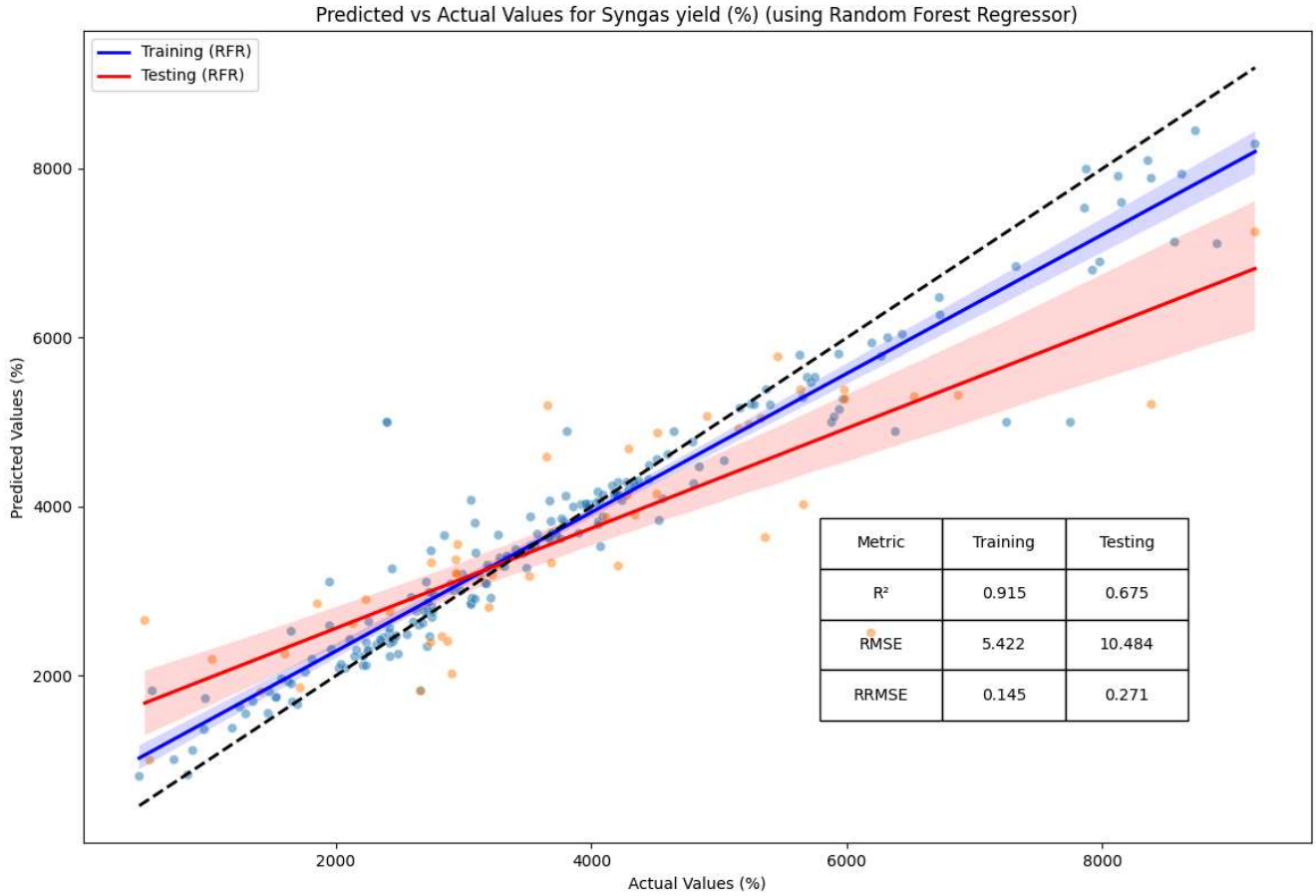


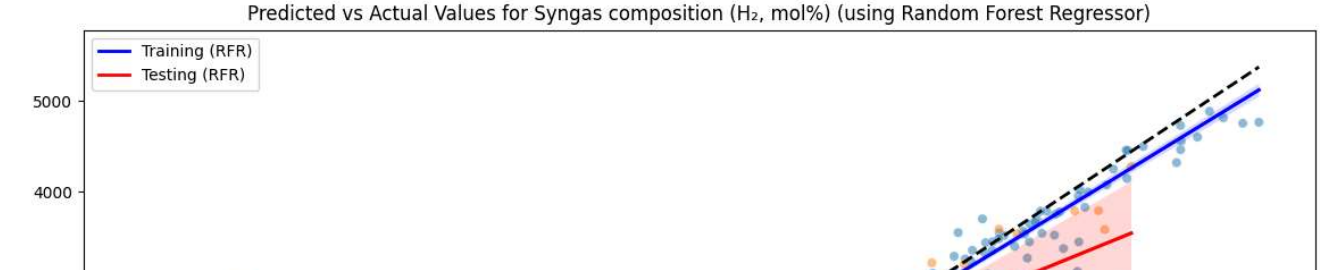

−100
−200
−200
−100
0
100
200
300
Actual Values (%)
RRMSE
0.511
0.726

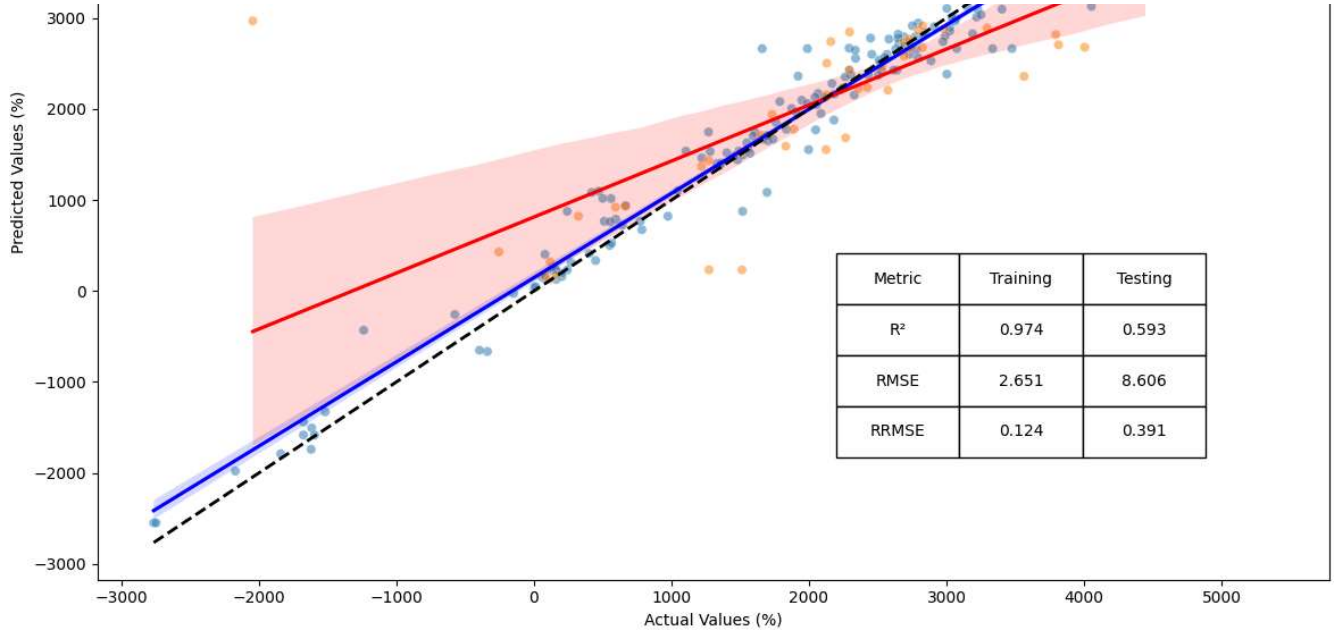
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error

def plot_predicted_vs_actual_multi(df, feature_columns, target_columns, output_file=None):
    num_cols = len(target_columns)
    fig, axs = plt.subplots(num_cols, 1, figsize=(12, 8*num_cols))

    for idx, target_column in enumerate(target_columns):
        # Assign the feature columns to X and the target column to y
        X = df[feature_columns].values
        y = df[target_column].values

        # Split the dataset 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)

        # Standardize the features
        scaler = StandardScaler()
        X_train = scaler.fit_transform(X_train)
        X_test = scaler.transform(X_test)

        # Create and train the Random Forest model
        rf = RandomForestRegressor(n_estimators=100, random_state=42)
        rf.fit(X_train, y_train)

        # Make predictions
        y_train_pred = rf.predict(X_train)
        y_test_pred = rf.predict(X_test)

        # Calculate R²
        r2_train = r2_score(y_train, y_train_pred)
        r2_test = r2_score(y_test, y_test_pred)

        # Calculate RMSE
        rmse_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
        rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))

        # Calculate RRMSE
        rrmse_train = rmse_train / np.mean(y_train)
        rrmse_test = rmse_test / np.mean(y_test)

        # Convert actual and predicted values to percentages for plotting
        y_train_percentage = y_train * 100
        y_train_pred_percentage = y_train_pred * 100
        y_test_percentage = y_test * 100
        y_test_pred_percentage = y_test_pred * 100

        # Combine data into a DataFrame for plotting with seaborn
        df_plot = pd.DataFrame({
            'Actual': np.concatenate((y_train_percentage, y_test_percentage)),
            'Predicted': np.concatenate((y_train_pred_percentage, y_test_pred_percentage)),
            'Dataset': ['Training'] * len(y_train_percentage) + ['Testing'] * len(y_test_percentage)
        })

        # Plot on the respective subplot
        ax = axs[idx]
        sns.scatterplot(data=df_plot, x='Actual', y='Predicted', hue='Dataset', alpha=0.5, ax=ax)

        # Training data trend line and confidence interval
        sns.regplot(data=df_plot[df_plot['Dataset'] == 'Training'], x='Actual', y='Predicted', scatter=False, color='blue', ax=ax)

        # Testing data trend line and confidence interval
        sns.regplot(data=df_plot[df_plot['Dataset'] == 'Testing'], x='Actual', y='Predicted', scatter=False, color='red', ax=ax)

        # Add diagonal line
        ax.plot([min(y) * 100, max(y) * 100], [min(y) * 100, max(y) * 100], 'k--', lw=2)

        # Add labels and title with target column
        ax.set_xlabel('Actual Values (%)')
        ax.set_ylabel('Predicted Values (%)')
        ax.set_title(f'Predicted vs Actual Values for {target_column} (using Random Forest Regressor)')

        # Create custom legend with both training and testing labels
        custom_lines = [plt.Line2D([0], [0], color='blue', lw=2),
                        plt.Line2D([0], [0], color='red', lw=2)]
        ax.legend(custom_lines, ['Training (RFR)', 'Testing (RFR)'], loc='upper left')
```

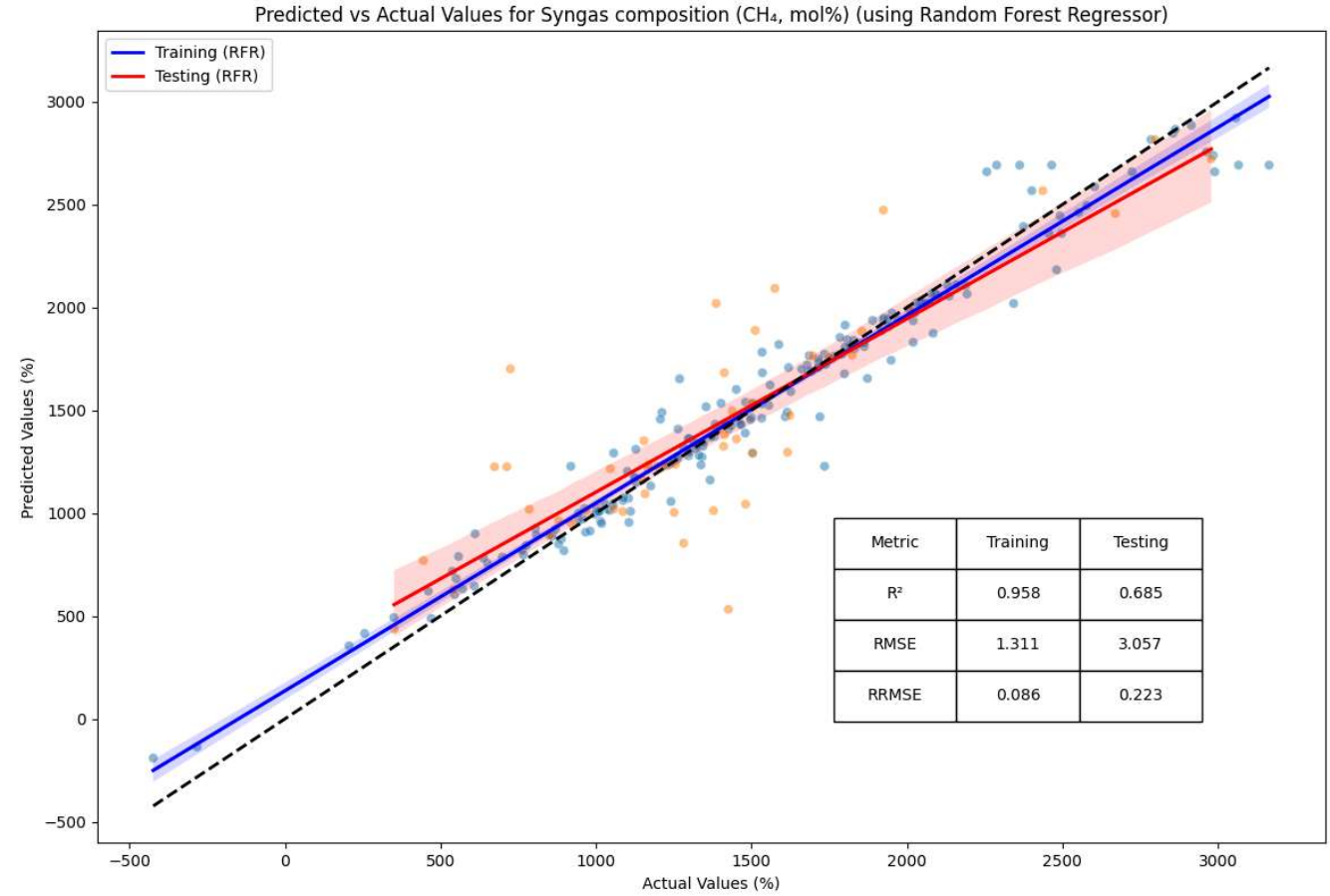
```
            # Add table with performance metrics
            table_data = [['R²', f'{r2_train:.3f}', f'{r2_test:.3f}'],
                          ['RMSE', f'{rmse_train:.3f}', f'{rmse_test:.3f}'],
                          ['RRMSE', f'{rrmse_train:.3f}', f'{rrmse_test:.3f}']]
            table = ax.table(cellText=table_data,
                             colLabels=['Metric', 'Training', 'Testing'],
                             cellLoc='center',
                             loc='bottom right',  # Adjusted to bottom right
                             bbox=[0.6, 0.15, 0.3, 0.25])  # Adjust bbox to control position and size

            # Adjust table font size and style
            table.auto_set_font_size(False)
            table.set_fontsize(10)

        plt.tight_layout()

        # Save the figure to a file if output_file is provided
        if output_file:
            plt.savefig(output_file)

        plt.show()

    # Example usage:
    # Assuming `df`, `feature_columns`, and `target_columns` are defined appropriately
    # Also assuming `output_file` is the file path where you want to save the plot
    output_file = 'RFR_predicted_vs_actual_multi_rf.png'
    plot_predicted_vs_actual_multi(df, feature_columns, target_columns, output_file)
```

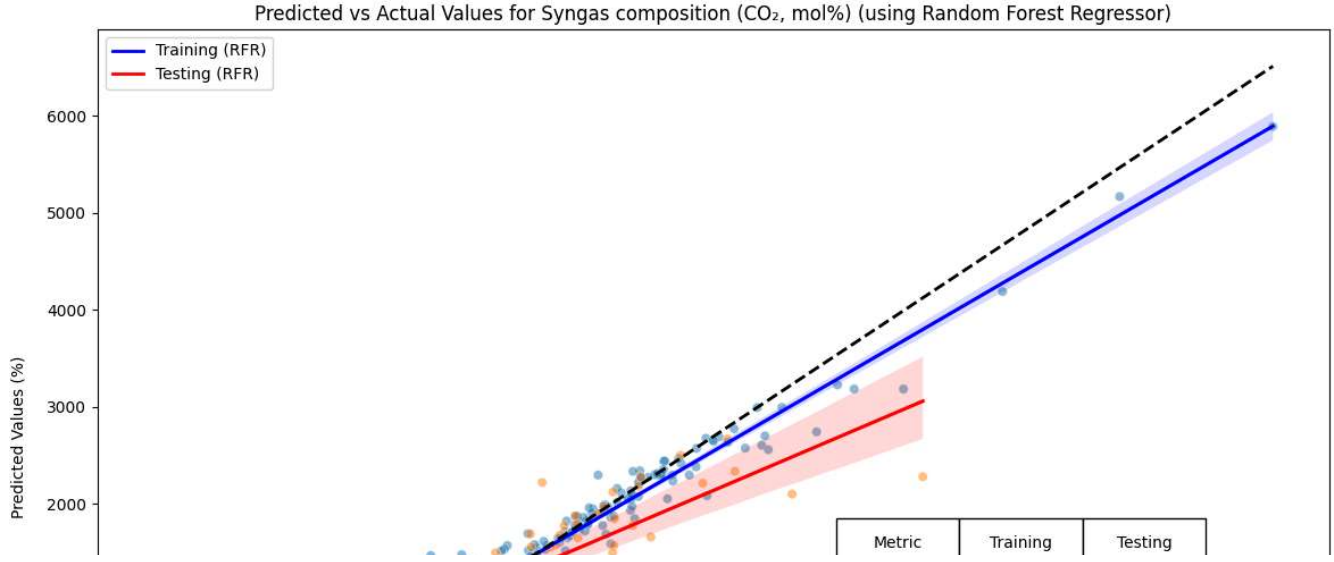

Predicted vs Actual Values for Bio-oil yield (%) (using Random Forest Regressor)
Training (RFR)
Testing (RFR)
Predicted Values (%)
Actual Values (%)
0
1000
2000
3000
4000
5000
6000
7000
Metric | Training | Testing
R² | 0.898 | 0.666
RMSE | 4.447 | 8.055
RRMSE | 0.148 | 0.264

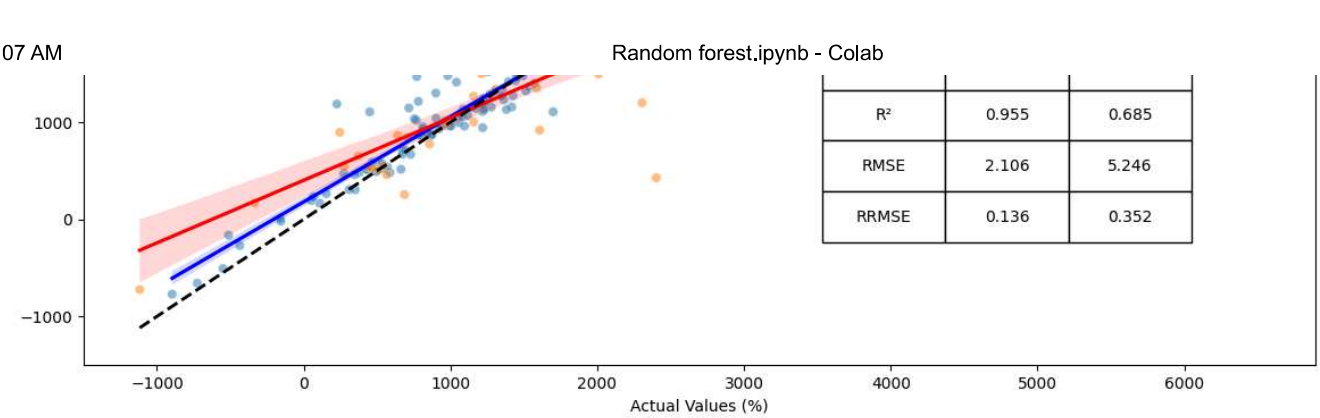

Predicted vs Actual Values for Syngas yield (%) (using Random Forest Regressor)
Training (RFR)
Testing (RFR)
Predicted Values (%)
Actual Values (%)
2000
4000
6000
8000
Metric | Training | Testing
R² | 0.915 | 0.675
RMSE | 5.422 | 10.484
RRMSE | 0.145 | 0.271


Predicted vs Actual Values for Syngas composition (H₂, mol%) (using Random Forest Regressor)
Training (RFR)
Testing (RFR)
5000
4000

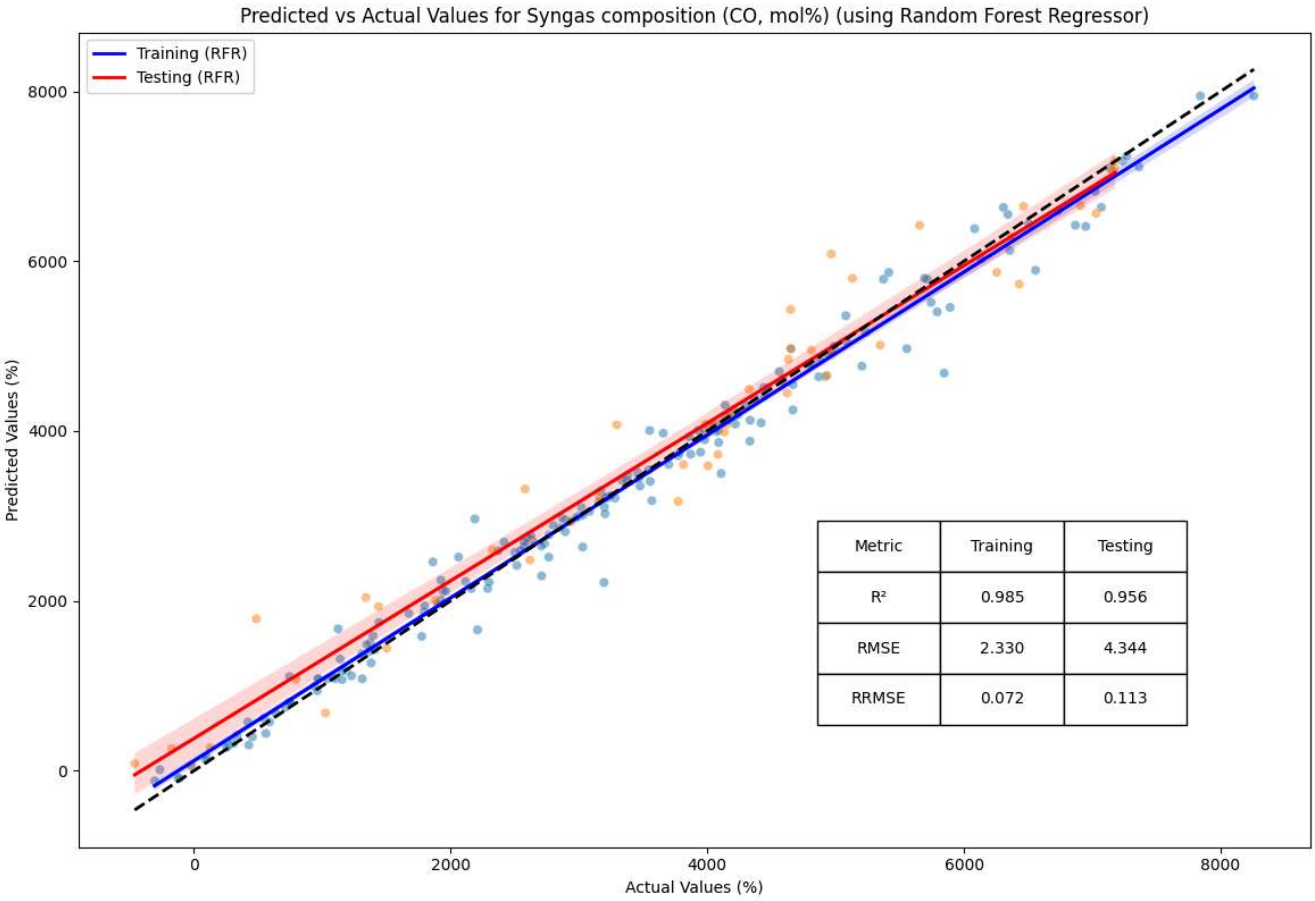

Predicted Values (%)
Actual Values (%)
Metric | Training | Testing
R² | 0.974 | 0.593
RMSE | 2.651 | 8.606
RRMSE | 0.124 | 0.391


Predicted vs Actual Values for Syngas composition (CH₄, mol%) (using Random Forest Regressor)
Training (RFR)
Testing (RFR)
Predicted Values (%)
Actual Values (%)
Metric | Training | Testing
R² | 0.958 | 0.685
RMSE | 1.311 | 3.057
RRMSE | 0.086 | 0.223

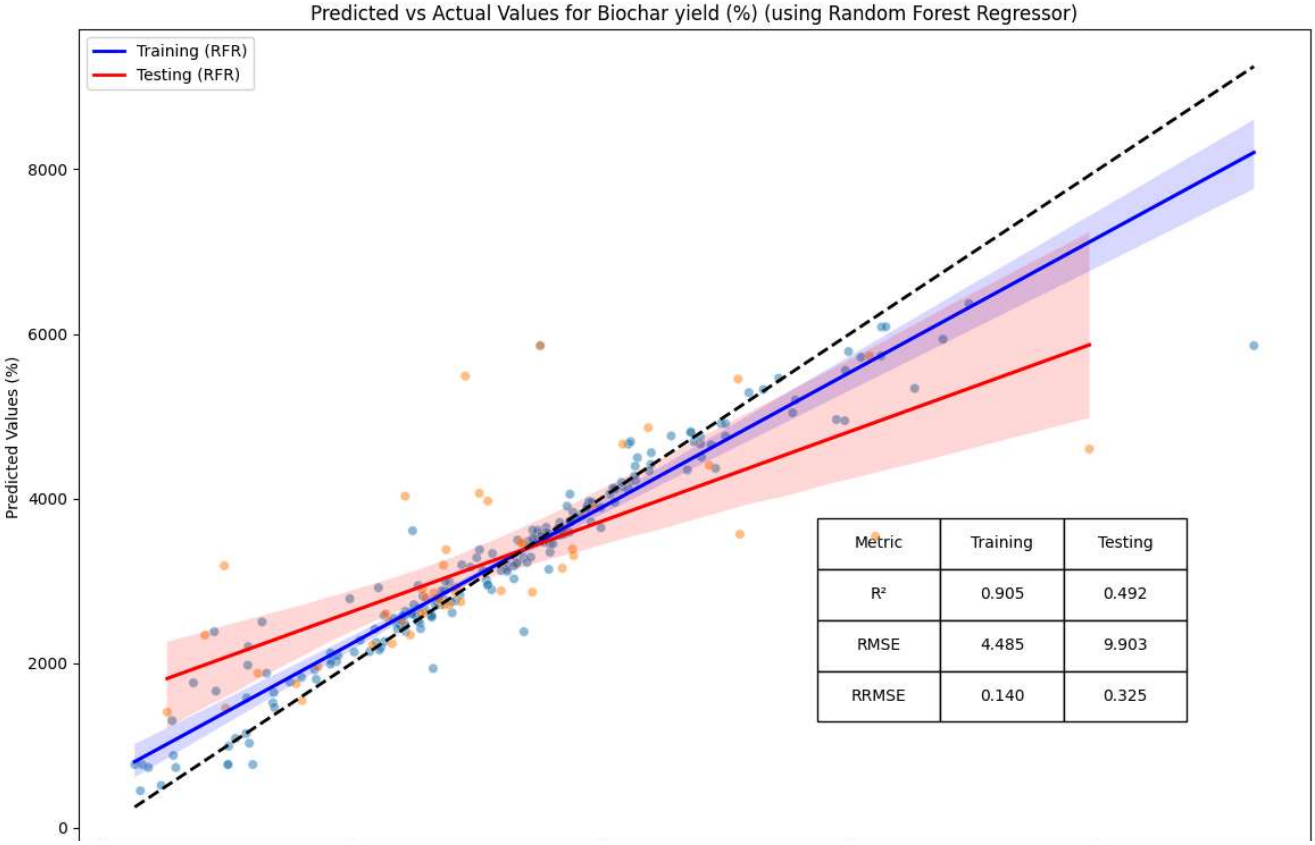

Predicted vs Actual Values for Syngas composition (CO₂, mol%) (using Random Forest Regressor)
Training (RFR)
Testing (RFR)
Predicted Values (%)
Metric | Training | Testing



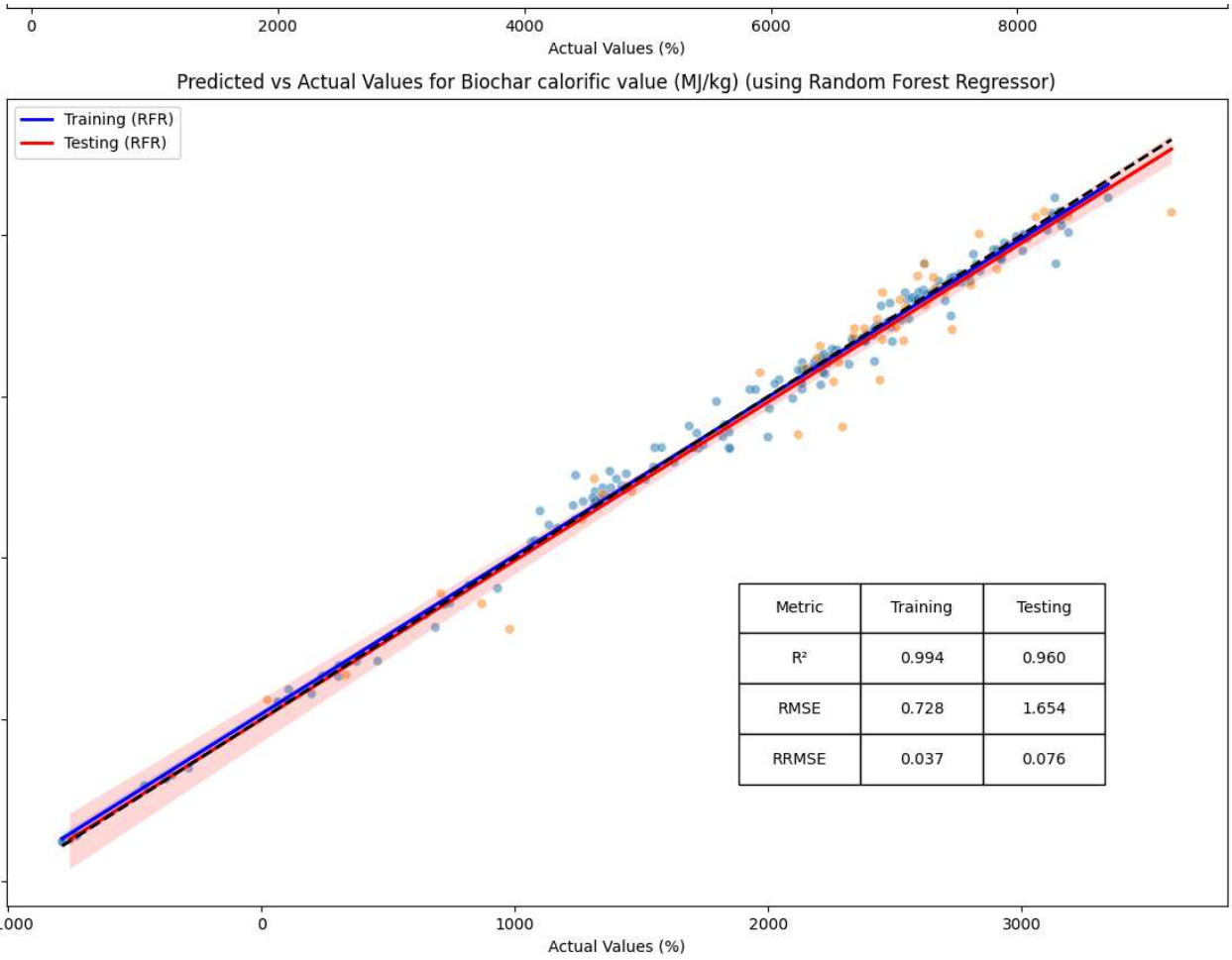
R²	0.955	0.685
RMSE	2.106	5.246
RRMSE	0.136	0.352

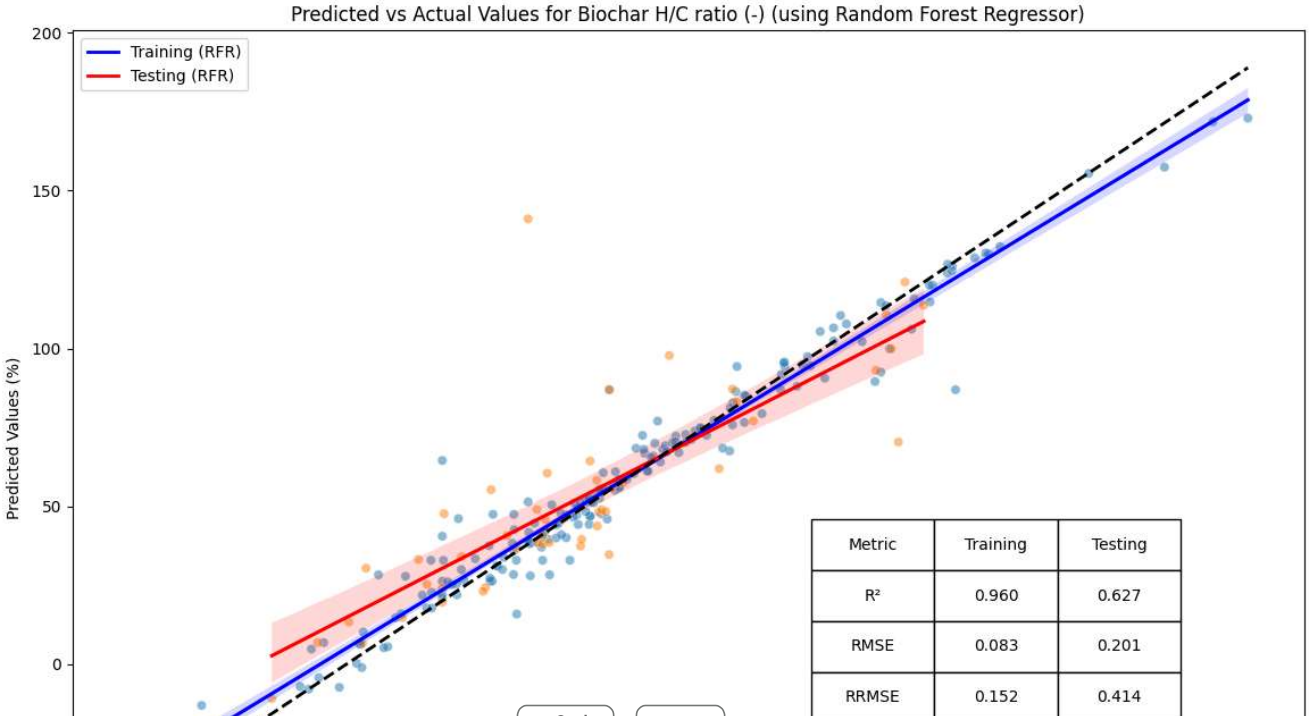


Metric	Training	Testing
R²	0.985	0.956
RMSE	2.330	4.344
RRMSE	0.072	0.113



Metric	Training	Testing
R²	0.905	0.492
RMSE	4.485	9.903
RRMSE	0.140	0.325


0
2000
4000
6000
8000
Actual Values (%)
Predicted vs Actual Values for Biochar calorific value (MJ/kg) (using Random Forest Regressor)
Training (RFR)
Testing (RFR)
3000
2000
1000
0
−1000
Predicted Values (%)
−1000
0
1000
2000
3000
Actual Values (%)
Metric | Training | Testing
R² | 0.994 | 0.960
RMSE | 0.728 | 1.654
RRMSE | 0.037 | 0.076

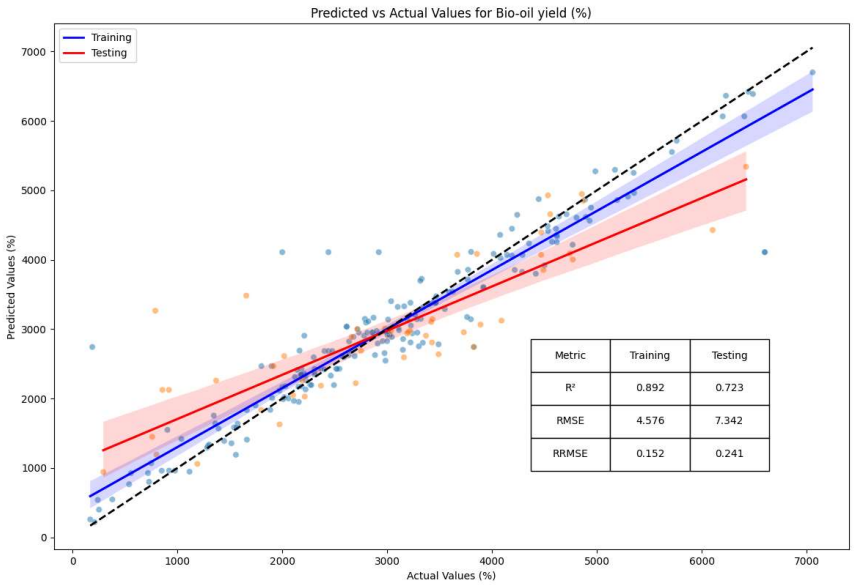

Predicted vs Actual Values for Biochar H/C ratio (-) (using Random Forest Regressor)
Training (RFR)
Testing (RFR)
200
150
100
50
0
Predicted Values (%)
Metric | Training | Testing
R² | 0.960 | 0.627
RMSE | 0.083 | 0.201
RRMSE | 0.152 | 0.414

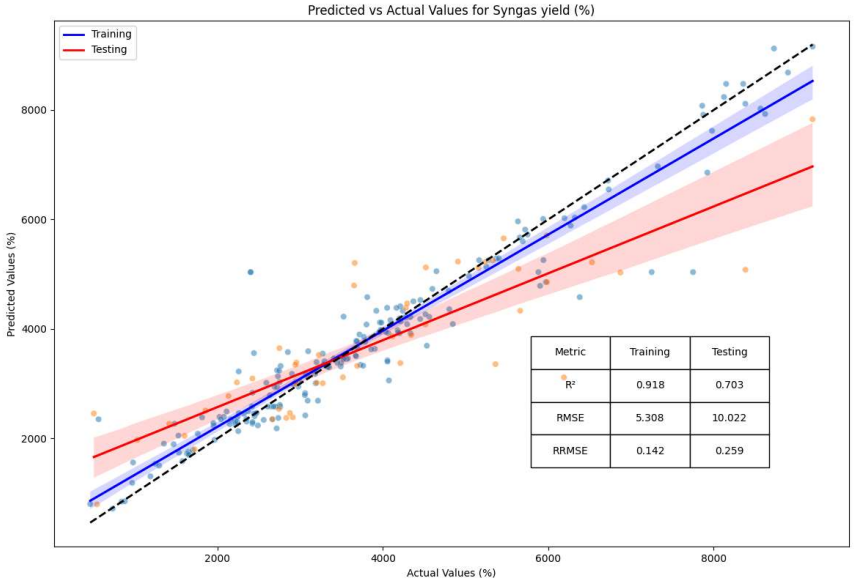
```
import numpy as np
import pandas as pd
from sklearn.svm import SVR
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error


df = pd.read_excel('/content/cleaned_imputed_data.xlsx')

df.head()
print(df.columns)

feature_columns = [
    'Carbon content (wt%)',
    'Hydrogen content (wt%)',
    'Nitrogen content (wt%)',
    'Oxygen content (wt%)',
    'Sulfur content (wt%)',
    'Volatile matter (wt%)',
    'Fixed carbon (wt%)',
    'Ash content (wt%)',
    'Reaction temperature (°C)',
    'Microwave power (W)',
    'Reaction time (min)',
    'Microwave absorber percentage (%)',
    'Dielectric constant of absorber (ε’)',
    'Dielectric loss factor of absorber (ε“)'
]
target_columns = ['Bio-oil yield (%)',
       'Syngas yield (%)', 'Syngas composition (H₂, mol%)',
       'Syngas composition (CH₄, mol%)', 'Syngas composition (CO₂, mol%)',
       'Syngas composition (CO, mol%)', 'Biochar yield (%)',
       'Biochar calorific value (MJ/kg)', 'Biochar H/C ratio (-)',
       'Biochar H/N ratio (-)', 'Biochar O/C ratio (-)']
```

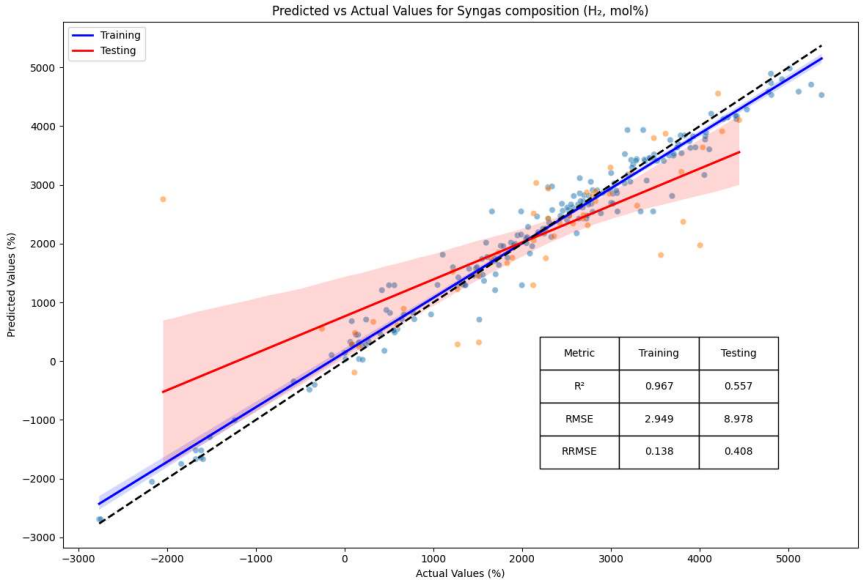
```
Index(['Reference (DOI)', 'Biomass type', 'Carbon content (wt%)',
       'Hydrogen content (wt%)', 'Nitrogen content (wt%)',
       'Oxygen content (wt%)', 'Sulfur content (wt%)', 'Volatile matter (wt%)',
       'Fixed carbon (wt%)', 'Ash content (wt%)', 'Reaction temperature (°C)',
       'Microwave power (W)', 'Reaction time (min)',
       'Microwave absorber percentage (%)',
       'Dielectric constant of absorber (ε’)',
       'Dielectric loss factor of absorber (ε“)', 'Bio-oil yield (%)',
       'Syngas yield (%)', 'Syngas composition (H₂, mol%)',
       'Syngas composition (CH₄, mol%)', 'Syngas composition (CO₂, mol%)',
       'Syngas composition (CO, mol%)', 'Biochar yield (%)',
       'Biochar calorific value (MJ/kg)', 'Biochar H/C ratio (-)',
       'Biochar H/N ratio (-)', 'Biochar O/C ratio (-)'],
      dtype='object')
```

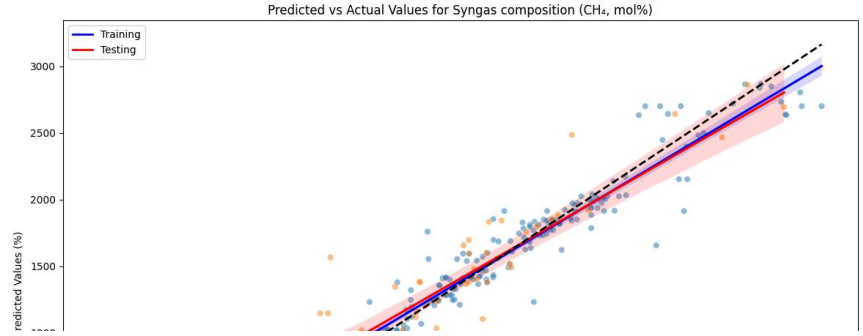
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error

def plot_predicted_vs_actual_multi(df, feature_columns, target_columns, output_file=None):
    num_cols = len(target_columns)
    fig, axs = plt.subplots(num_cols, 1, figsize=(12, 8*num_cols))

    for idx, target_column in enumerate(target_columns):
        # Assign the feature columns to X and the target column to y
        X = df[feature_columns].values
        y = df[target_column].values

        # Split the dataset 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)

        # Standardize the features
        scaler = StandardScaler()
        X_train = scaler.fit_transform(X_train)
        X_test = scaler.transform(X_test)

        # Create and train the Gradient Boosting model
        gb = GradientBoostingRegressor(n_estimators=100, random_state=42)
        gb.fit(X_train, y_train)

        # Make predictions
        y_train_pred = gb.predict(X_train)
        y_test_pred = gb.predict(X_test)

        # Calculate R²
        r2_train = r2_score(y_train, y_train_pred)
        r2_test = r2_score(y_test, y_test_pred)

        # Calculate RMSE
        rmse_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
        rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))

        # Calculate RRMSE
        rrmse_train = rmse_train / np.mean(y_train)
        rrmse_test = rmse_test / np.mean(y_test)

        # Convert actual and predicted values to percentages for plotting
        y_train_percentage = y_train * 100
        y_train_pred_percentage = y_train_pred * 100
        y_test_percentage = y_test * 100
        y_test_pred_percentage = y_test_pred * 100

        # Combine data into a DataFrame for plotting with seaborn
        df_plot = pd.DataFrame({
            'Actual': np.concatenate((y_train_percentage, y_test_percentage)),
            'Predicted': np.concatenate((y_train_pred_percentage, y_test_pred_percentage)),
            'Dataset': ['Training'] * len(y_train_percentage) + ['Testing'] * len(y_test_percentage)
        })

        # Plot on the respective subplot
        ax = axs[idx]
        sns.scatterplot(data=df_plot, x='Actual', y='Predicted', hue='Dataset', alpha=0.5, ax=ax)

        # Training data trend line and confidence interval
        sns.regplot(data=df_plot[df_plot['Dataset'] == 'Training'], x='Actual', y='Predicted', scatter=False, color='blue', ax=ax)

        # Testing data trend line and confidence interval
        sns.regplot(data=df_plot[df_plot['Dataset'] == 'Testing'], x='Actual', y='Predicted', scatter=False, color='red', ax=ax)

        # Add diagonal line
        ax.plot([min(y) * 100, max(y) * 100], [min(y) * 100, max(y) * 100], 'k--', lw=2)

        # Add labels and title with target column
        ax.set_xlabel('Actual Values (%)')
        ax.set_ylabel('Predicted Values (%)')
        ax.set_title(f'Predicted vs Actual Values for {target_column}')

        # Create custom legend with both training and testing labels
        custom_lines = [plt.Line2D([0], [0], color='blue', lw=2),
                        plt.Line2D([0], [0], color='red', lw=2)]
        ax.legend(custom_lines, ['Training', 'Testing'], loc='upper left')
```

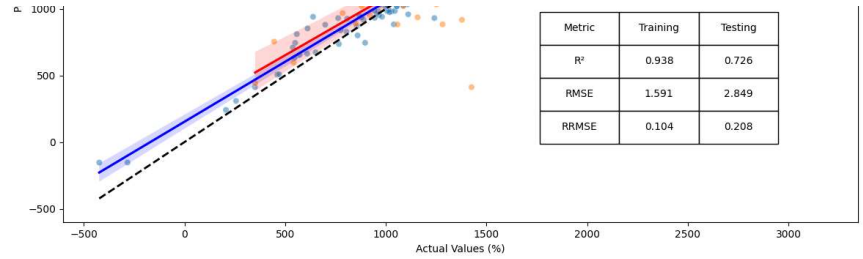
```
            # Add table with performance metrics
            table_data = [['R²', f'{r2_train:.3f}', f'{r2_test:.3f}'],
                          ['RMSE', f'{rmse_train:.3f}', f'{rmse_test:.3f}'],
                          ['RRMSE', f'{rrmse_train:.3f}', f'{rrmse_test:.3f}']]
            table = ax.table(cellText=table_data,
                             colLabels=['Metric', 'Training', 'Testing'],
                             cellLoc='center',
                             loc='bottom right',  # Adjusted to bottom right
                             bbox=[0.6, 0.15, 0.3, 0.25])  # Adjust bbox to control position and size

            # Adjust table font size and style
            table.auto_set_font_size(False)
            table.set_fontsize(10)

        plt.tight_layout()

        # Save the figure to a file if output_file is provided
        if output_file:
            plt.savefig(output_file)

        plt.show()

    # Example usage:
    # Assuming `df`, `feature_columns`, and `target_columns` are defined appropriately
    # Also assuming `output_file` is the file path where you want to save the plot
    output_file = 'predicted_vs_actual_multi_gb.png'
    plot_predicted_vs_actual_multi(df, feature_columns, target_columns, output_file)
```


Predicted vs Actual Values for Bio-oil yield (%)
Training
Testing
Predicted Values (%)
Actual Values (%)
Metric | Training | Testing
R² | 0.892 | 0.723
RMSE | 4.576 | 7.342
RRMSE | 0.152 | 0.241


Predicted vs Actual Values for Syngas yield (%)
Training
Testing
Predicted Values (%)
Actual Values (%)
Metric | Training | Testing
R² | 0.918 | 0.703
RMSE | 5.308 | 10.022
RRMSE | 0.142 | 0.259

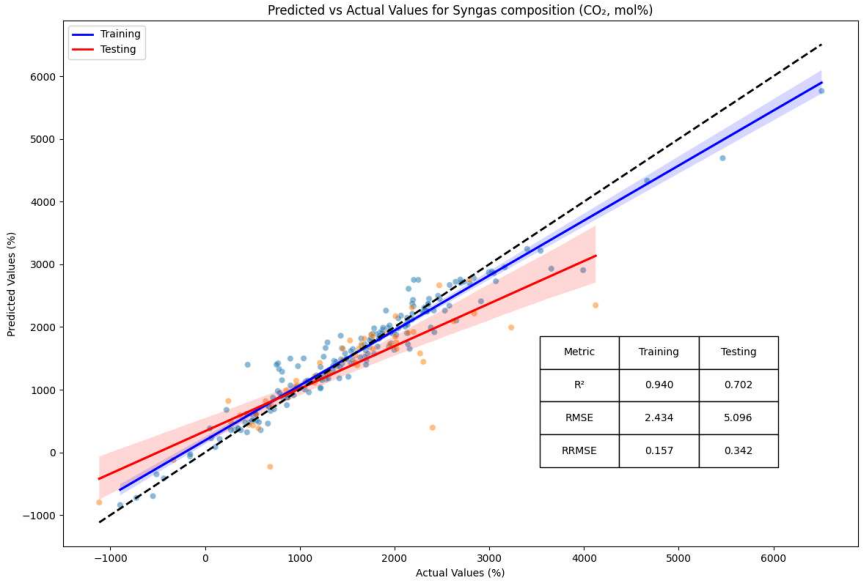

Predicted vs Actual Values for Syngas composition (H₂, mol%)
Training
Testing
Predicted Values (%)
Actual Values (%)
Metric | Training | Testing
R² | 0.967 | 0.557
RMSE | 2.949 | 8.978
RRMSE | 0.138 | 0.408


Predicted vs Actual Values for Syngas composition (CH₄, mol%)
Training
Testing

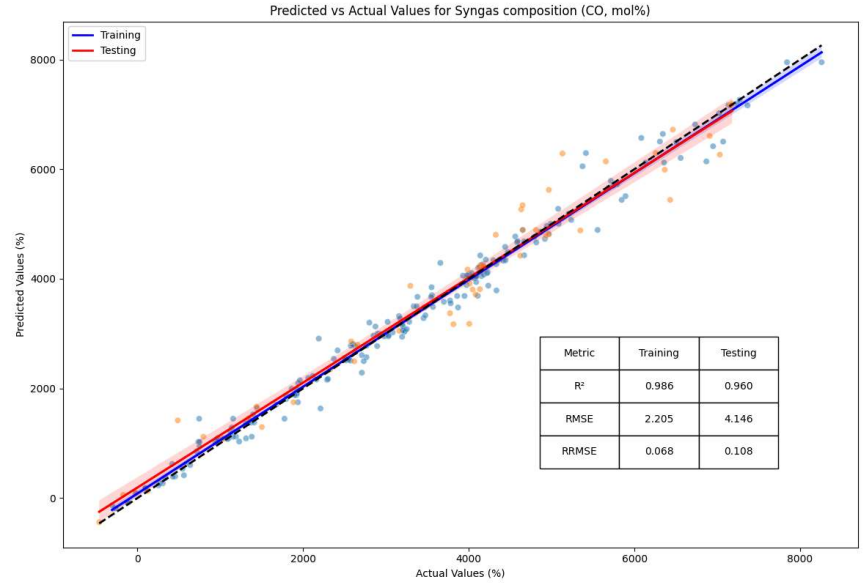


Metric	Training	Testing
R^2	0.938	0.726
RMSE	1.591	2.849
RRMSE	0.104	0.208

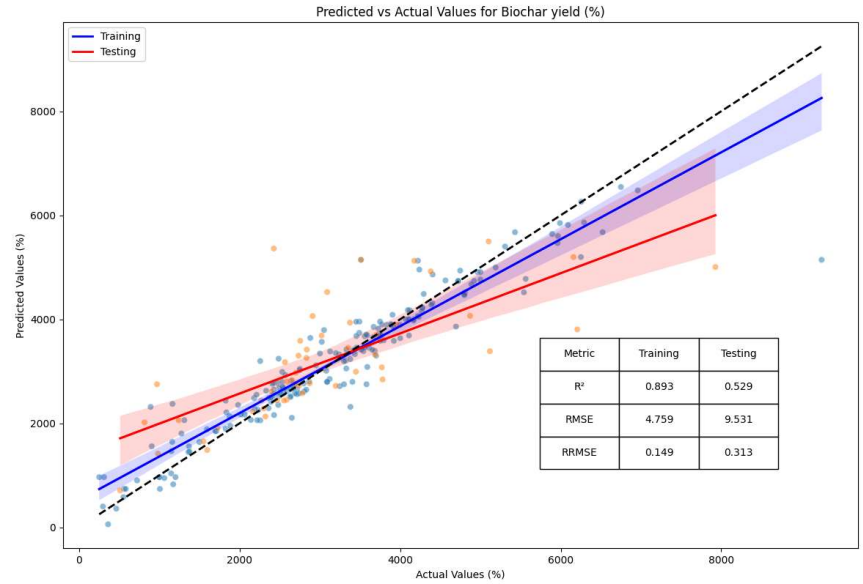
Actual Values (%)



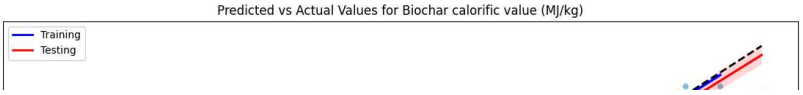
Metric	Training	Testing
R^2	0.940	0.702
RMSE	2.434	5.096
RRMSE	0.157	0.342

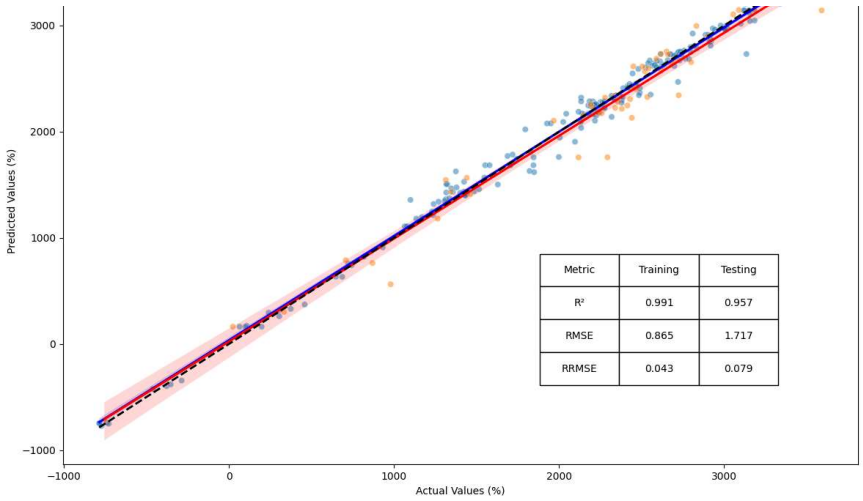


Metric	Training	Testing
R^2	0.986	0.960
RMSE	2.205	4.146
RRMSE	0.068	0.108

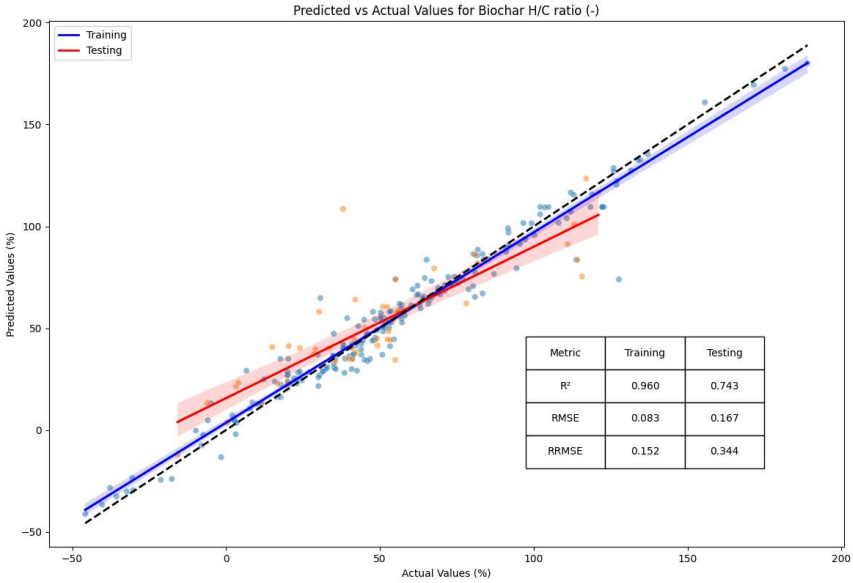


Metric	Training	Testing
R^2	0.893	0.529
RMSE	4.759	9.531
RRMSE	0.149	0.313

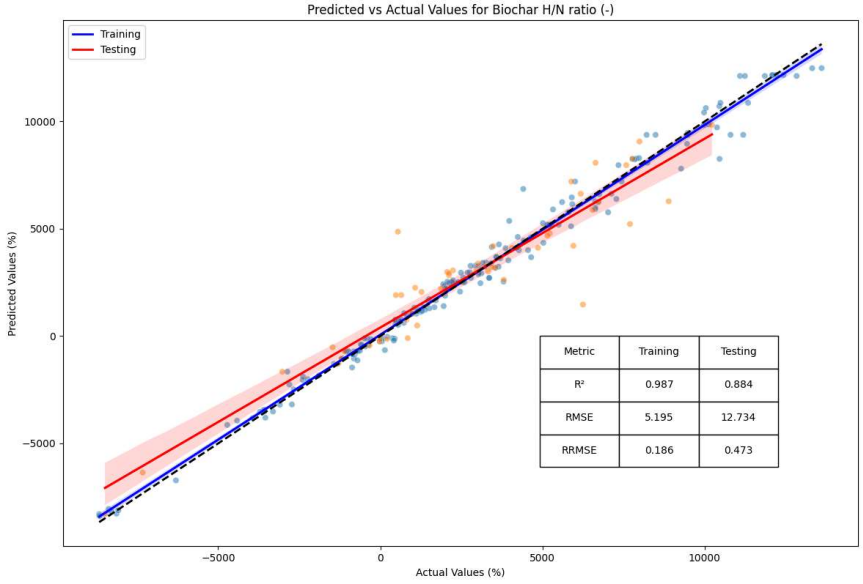




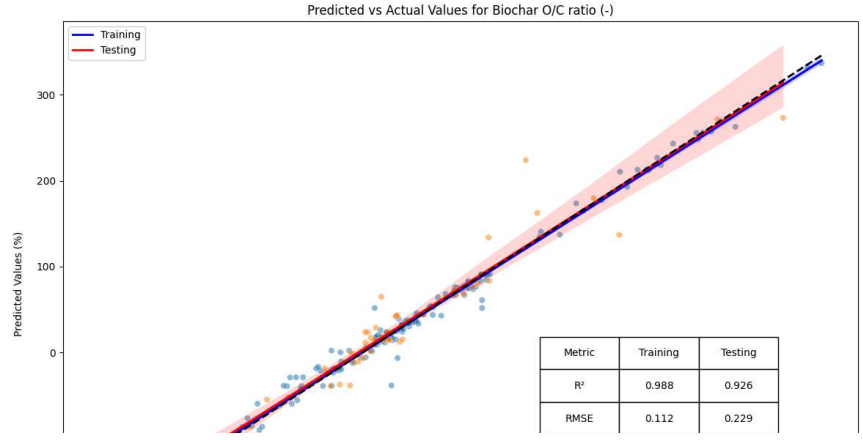
Predicted vs Actual Values for Biochar H/C ratio (-)

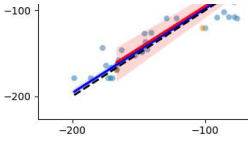


Predicted vs Actual Values for Biochar H/N ratio (-)



Predicted vs Actual Values for Biochar O/C ratio (-)

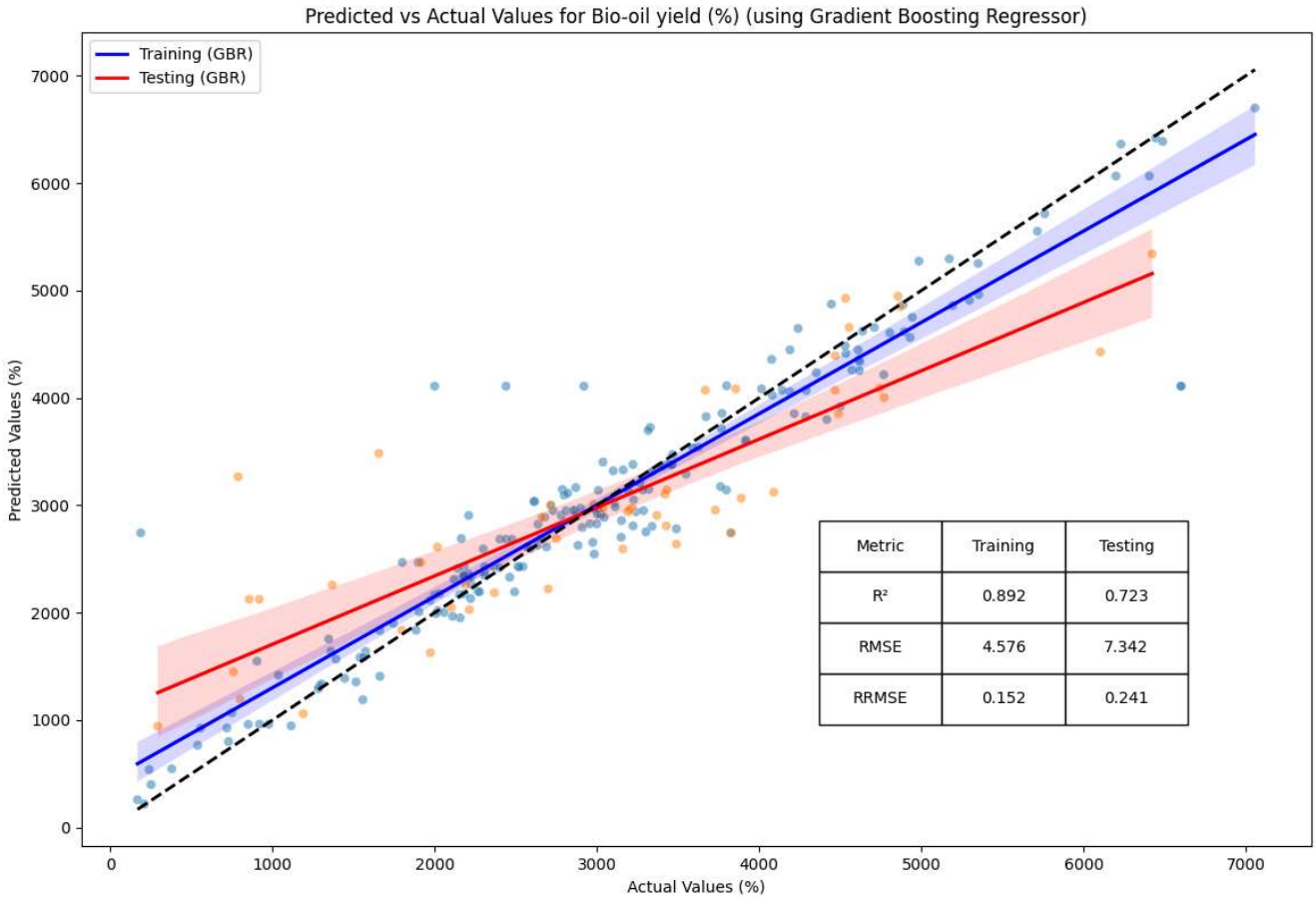


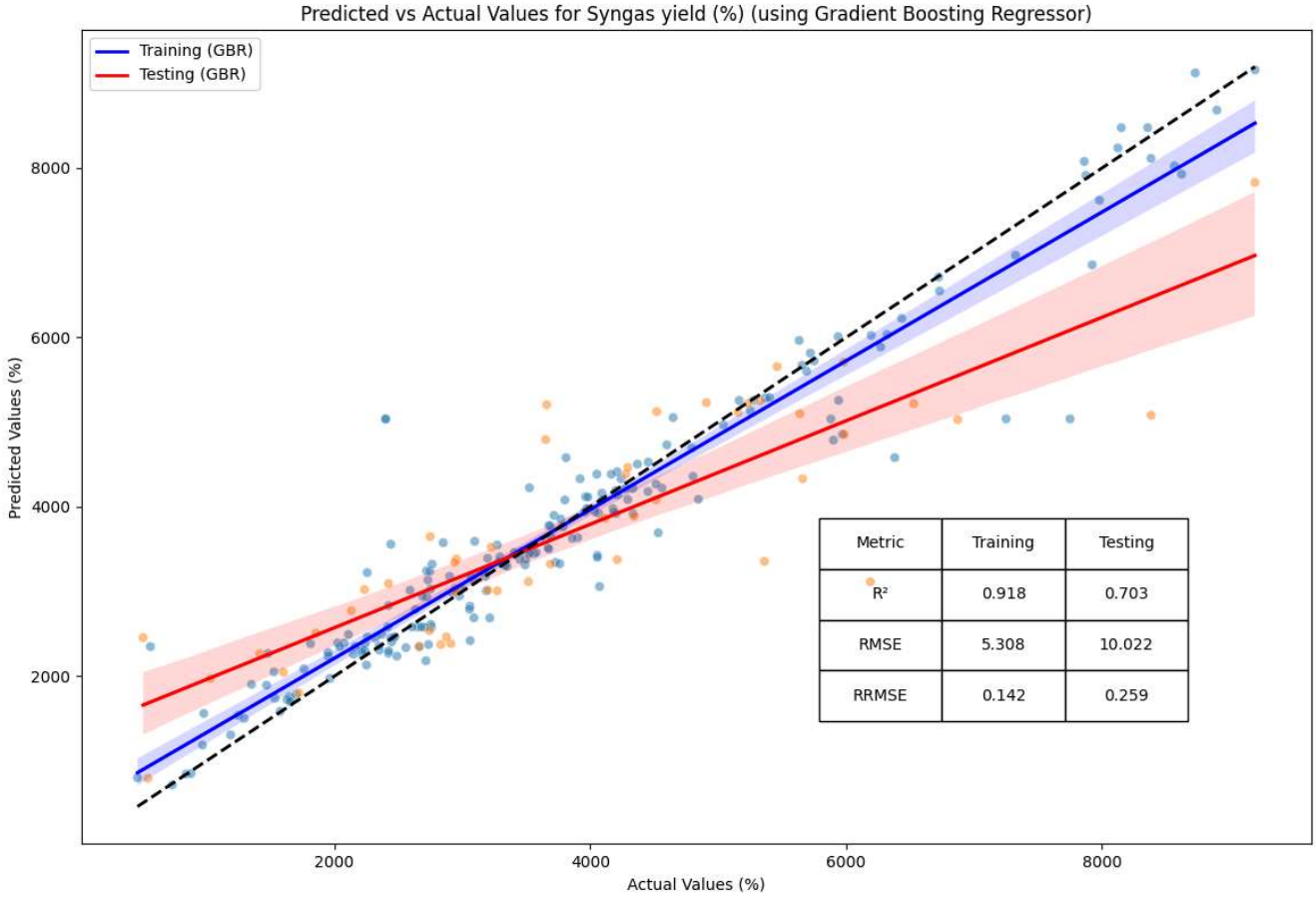


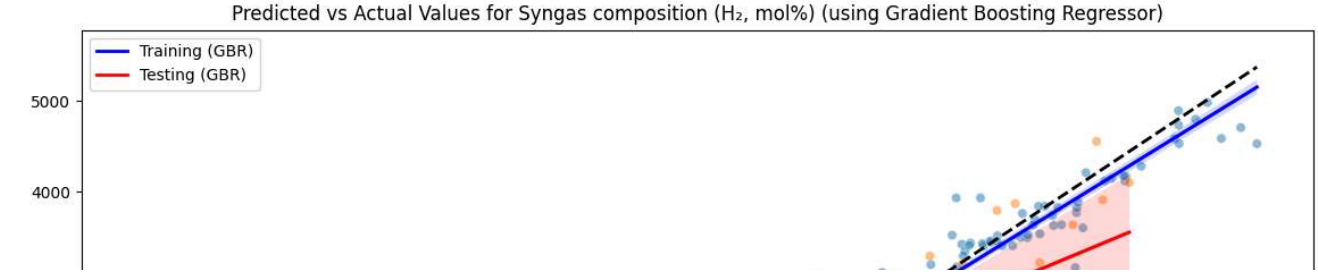
RRMSE	0.590	0.751

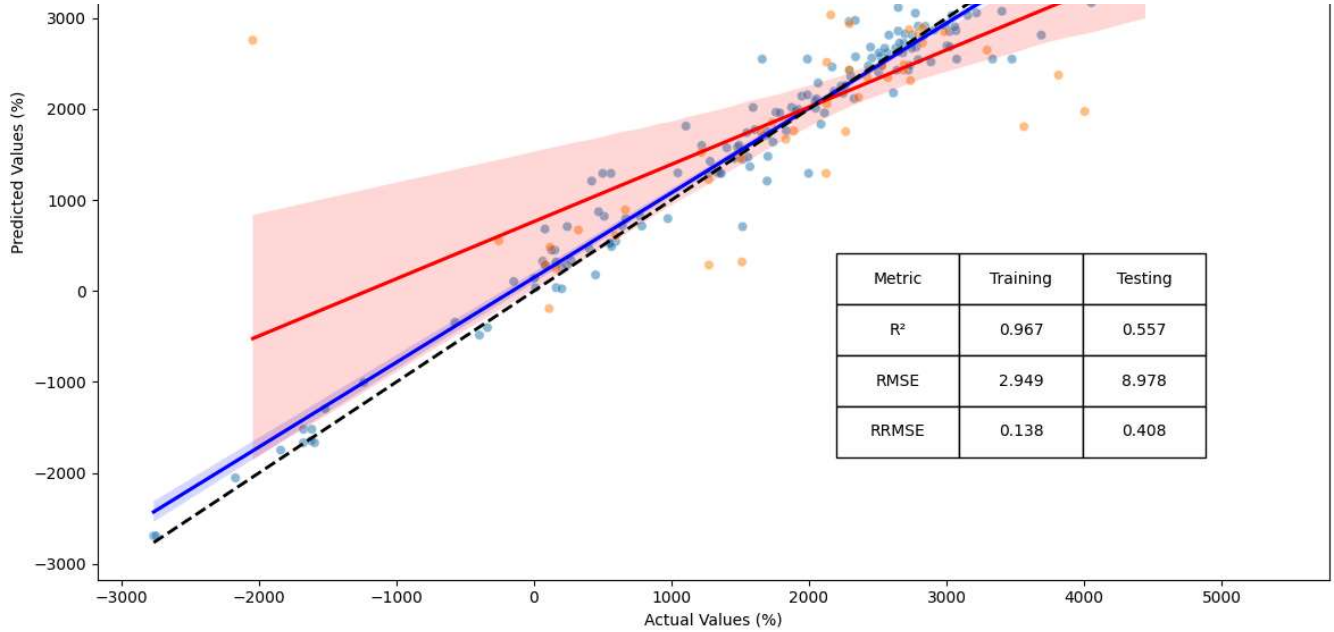
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error

def plot_predicted_vs_actual_multi(df, feature_columns, target_columns, output_file=None):
    num_cols = len(target_columns)
    fig, axs = plt.subplots(num_cols, 1, figsize=(12, 8*num_cols))

    for idx, target_column in enumerate(target_columns):
        # Assign the feature columns to X and the target column to y
        X = df[feature_columns].values
        y = df[target_column].values

        # Split the dataset 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)

        # Standardize the features
        scaler = StandardScaler()
        X_train = scaler.fit_transform(X_train)
        X_test = scaler.transform(X_test)

        # Create and train the Gradient Boosting model
        gb = GradientBoostingRegressor(n_estimators=100, random_state=42)
        gb.fit(X_train, y_train)

        # Make predictions
        y_train_pred = gb.predict(X_train)
```

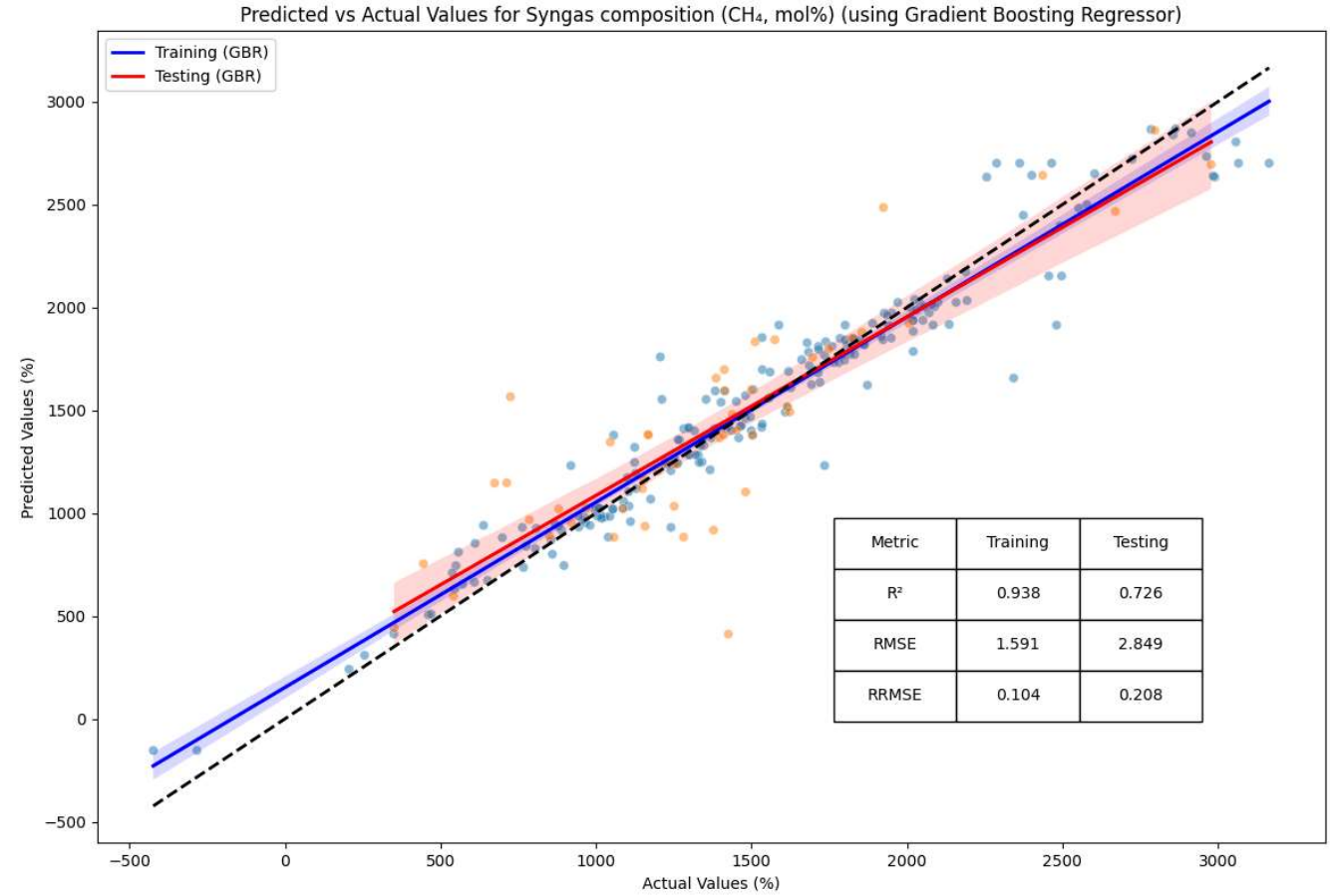
```
            y_test_pred = gb.predict(X_test)

            # Calculate R²
            r2_train = r2_score(y_train, y_train_pred)
            r2_test = r2_score(y_test, y_test_pred)

            # Calculate RMSE
            rmse_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
            rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))

            # Calculate RRMSE
            rrmse_train = rmse_train / np.mean(y_train)
            rrmse_test = rmse_test / np.mean(y_test)

            # Convert actual and predicted values to percentages for plotting
            y_train_percentage = y_train * 100
            y_train_pred_percentage = y_train_pred * 100
            y_test_percentage = y_test * 100
            y_test_pred_percentage = y_test_pred * 100

            # Combine data into a DataFrame for plotting with seaborn
            df_plot = pd.DataFrame({
                'Actual': np.concatenate((y_train_percentage, y_test_percentage)),
                'Predicted': np.concatenate((y_train_pred_percentage, y_test_pred_percentage)),
                'Dataset': ['Training'] * len(y_train_percentage) + ['Testing'] * len(y_test_percentage)
            })

            # Plot on the respective subplot
            ax = axs[idx]
            sns.scatterplot(data=df_plot, x='Actual', y='Predicted', hue='Dataset', alpha=0.5, ax=ax)

            # Training data trend line and confidence interval
            sns.regplot(data=df_plot[df_plot['Dataset'] == 'Training'], x='Actual', y='Predicted', scatter=False, color='blue', ax=ax)

            # Testing data trend line and confidence interval
            sns.regplot(data=df_plot[df_plot['Dataset'] == 'Testing'], x='Actual', y='Predicted', scatter=False, color='red', ax=ax)

            # Add diagonal line
            ax.plot([min(y) * 100, max(y) * 100], [min(y) * 100, max(y) * 100], 'k--', lw=2)

            # Add labels and title with target column
            ax.set_xlabel('Actual Values (%)')
            ax.set_ylabel('Predicted Values (%)')
            ax.set_title(f'Predicted vs Actual Values for {target_column} (using Gradient Boosting Regressor)')

            # Create custom legend with both training and testing labels
            custom_lines = [plt.Line2D([0], [0], color='blue', lw=2),
                            plt.Line2D([0], [0], color='red', lw=2)]
            ax.legend(custom_lines, ['Training (GBR)', 'Testing (GBR)'], loc='upper left')

            # Add table with performance metrics
            table_data = [['R²', f'{r2_train:.3f}', f'{r2_test:.3f}'],
                          ['RMSE', f'{rmse_train:.3f}', f'{rmse_test:.3f}'],
                          ['RRMSE', f'{rrmse_train:.3f}', f'{rrmse_test:.3f}']]
            table = ax.table(cellText=table_data,
                             colLabels=['Metric', 'Training', 'Testing'],
                             cellLoc='center',
                             loc='bottom right',  # Adjusted to bottom right
                             bbox=[0.6, 0.15, 0.3, 0.25])  # Adjust bbox to control position and size

            # Adjust table font size and style
            table.auto_set_font_size(False)
            table.set_fontsize(10)

        plt.tight_layout()

        # Save the figure to a file if output_file is provided
        if output_file:
            plt.savefig(output_file)

        plt.show()

    # Example usage:
    # Assuming `df`, `feature_columns`, and `target_columns` are defined appropriately
    # Also assuming `output_file` is the file path where you want to save the plot
    output_file = 'GBR_predicted_vs_actual_multi_gb.png'
    plot_predicted_vs_actual_multi(df, feature_columns, target_columns, output_file)
```

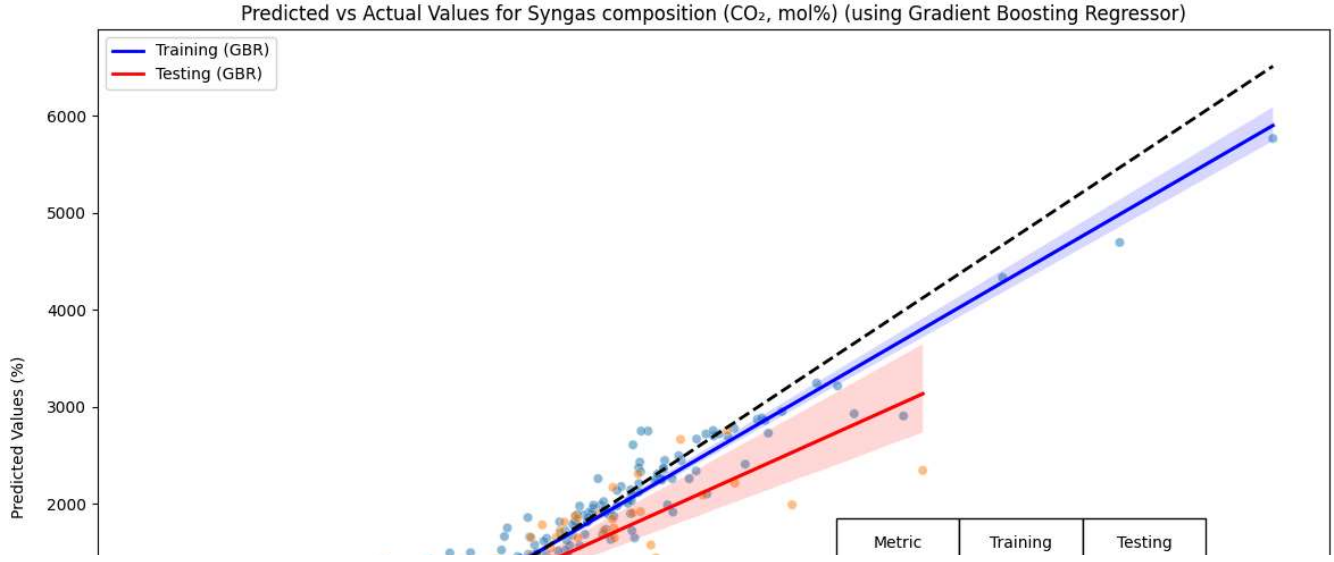

Predicted vs Actual Values for Bio-oil yield (%) (using Gradient Boosting Regressor)
Training (GBR)
Testing (GBR)
Predicted Values (%)
Actual Values (%)
Metric | Training | Testing
R² | 0.892 | 0.723
RMSE | 4.576 | 7.342
RRMSE | 0.152 | 0.241

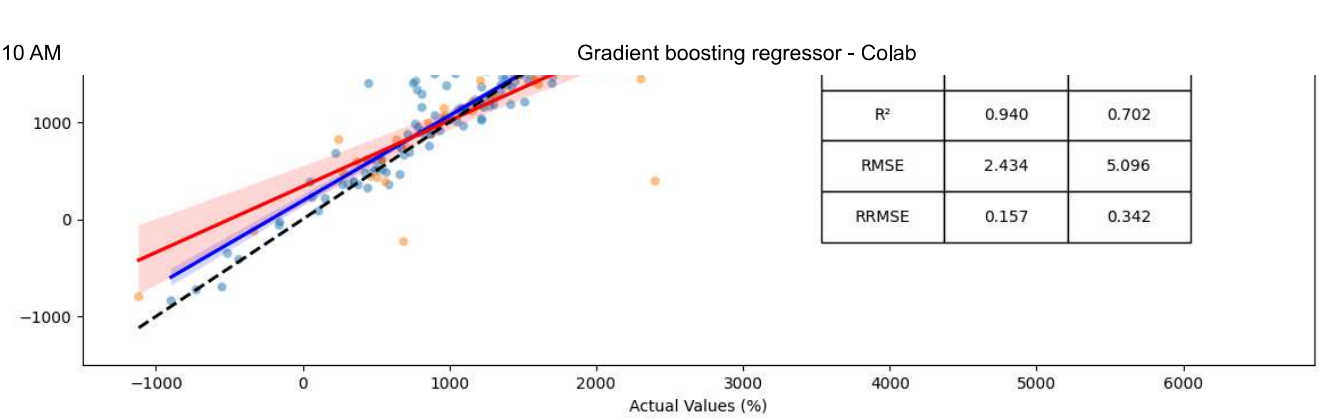

Predicted vs Actual Values for Syngas yield (%) (using Gradient Boosting Regressor)
Training (GBR)
Testing (GBR)
Predicted Values (%)
Actual Values (%)
Metric | Training | Testing
R² | 0.918 | 0.703
RMSE | 5.308 | 10.022
RRMSE | 0.142 | 0.259


Predicted vs Actual Values for Syngas composition (H₂, mol%) (using Gradient Boosting Regressor)
Training (GBR)
Testing (GBR)

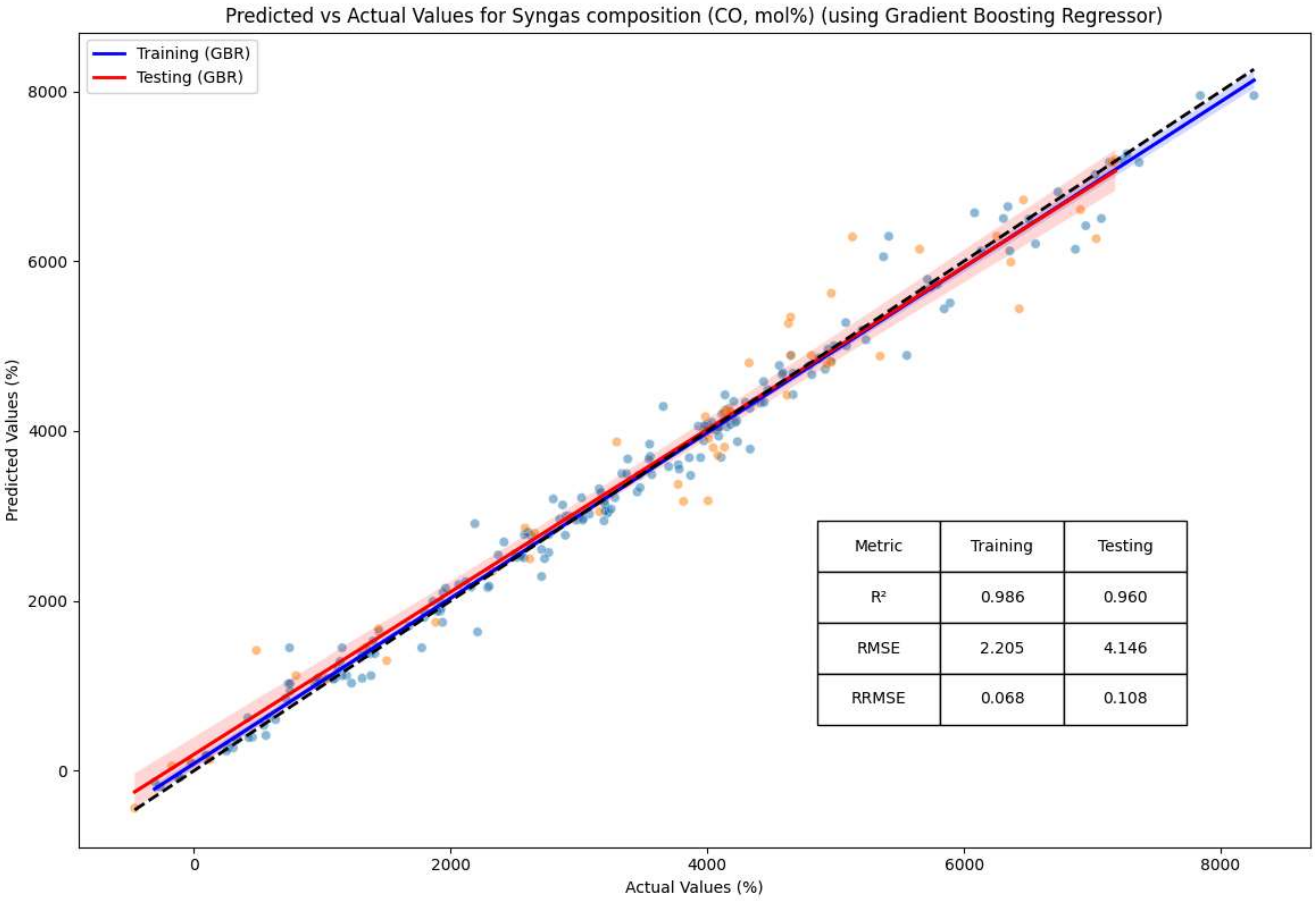

Predicted Values (%)
Actual Values (%)
Metric | Training | Testing
R² | 0.967 | 0.557
RMSE | 2.949 | 8.978
RRMSE | 0.138 | 0.408


Predicted vs Actual Values for Syngas composition (CH₄, mol%) (using Gradient Boosting Regressor)
Training (GBR)
Testing (GBR)
Predicted Values (%)
Actual Values (%)
Metric | Training | Testing
R² | 0.938 | 0.726
RMSE | 1.591 | 2.849
RRMSE | 0.104 | 0.208

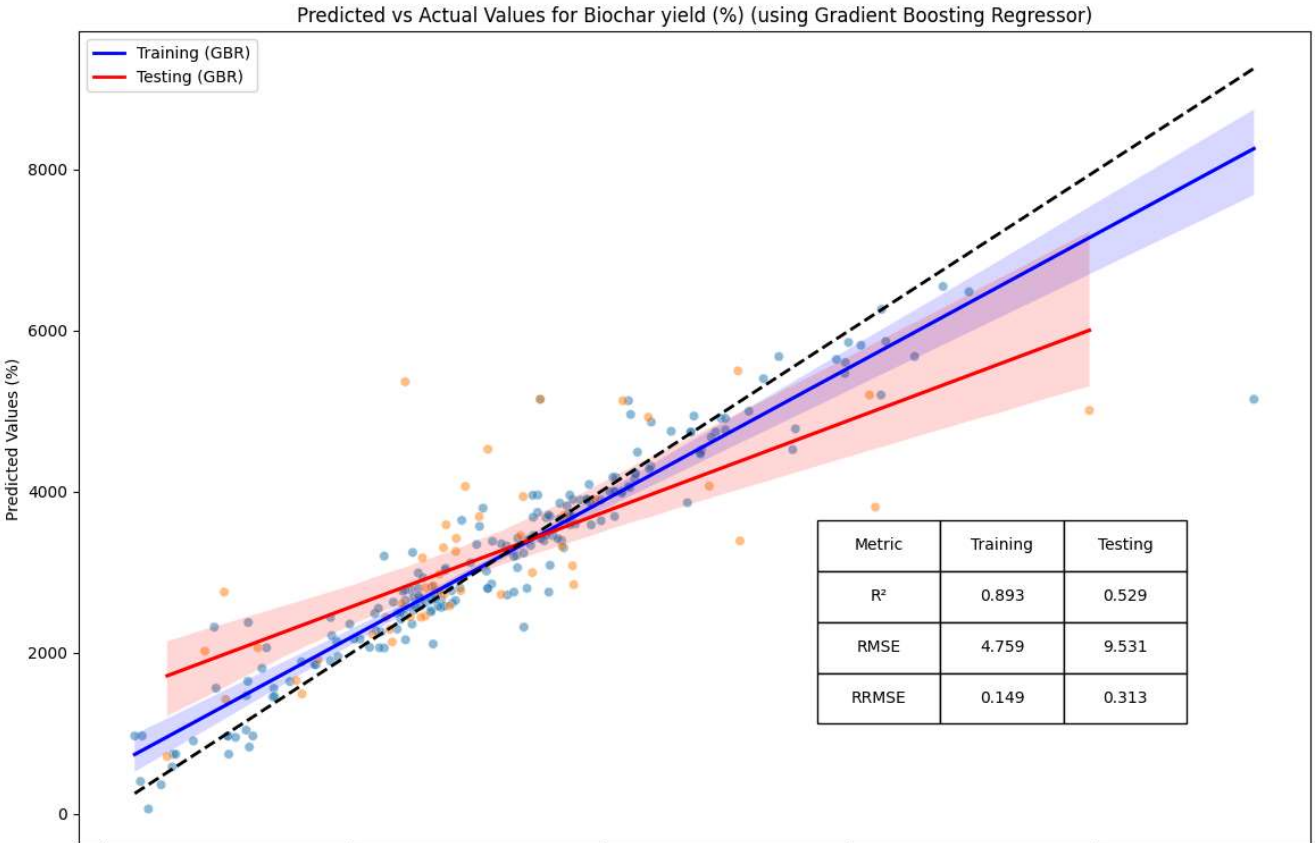

Predicted vs Actual Values for Syngas composition (CO₂, mol%) (using Gradient Boosting Regressor)
Training (GBR)
Testing (GBR)
Predicted Values (%)
Metric | Training | Testing



R²	0.940	0.702
RMSE	2.434	5.096
RRMSE	0.157	0.342



Metric	Training	Testing
R²	0.986	0.960
RMSE	2.205	4.146
RRMSE	0.068	0.108



Metric	Training	Testing
R²	0.893	0.529
RMSE	4.759	9.531
RRMSE	0.149	0.313

Start coding or generate with AI.

Import Necessary Libraries:

```
pip install shap scikit-learn pandas numpy
```

```
Collecting shap
  Downloading shap-0.46.0-cp310-cp310-manylinux_2_12_x86_64.manylinux2010_x86_64.manylinux_2_17_x86_64.manylinux2014_x86_64.whl (546
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 540.1/540.1 kB 3.3 MB/s eta 0:00:00
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.10/dist-packages (1.2.2)
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (2.0.3)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (1.25.2)
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-packages (from shap) (1.11.4)
Requirement already satisfied: tqdm>=4.27.0 in /usr/local/lib/python3.10/dist-packages (from shap) (4.66.4)
Requirement already satisfied: packaging>20.9 in /usr/local/lib/python3.10/dist-packages (from shap) (24.1)
Collecting slicer==0.0.8 (from shap)
  Downloading slicer-0.0.8-py3-none-any.whl (15 kB)
Requirement already satisfied: numba in /usr/local/lib/python3.10/dist-packages (from shap) (0.58.1)
Requirement already satisfied: cloudpickle in /usr/local/lib/python3.10/dist-packages (from shap) (2.2.1)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from scikit-learn) (1.4.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn) (3.5.0)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packages (from pandas) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas) (2023.4)
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-packages (from pandas) (2024.1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.2->pandas) (1.16.0)
Requirement already satisfied: llvmlite<0.42,>=0.41.0dev0 in /usr/local/lib/python3.10/dist-packages (from numba->shap) (0.41.1)
Installing collected packages: slicer, shap
Successfully installed shap-0.46.0 slicer-0.0.8
```

```
import pandas as pd
import shap
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
```

Load Your Data: Load and prepare your data as described earlier.

Train Models and Generate SHAP Values: Train a model for each target column and compute SHAP values.

```
import pandas as pd
import shap
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
import os
from PIL import Image

# Load your dataset from Excel
file_path = 'cleaned_imputed_data.xlsx'
data = pd.read_excel(file_path)

# Define feature columns and target columns
feature_columns = [
    'Carbon content (wt%)',
    'Hydrogen content (wt%)',
    'Nitrogen content (wt%)',
    'Oxygen content (wt%)',
    'Sulfur content (wt%)',
    'Volatile matter (wt%)',
    'Fixed carbon (wt%)',
    'Ash content (wt%)',
    'Reaction temperature (°C)',
    'Microwave power (W)',
    'Reaction time (min)',
    'Microwave absorber percentage (%)',
    'Dielectric constant of absorber (ε’)',
    'Dielectric loss factor of absorber (ε“)'
]

target_columns = [
    'Bio-oil yield (%)',
    'Syngas yield (%)',
    'Syngas composition (H₂, mol%)',
```

```
        'Syngas composition (CH₄, mol%)',
        'Syngas composition (CO₂, mol%)',
        'Syngas composition (CO, mol%)',
        'Biochar yield (%)',
        'Biochar calorific value (MJ/kg)',
        'Biochar H/C ratio (-)',
        'Biochar H/N ratio (-)',
        'Biochar O/C ratio (-)'
    ]

    # Extract features and targets
    X = data[feature_columns]
    y = data[target_columns]

    # Directory to save plots
    plot_dir = 'shap_plots/'
    if not os.path.exists(plot_dir):
        os.makedirs(plot_dir)

    def sanitize_filename(filename):
        return ''.join(c for c in filename if c.isalnum() or c in (' ', '_')).rstrip()

    # Generate and save SHAP summary plots
    for target_column in target_columns:
        print(f"Analyzing target: {target_column}")

        # Extract the target values
        y_target = y[target_column]

        # Split the data into train and test sets
        X_train, X_test, y_train, y_test = train_test_split(X, y_target, test_size=0.2, random_state=42)

        # Train a Random Forest regressor
        model = RandomForestRegressor(n_estimators=100, random_state=42)
        model.fit(X_train, y_train)

        # Create a SHAP explainer
        explainer = shap.TreeExplainer(model)

        # Calculate SHAP values
        shap_values = explainer.shap_values(X_test)

        # Save the summary plot to a file
        try:
            plt.figure(figsize=(10, 5))
            shap.summary_plot(shap_values, X_test, show=False)
            plt.title(f"SHAP Summary Plot for {target_column}")

            # Sanitize the filename
            safe_target_column = sanitize_filename(target_column)
            filename = os.path.join(plot_dir, f'shap_summary_{safe_target_column}.png')
            plt.savefig(filename)
            plt.close()
        except Exception as e:
            print(f"Failed to save plot for {target_column}: {e}")

    # Combine saved SHAP plot images into one large image
    plot_files = sorted([os.path.join(plot_dir, f) for f in os.listdir(plot_dir) if f.endswith('.png')])

    # Open images
    images = [Image.open(f) for f in plot_files]

    # Assuming all images are the same size, get dimensions
    width, height = images[0].size

    # Create a new image with a height that can fit all plots
    total_height = height * len(images)
    combined_image = Image.new('RGB', (width, total_height))

    # Paste each image into the combined image
    y_offset = 0
    for img in images:
        combined_image.paste(img, (0, y_offset))
        y_offset += height

    # Save and show the combined image
    combined_image_path = 'combined_shap_plots.png'
    combined_image.save(combined_image_path)

    # Display the combined image
    combined_image.show()
```

```
Analyzing target: Bio-oil yield (%)
Analyzing target: Syngas yield (%)
Analyzing target: Syngas composition (H₂, mol%)
Analyzing target: Syngas composition (CH₄, mol%)
Analyzing target: Syngas composition (CO₂, mol%)
Analyzing target: Syngas composition (CO, mol%)
Analyzing target: Biochar yield (%)
Analyzing target: Biochar calorific value (MJ/kg)
Analyzing target: Biochar H/C ratio (-)
Analyzing target: Biochar H/N ratio (-)
Analyzing target: Biochar O/C ratio (-)
```