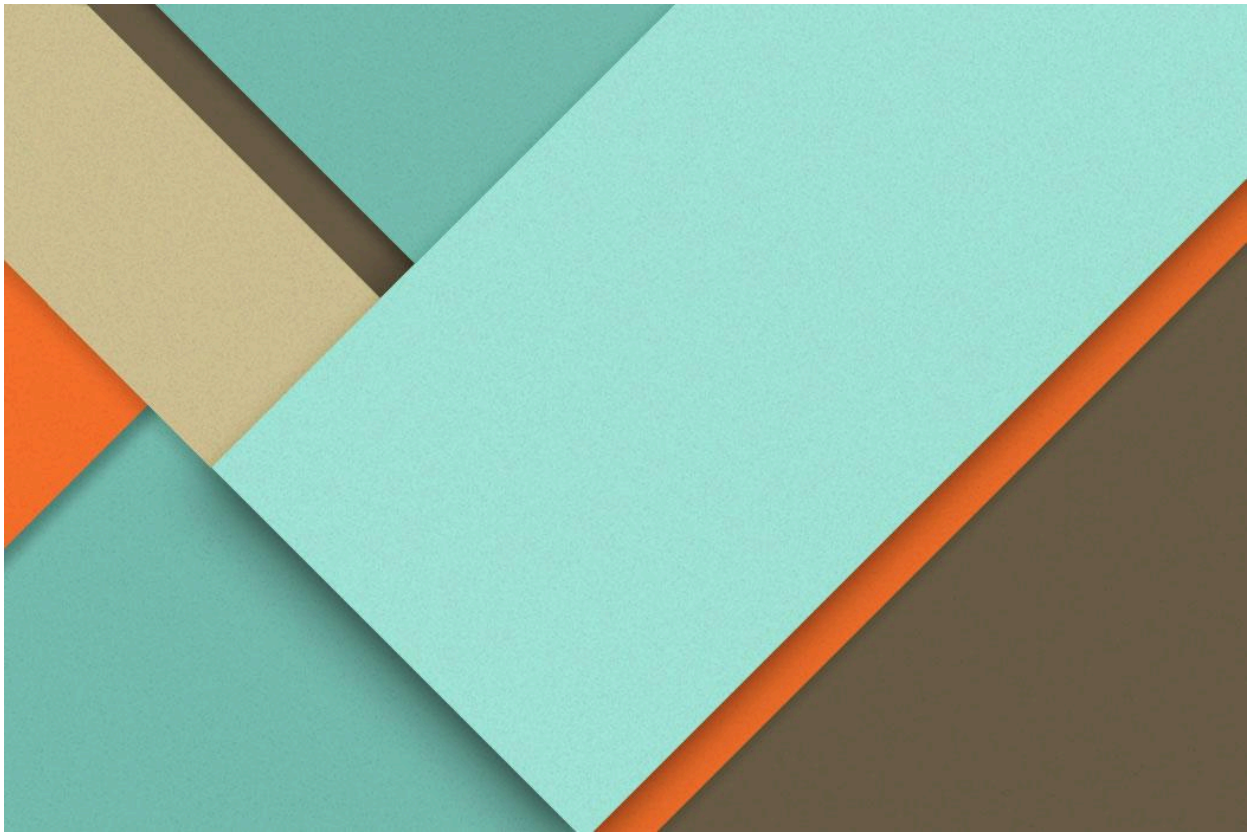


CSTP2301 Final Project Final

10.04.2024

ML

Jia Xi Lin
Muochu Hu

Goals

1. Picking IDs 542236, 67321, 549295, 41108, 54982 to do the calculations.
2. Using regression models to train and test the data set.
3. Comparing the actual value of y and predicted value of y and calculate the accuracy of them, try to improve the accuracy and fill the results in the Excel files.

Step 1: Read the Dataset

Begin by importing necessary libraries and loading the dataset. Use pandas to read the CSV file containing the data. This step is crucial as it sets up your data for preprocessing and analysis.

```
import pandas as pd
data = pd.read_csv('Sub_Oil_VLCC_Monthly.csv')
data
```

Step 2: Data Preprocessing

Prepare the dataset for the model. This involves splitting the dataset into features (X) and target variables (y). Since the model requires numerical inputs, ensure that the data is cleaned and appropriately formatted.

- Exclude the last row from the dataset for features (X).
- Define target columns for various scenarios (good, mid, and bad results).
- Exclude the first row from target columns to align with X.

To generate X and y, we can consider the entire table as X and copy the target column in a vector as y. Remember that the label of sample t in X is in row t+1 in y.

```
#Splitting the data into training and testing data

X = data[:-1]  #whole data exclude the last row as X

#target columns
good_result1_target_column = data['542236']
good_result2_target_column = data['67321']
mid_result_target_column = data['549295']
bad_result1_target_column = data['41108']
bad_result2_target_column = data['541982']

#target column exclude the first row as y
good1_y = good_result1_target_column[1:]
good2_y = good_result2_target_column[1:]
mid_y = mid_result_target_column[1:]
bad1_y = bad_result1_target_column[1:]
bad2_y = bad_result2_target_column[1:]
```

Step 3: Splitting the Data into Training and Testing Sets

It's essential to evaluate the model's performance on unseen data. Split the dataset into a training set used for learning and a testing set for evaluation. We aim to assess the model's accuracy over the last three years available in the dataset. To achieve this, consider training the model on all samples from the beginning up to n-36, and then test it on the last 36 samples (n is the total number of samples excluding the last one that doesn't include a label)

```
n = len(X)-36 #number of data points for training and testing

X_train = X.iloc[:n, :] #training data
X_test = X.iloc[n:, :] #testing data

#good1 target column training and testing data
good1_y_train = good1_y.iloc[:n]
good1_y_test = good1_y.iloc[n:]

#good2 target column training and testing data
good2_y_train = good2_y.iloc[:n]
good2_y_test = good2_y.iloc[n:]

#mid target column training and testing data
mid_y_train = mid_y.iloc[:n]
mid_y_test = mid_y.iloc[n:]

#bad1 target column training and testing data
bad1_y_train = bad1_y.iloc[:n]
bad1_y_test = bad1_y.iloc[n:]

#bad2 target column training and testing data
bad2_y_train = bad2_y.iloc[:n]
bad2_y_test = bad2_y.iloc[n:]
```

Making sure of the training data are the data from 19910101 to 20201201, and the testing data are from 20210101 to 20231201 by printing them out.

```
print("X_train_Head: ", X_train.iloc[0,0])
print("X_train_Tail: ", X_train.iloc[-1,0])
print("X_test_Head: ", X_test.iloc[0,0])
print("X_test_Tail: ", X_test.iloc[-1,0])
```

✓ 2m 4.9s

```
X_train_Head:  19910101
X_train_Tail:  20201201
X_test_Head:  20210101
X_test_Tail:  20231201
```

Step 4: Feature Elimination

We engaged in a critical phase of our machine learning project by implementing feature elimination, aiming to enhance model simplicity and performance. Using a DecisionTreeRegressor as our estimator, we employed **Recursive Feature Elimination (RFE)** to methodically select the top 5 features for each of our target IDs.

This selection process involved iteratively fitting the **DecisionTreeRegressor**, each time removing the least important feature until only the desired number of top features remained. Consequently, for each target variable, we derived a transformed training and testing dataset, retaining only the most influential features.

This streamlined approach not only reduces model complexity but also potentially improves model accuracy by focusing on the most relevant predictors.

```
#Feature Elimination
# Using DecisionTreeRegressor for feature elimination
estimator = DecisionTreeRegressor()
selector = RFE(estimator, n_features_to_select=5, step=1)
good1_selector = selector.fit(X_train, good1_y_train)
good2_selector = selector.fit(X_train, good2_y_train)
mid_selector = selector.fit(X_train, mid_y_train)
bad1_selector = selector.fit(X_train, bad1_y_train)
bad2_selector = selector.fit(X_train, bad2_y_train)

#Training and Testing data after feature elimination
good1_X_train = good1_selector.transform(X_train)
good1_X_test = good1_selector.transform(X_test)
good2_X_train = good2_selector.transform(X_train)
good2_X_test = good2_selector.transform(X_test)
mid_X_train = mid_selector.transform(X_train)
mid_X_test = mid_selector.transform(X_test)
bad1_X_train = bad1_selector.transform(X_train)
bad1_X_test = bad1_selector.transform(X_test)
bad2_X_train = bad2_selector.transform(X_train)
bad2_X_test = bad2_selector.transform(X_test)
```

Step 5: Normalization

We proceeded to the normalization stage to ensure that our dataset's feature values had a uniform scale, an essential step for optimizing the performance of our machine learning models. Initially, we extracted the 'time' feature, representing dates, from our dataset and excluding the last row to align with our dataset's structure.

Following this, we normalized the remaining features (excluding the first 'time' column and the last row) using Min-Max scaling, which rescales the feature values to a range of 0 to 1. This normalization technique is particularly effective in avoiding distortions caused by extreme values or outliers.

After normalization, we reinserted the 'time' column back into our dataset to maintain its original structure. We then split the normalized dataset into training and testing sets based on a predefined threshold 'n'.

Subsequently, we applied the previously determined feature selection criteria (from Step 4) to this normalized dataset to obtain the final sets of training and testing data for each target ID. This rigorous preparation of our data, encompassing both feature elimination and normalization, lays a solid foundation for the subsequent training of our predictive models.

```
#Normalization
#get the first column of the data
time = data.iloc[:-1, 0].astype(int)

#Normalize the data
scaledX = data.iloc[:-1, 1:]  # Excluding the last row and first column for dates
scaledX = (scaledX - scaledX.min()) / (scaledX.max() - scaledX.min())

# Assign the 'date' column back as integers
scaledX.insert(0,'Unnamed: 0' ,time)

scaledX_train = scaledX.iloc[:n, :] #scaled training data
scaledX_test = scaledX.iloc[n:, :] #scaled testing data
# print(scaledX_train)
# print(scaledX_test)

#Training and Testing data after feature elimination and normalization
good1_scaledX_train = good1_selector.transform(scaledX_train)
good1_scaledX_test = good1_selector.transform(scaledX_test)
good2_scaledX_train = good2_selector.transform(scaledX_train)
good2_scaledX_test = good2_selector.transform(scaledX_test)
mid_scaledX_train = mid_selector.transform(scaledX_train)
mid_scaledX_test = mid_selector.transform(scaledX_test)
bad1_scaledX_train = bad1_selector.transform(scaledX_train)
bad1_scaledX_test = bad1_selector.transform(scaledX_test)
bad2_scaledX_train = bad2_selector.transform(scaledX_train)
bad2_scaledX_test = bad2_selector.transform(scaledX_test)
```

Step 6: Lag Feature Creation and Data Preparation

We concatenate two rows of X to generate lag 1, and the output of this sample will be the target value for the upcoming month. For implementation, we delete the two first rows of y and the two last rows of X to have the output in the same row as the input.

```
lagged_features = []

# Loop over the dataset starting from the second row to the second last row
for i in range(1, len(data) - 1):  # Stop before the last row to exclude last two from features
    # Concatenate the current row and the previous row
    current_row = data.iloc[i, :].tolist()
    previous_row = data.iloc[i - 1, :].tolist()
    lagged_row = previous_row + current_row
    lagged_features.append(lagged_row)

# Convert the list to a DataFrame
X_lag1 = pd.DataFrame(lagged_features)

lag_n = len(X_lag1)-36 #number of data points for training and testing

X_lag1_train = X_lag1.iloc[:lag_n, :] #training data
X_lag1_test = X_lag1.iloc[lag_n:, :] #testing data

#Create lagged y dataset
#exclude the first two rows
good1_y_lag1 = good_result1_target_column[2:]
good2_y_lag1 = good_result2_target_column[2:]
mid_y_lag1 = mid_result_target_column[2:]
bad1_y_lag1 = bad_result1_target_column[2:]
bad2_y_lag1 = bad_result2_target_column[2:]

#Splitting the lagged data into training and testing data
X_lag1_train = X_lag1.iloc[:lag_n, :] #training data
X_lag1_test = X_lag1.iloc[lag_n:, :] #testing data

#good1 target column training and testing data
good1_y_lag1_train = good1_y_lag1.iloc[:lag_n]
good1_y_lag1_test = good1_y_lag1.iloc[lag_n:]

#good2 target column training and testing data
good2_y_lag1_train = good2_y_lag1.iloc[:lag_n]
good2_y_lag1_test = good2_y_lag1.iloc[lag_n:]

#mid target column training and testing data
mid_y_lag1_train = mid_y_lag1.iloc[:lag_n]
mid_y_lag1_test = mid_y_lag1.iloc[lag_n:]

#bad1 target column training and testing data
bad1_y_lag1_train = bad1_y_lag1.iloc[:lag_n]
bad1_y_lag1_test = bad1_y_lag1.iloc[lag_n:]

#bad2 target column training and testing data
bad2_y_lag1_train = bad2_y_lag1.iloc[:lag_n]
bad2_y_lag1_test = bad2_y_lag1.iloc[lag_n:]
```

We also apply the feature elimination and normalization logic for the lagged data set, same as Step 4 and Step5.

```
#Feature Elimination
# Using DecisionTreeRegressor for feature elimination for lagged data
estimator = DecisionTreeRegressor()
selector = RFE(estimator, n_features_to_select=5, step=1)
good1_lag1_selector = selector.fit(X_lag1_train, good1_y_lag1_train)
good2_lag1_selector = selector.fit(X_lag1_train, good2_y_lag1_train)
mid_lag1_selector = selector.fit(X_lag1_train, mid_y_lag1_train)
bad1_lag1_selector = selector.fit(X_lag1_train, bad1_y_lag1_train)
bad2_lag1_selector = selector.fit(X_lag1_train, bad2_y_lag1_train)

#Training and Testing data after feature elimination
good1_lag1_X_train = good1_lag1_selector.transform(X_lag1_train)
good1_lag1_X_test = good1_lag1_selector.transform(X_lag1_test)
good2_lag1_X_train = good2_lag1_selector.transform(X_lag1_train)
good2_lag1_X_test = good2_lag1_selector.transform(X_lag1_test)
mid_lag1_X_train = mid_lag1_selector.transform(X_lag1_train)
mid_lag1_X_test = mid_lag1_selector.transform(X_lag1_test)
bad1_lag1_X_train = bad1_lag1_selector.transform(X_lag1_train)
bad1_lag1_X_test = bad1_lag1_selector.transform(X_lag1_test)
bad2_lag1_X_train = bad2_lag1_selector.transform(X_lag1_train)
bad2_lag1_X_test = bad2_lag1_selector.transform(X_lag1_test)

#Normalization for lagged data
scaled_lagged_features = []

# Loop over the dataset starting from the second row to the second last row
for i in range(1, len(data) - 1):  # Stop before the last row to exclude last two from features
    # Concatenate the current row and the previous row
    current_row = scaledX.iloc[i, :].tolist()
    previous_row = scaledX.iloc[i - 1, :].tolist()
    lagged_row = previous_row + current_row
    scaled_lagged_features.append(lagged_row)

# Convert the list to a DataFrame
scaledX_lag1 = pd.DataFrame(scaled_lagged_features)

scaledX_lag1_train = scaledX_lag1.iloc[:lag_n, :] #training data
scaledX_lag1_test = scaledX_lag1.iloc[lag_n:, :] #testing data
#print(scaledX_lag1_train)
#print(scaledX_lag1_test)

#Training and Testing data after feature elimination and normalization for lagged data
good1_lag1_scaledX_train = good1_lag1_selector.transform(scaledX_lag1_train)
good1_lag1_scaledX_test = good1_lag1_selector.transform(scaledX_lag1_test)
good2_lag1_scaledX_train = good2_lag1_selector.transform(scaledX_lag1_train)
good2_lag1_scaledX_test = good2_lag1_selector.transform(scaledX_lag1_test)
mid_lag1_scaledX_train = mid_lag1_selector.transform(scaledX_lag1_train)
mid_lag1_scaledX_test = mid_lag1_selector.transform(scaledX_lag1_test)
bad1_lag1_scaledX_train = bad1_lag1_selector.transform(scaledX_lag1_train)
bad1_lag1_scaledX_test = bad1_lag1_selector.transform(scaledX_lag1_test)
bad2_lag1_scaledX_train = bad2_lag1_selector.transform(scaledX_lag1_train)
bad2_lag1_scaledX_test = bad2_lag1_selector.transform(scaledX_lag1_test)
```

Conclusion before training the models

In our machine learning project, we have organized and processed our data into two main types: one set using the normal dataset without any lag adjustments (non-lagged dataset), and another incorporating a one-step lag (lagged dataset). Within each of these two main types, we have further subdivided the data into four distinct groups to thoroughly explore different preprocessing strategies and their impact on model performance:

- **Normal Data**: This is the baseline dataset for each type. For the non-lagged dataset, it includes the original features without any modifications. For the lagged dataset, it contains the same original features but extended with one lag, meaning each row includes information from the current and the previous time step.
- **Data After Feature Elimination**: In this version, we apply feature elimination techniques to reduce the dimensionality of the data. This process involves removing less informative or redundant features to improve model efficiency and potentially enhance model accuracy. Feature elimination is applied separately to both the non-lagged and lagged datasets.
- **Data After Normalization**: This dataset undergoes a normalization process, where feature values are scaled typically to a range or distribution that is more suitable for the algorithms in use. Normalization helps in speeding up the learning and convergence of the model. Like feature elimination, normalization is also applied independently to both non-lagged and lagged datasets.
- **Data After Elimination Plus Normalization**: This dataset combines both preprocessing steps—feature elimination followed by normalization. This comprehensive preprocessing aims to leverage the benefits of both reducing dimensionality and scaling features, thereby preparing an optimized dataset for training. Again, this combined approach is implemented for both non-lagged and lagged data sets.

By structuring our data this way, we are able to systematically evaluate and compare the effects of different preprocessing techniques on the performance of our models, both with and without the inclusion of temporal dependencies introduced by lagging. This methodical approach allows us to identify the most effective data preparation pipeline for enhancing model predictions based on our specific dataset and problem context.

Step 6: Training Models and Predictions

We transitioned to the crucial phase of training our predictive models, employing a comprehensive approach to evaluate the performance of various regression techniques under different data preparation strategies.

Our methodology encompassed training and predicting with multiple regressors: **Decision Tree, Linear Regression, Lasso Regression, Support Vector Regression (SVR), Multi-Layer Perceptron (MLP) Regressor, Gradient Boosting Regressor, Random Forest Regressor, and Ridge Regression**.

Each model was trained and tested on four distinct versions of our dataset: **the original dataset, a dataset after feature elimination, a normalized dataset, and a dataset subjected to both feature elimination and normalization**.

This exhaustive exploration and comparison of models and preprocessing strategies are instrumental in selecting the optimal approach for achieving the highest prediction accuracy in our machine learning project.

The normal dataset without any lag adjustments (non-lagged dataset) and the lagged dataset using the same logic, so we are not repeating for the lagged dataset.

```
def model_fit_predict(X_train, X_test, eliminatedX_train, eliminatedX_test, scaledX_train, scaledX_test, eliminatedScaledX_train, eliminatedScaledX_test, y_train):
    predictions = {}  # Initialize an empty dictionary

    # Decision Tree Regressor
    #Noraml Data set
    DT = DecisionTreeRegressor(random_state=42)
    DT.fit(X_train, y_train)
    predictions['DT'] = DT.predict(X_test)

    #After Feature Elimination Data set
    EliminatedDT = DecisionTreeRegressor(random_state=42)
    EliminatedDT.fit(eliminatedX_train, y_train)
    predictions['Eliminated DT'] = EliminatedDT.predict(eliminatedX_test)

    #After Normalization Data set
    ScaledDT = DecisionTreeRegressor(random_state=42)
    ScaledDT.fit(scaledX_train, y_train)
    predictions['Scaled DT'] = ScaledDT.predict(scaledX_test)

    #After Feature Elimination and Normalization Data set
    EliminatedScaledDT = DecisionTreeRegressor(random_state=42)
    EliminatedScaledDT.fit(eliminatedScaledX_train, y_train)
    predictions['Eliminated Scaled DT'] = EliminatedScaledDT.predict(eliminatedScaledX_test)
```

```python
# Linear Regression
#Noraml Data set
LR = LinearRegression()
LR.fit(X_train, y_train)
predictions['LR'] = LR.predict(X_test)

#After Feature Elimination Data set
EliminatedLR = LinearRegression()
EliminatedLR.fit(eliminatedX_train, y_train)
predictions['Eliminated LR'] = EliminatedLR.predict(eliminatedX_test)

#After Normalization Data set
ScaledLR = LinearRegression()
ScaledLR.fit(scaledX_train, y_train)
predictions['Scaled LR'] = ScaledLR.predict(scaledX_test)

#After Feature Elimination and Normalization Data set
EliminatedScaledLR = LinearRegression()
EliminatedScaledLR.fit(eliminatedScaledX_train, y_train)
predictions['Eliminated Scaled LR'] = EliminatedScaledLR.predict(eliminatedScaledX_test)

# Lasso Regression
#Noraml Data set
Lasso_model = Lasso(alpha=0.1)
Lasso_model.fit(X_train, y_train)
predictions['Lasso'] = Lasso_model.predict(X_test)

#After Feature Elimination Data set
EliminatedLasso_model = Lasso(alpha=0.1)
EliminatedLasso_model.fit(eliminatedX_train, y_train)
predictions['Eliminated Lasso'] = EliminatedLasso_model.predict(eliminatedX_test)

#After Normalization Data set
ScaledLasso_model = Lasso(alpha=0.1)
ScaledLasso_model.fit(scaledX_train, y_train)
predictions['Scaled Lasso'] = ScaledLasso_model.predict(scaledX_test)

#After Feature Elimination and Normalization Data set
EliminatedScaledLasso_model = Lasso(alpha=0.1)
EliminatedScaledLasso_model.fit(eliminatedScaledX_train, y_train)
predictions['Eliminated Scaled Lasso'] = EliminatedScaledLasso_model.predict(eliminatedScaledX_test)
```

```python
# Support Vector Regression
#Noraml Data set
SVR_model = SVR()
SVR_model.fit(X_train, y_train)
predictions['SVR'] = SVR_model.predict(X_test)

#After Feature Elimination Data set
EliminatedSVR_model = SVR()
EliminatedSVR_model.fit(eliminatedX_train, y_train)
predictions['Eliminated SVR'] = EliminatedSVR_model.predict(eliminatedX_test)

#After Normalization Data set
ScaledSVR_model = SVR()
ScaledSVR_model.fit(scaledX_train, y_train)
predictions['Scaled SVR'] = ScaledSVR_model.predict(scaledX_test)

#After Feature Elimination and Normalization Data set
EliminatedScaledSVR_model = SVR()
EliminatedScaledSVR_model.fit(eliminatedScaledX_train, y_train)
predictions['Eliminated Scaled SVR'] = EliminatedScaledSVR_model.predict(eliminatedScaledX_test)
```

```python
# Multi-Layer Perceptron Regressor
#Noraml Data set
MLP = MLPRegressor(hidden_layer_sizes=(100, 50), max_iter=1000)
MLP.fit(X_train, y_train)
predictions['MLP'] = MLP.predict(X_test)

#After Feature Elimination Data set
EliminatedMLP = MLPRegressor(hidden_layer_sizes=(100, 50), max_iter=1000)
EliminatedMLP.fit(eliminatedX_train, y_train)
predictions['Eliminated MLP'] = EliminatedMLP.predict(eliminatedX_test)

#After Normalization Data set
ScaledMLP = MLPRegressor(hidden_layer_sizes=(100, 50), max_iter=1000)
ScaledMLP.fit(scaledX_train, y_train)
predictions['Scaled MLP'] = ScaledMLP.predict(scaledX_test)

#After Feature Elimination and Normalization Data set
EliminatedScaledMLP = MLPRegressor(hidden_layer_sizes=(100, 50), max_iter=1000)
EliminatedScaledMLP.fit(eliminatedScaledX_train, y_train)
predictions['Eliminated Scaled MLP'] = EliminatedScaledMLP.predict(eliminatedScaledX_test)

# Gradient Boosting Regressor
#Noraml Data set
GBR = GradientBoostingRegressor(learning_rate=0.1, n_estimators=100, max_depth=3, random_state=42)
GBR.fit(X_train, y_train)
predictions['GBR'] = GBR.predict(X_test)

#After Feature Elimination Data set
EliminatedGBR = GradientBoostingRegressor(learning_rate=0.1, n_estimators=100, max_depth=3, random_state=42)
EliminatedGBR.fit(eliminatedX_train, y_train)
predictions['Eliminated GBR'] = EliminatedGBR.predict(eliminatedX_test)

#After Normalization Data set
ScaledGBR = GradientBoostingRegressor(learning_rate=0.1, n_estimators=100, max_depth=3, random_state=42)
ScaledGBR.fit(scaledX_train, y_train)
predictions['Scaled GBR'] = ScaledGBR.predict(scaledX_test)

#After Feature Elimination and Normalization Data set
EliminatedScaledGBR = GradientBoostingRegressor(learning_rate=0.1, n_estimators=100, max_depth=3, random_state=42)
EliminatedScaledGBR.fit(eliminatedScaledX_train, y_train)
predictions['Eliminated Scaled GBR'] = EliminatedScaledGBR.predict(eliminatedScaledX_test)
```

```python
# Random Forest Regressor
#Noraml Data set
RFR = RandomForestRegressor(random_state=42)
RFR.fit(X_train, y_train)
predictions['RFR'] = RFR.predict(X_test)

#After Feature Elimination Data set
EliminatedRFR = RandomForestRegressor(random_state=42)
EliminatedRFR.fit(eliminatedX_train, y_train)
predictions['Eliminated RFR'] = EliminatedRFR.predict(eliminatedX_test)

#After Normalization Data set
ScaledRFR = RandomForestRegressor(random_state=42)
ScaledRFR.fit(scaledX_train, y_train)
predictions['Scaled RFR'] = ScaledRFR.predict(scaledX_test)

#After Feature Elimination and Normalization Data set
EliminatedScaledRFR = RandomForestRegressor(random_state=42)
EliminatedScaledRFR.fit(eliminatedScaledX_train, y_train)
predictions['Eliminated Scaled RFR'] = EliminatedScaledRFR.predict(eliminatedScaledX_test)
```

```
    #Ridge Regression
    #Noraml Data set
    RR = Ridge(alpha=0.1)
    RR.fit(X_train, y_train)
    predictions['RR'] = RR.predict(X_test)

    #After Feature Elimination Data set
    EliminatedRR = Ridge(alpha=0.1)
    EliminatedRR.fit(eliminatedX_train, y_train)
    predictions['Eliminated RR'] = EliminatedRR.predict(eliminatedX_test)

    #After Normalization Data set
    ScaledRR = Ridge(alpha=0.1)
    ScaledRR.fit(scaledX_train, y_train)
    predictions['Scaled RR'] = ScaledRR.predict(scaledX_test)

    #After Feature Elimination and Normalization Data set
    EliminatedScaledRR = Ridge(alpha=0.1)
    EliminatedScaledRR.fit(eliminatedScaledX_train, y_train)
    predictions['Eliminated Scaled RR'] = EliminatedScaledRR.predict(eliminatedScaledX_test)

    return predictions

good1_predictions = model_fit_predict(X_train, X_test, good1_X_train, good1_X_test, scaledX_train, scaledX_test, good1_scaledX_train, good1_scaledX_test, good1_y_train)
good2_predictions = model_fit_predict(X_train, X_test, good2_X_train, good2_X_test, scaledX_train, scaledX_test, good2_scaledX_train, good2_scaledX_test, good2_y_train)
mid_predictions = model_fit_predict(X_train, X_test, mid_X_train, mid_X_test, scaledX_train, scaledX_test, mid_scaledX_train, mid_scaledX_test, mid_y_train)
bad1_predictions = model_fit_predict(X_train, X_test, bad1_X_train, bad1_X_test, scaledX_train, scaledX_test, bad1_scaledX_train, bad1_scaledX_test, bad1_y_train)
bad2_predictions = model_fit_predict(X_train, X_test, bad2_X_train, bad2_X_test, scaledX_train, scaledX_test, bad2_scaledX_train, bad2_scaledX_test, bad2_y_train)
```

Example of the function's return:

```
{'DT': array([26190.52, 26190.52, 25924.44, 26190.52, 23370.15, 26190.52,
       26190.52, 27437.5 , 27437.5 , 25073.01, 28510.48, 27437.5 ,
       27437.5 , 27437.5 , 23370.15, 25073.01, 27437.5 , 28510.48,
       28510.48, 33000.  , 40000.  , 44355.77, 64904.01, 40159.3 ,
       50875.  , 60000.  , 64904.01, 40159.3 , 40000.  , 40000.  ,
       41356.63, 40159.3 , 64904.01, 40159.3 , 64904.01, 64904.01]), 'Eliminated DT': array([38344.69, 38344.69, 28925.74, 28925.74, 28925.74, 28925.74,
       28925.74, 28925.74, 27786.95, 23150.  , 23150.  , 27352.83,
       27786.95, 27786.95, 23250.  , 24727.12, 27055.25, 23862.38,
       32310.18, 47756.36, 36625.  , 68229.91, 68229.91, 42500.  ,
       42000.  , 44355.77, 74375.  , 44355.77, 33000.  , 45875.19,
       33000.  , 33000.  , 37687.5 , 45875.19, 53595.95, 60779.89]), 'Scaled DT': array([25924.44, 25924.44, 25073.01, 25924.44, 25073.01, 25073.01,
       26190.52, 29437.5 , 29437.5 , 25924.44, 29437.5 , 29437.5 ,
       29437.5 , 29437.5 , 25073.01, 25073.01, 29437.5 , 29437.5 ,
       29437.5 , 33000.  , 40000.  , 43751.28, 80868.34, 43751.28,
       60000.  , 60000.  , 80868.34, 43751.28, 40000.  , 40000.  ,
       40000.  , 43751.28, 80868.34, 40000.  , 80868.34, 80868.34]), 'Eliminated Scaled DT': array([38344.69, 38344.69, 28925.74, 28925.74, 28925.74, 28925.74,
       28925.74, 28925.74, 27786.95, 23150.  , 23150.  , 27352.83,
       27786.95, 27786.95, 23250.  , 24727.12, 27055.25, 23862.38,
       32310.18, 47756.36, 36625.  , 68229.91, 68229.91, 42500.  ,
       42000.  , 44355.77, 74375.  , 44355.77, 33000.  , 45875.19,
       33000.  , 33000.  , 37687.5 , 45875.19, 53595.95, 60779.89]), 'LR': array([  47639.62406022,   76291.28892127,   84007.60211534,
         84367.32291944,   92340.22236385,   54781.46611355,
         61810.77499469,   45423.7240802 ,   61840.48634965,
         97622.29592426,   68620.39617335,   -5656.78853467,
         12197.3138489 ,   22208.96454527,   48551.26685627,
...
       49281.44743332, 53342.62218988, 58103.90151819, 47345.60234941,
       39857.54943817, 41177.40258455, 52205.26863965, 45252.36976414,
       39054.83548614, 41239.90651144, 38993.6674532 , 36884.38388488,
       36897.85515417, 40767.03091408, 45953.95275548, 43320.30498906])}
```

Step 7: Model Prediction and Evaluation

The calculate_accuracies function measures how well our models predict outcomes for both non_lagged data and lagged data. It measures the difference between what the model predicts and the actual results, turning this into a percentage that represents the model's accuracy. This is done for each prediction and then averaged out to get the model's overall accuracy. We've applied this function to several datasets, allowing us to see which model gives the most accurate predictions for each case. The results are kept neat, with individual prediction accuracies and the model's average accuracy neatly recorded and easy to compare.

```python
import numpy as np
def calculate_accuracies(predictions, y_test):
    average_accuracies = {}
    individual_accuracies = {}

    for model, model_predictions in predictions.items():
        # Calculate accuracy for each prediction
        accuracies = (1 - np.abs(model_predictions - y_test) / y_test) * 100

        # Calculate average accuracy for the model
        average_accuracy = np.mean(accuracies)

        # Store the average accuracy, rounded to two decimal places
        average_accuracies[model] = round(average_accuracy, 2)

        # Store individual accuracies
        individual_accuracies[model] = accuracies

    return average_accuracies, individual_accuracies

# Calculate accuracies for non-lagged data
good1_avg_acc, good1_ind_acc = calculate_accuracies(good1_predictions, good1_y_test)
good2_avg_acc, good2_ind_acc = calculate_accuracies(good2_predictions, good2_y_test)
mid_avg_acc, mid_ind_acc = calculate_accuracies(mid_predictions, mid_y_test)
bad1_avg_acc, bad1_ind_acc = calculate_accuracies(bad1_predictions, bad1_y_test)
bad2_avg_acc, bad2_ind_acc = calculate_accuracies(bad2_predictions, bad2_y_test)

# Calculate accuracies for lagged data
good1_lag1_avg_acc, good1_lag1_ind_acc = calculate_accuracies(good1_lag1_predictions, good1_y_lag1_test)
good2_lag1_avg_acc, good2_lag1_ind_acc = calculate_accuracies(good2_lag1_predictions, good2_y_lag1_test)
mid_lag1_avg_acc, mid_lag1_ind_acc = calculate_accuracies(mid_lag1_predictions, mid_y_lag1_test)
bad1_lag1_avg_acc, bad1_lag1_ind_acc = calculate_accuracies(bad1_lag1_predictions, bad1_y_lag1_test)
bad2_lag1_avg_acc, bad2_lag1_ind_acc = calculate_accuracies(bad2_lag1_predictions, bad2_y_lag1_test)
```

After the accuracies are calculated, the next step is to find the best model for each target ID for non-lagged data and lagged data.

```
#pick the best model for each target column for non-lagged data
good1_best_model = max(good1_avg_acc, key=good1_avg_acc.get)
good2_best_model = max(good2_avg_acc, key=good2_avg_acc.get)
mid_best_model = max(mid_avg_acc, key=mid_avg_acc.get)
bad1_best_model = max(bad1_avg_acc, key=bad1_avg_acc.get)
bad2_best_model = max(bad2_avg_acc, key=bad2_avg_acc.get)

#pick the best model for each target column for lagged data
good1_lag1_best_model = max(good1_lag1_avg_acc, key=good1_lag1_avg_acc.get)
good2_lag1_best_model = max(good2_lag1_avg_acc, key=good2_lag1_avg_acc.get)
mid_lag1_best_model = max(mid_lag1_avg_acc, key=mid_lag1_avg_acc.get)
bad1_lag1_best_model = max(bad1_lag1_avg_acc, key=bad1_lag1_avg_acc.get)
bad2_lag1_best_model = max(bad2_lag1_avg_acc, key=bad2_lag1_avg_acc.get)
```

Finally, comparing the best model for each target ID for non-lagged data and lagged data. Picking the best of the best.

```
#compare the best model for each target column for non-lagged and lagged data
#pick best of the best
best_of_the_best = {}
best_of_the_best['Good1'] = good1_best_model if good1_avg_acc[good1_best_model] > good1_lag1_avg_acc[good1_lag1_best_model] else good1_lag1_best_model
best_of_the_best['Good2'] = good2_best_model if good2_avg_acc[good2_best_model] > good2_lag1_avg_acc[good2_lag1_best_model] else good2_lag1_best_model
best_of_the_best['Mid'] = mid_best_model if mid_avg_acc[mid_best_model] > mid_lag1_avg_acc[mid_lag1_best_model] else mid_lag1_best_model
best_of_the_best['Bad1'] = bad1_best_model if bad1_avg_acc[bad1_best_model] > bad1_lag1_avg_acc[bad1_lag1_best_model] else bad1_lag1_best_model
best_of_the_best['Bad2'] = bad2_best_model if bad2_avg_acc[bad2_best_model] > bad2_lag1_avg_acc[bad2_lag1_best_model] else bad2_lag1_best_model
```

We can easily find out the best model for each target ID by printing the results:

```
Non-Lagged Data:
Avg_Good1:  {'DT': 84.79, 'Eliminated DT': 73.8, 'Scaled DT': 78.99, 'Eliminated Scaled DT': 73.8, 'LR': -183.68, 'Eliminated LR': 7
Avg_Good2:  {'DT': 96.47, 'Eliminated DT': 79.27, 'Scaled DT': 96.47, 'Eliminated Scaled DT': 79.27, 'LR': 70.49, 'Eliminated LR': 7
Avg_Mid:  {'DT': 79.45, 'Eliminated DT': 68.55, 'Scaled DT': 79.45, 'Eliminated Scaled DT': 68.55, 'LR': -212.78, 'Eliminated LR': 7
Avg_Bad1:  {'DT': 44.27, 'Eliminated DT': 78.44, 'Scaled DT': 40.38, 'Eliminated Scaled DT': 78.44, 'LR': -734.57, 'Eliminated LR':
Avg_Bad2:  {'DT': 27.04, 'Eliminated DT': 58.26, 'Scaled DT': 28.62, 'Eliminated Scaled DT': 58.26, 'LR': -1504.31, 'Eliminated LR':
Best_Good1:  RFR 88.61
Best_Good2:  Scaled Lasso 97.91
Best_Mid:  RFR 89.14
Best_Bad1:  Scaled Lasso 87.06
Best_Bad2:  Eliminated LR 84.06
Lagged Data:
Avg_Good1_Lag1:  {'Lagged DT': 88.72, 'Eliminated Lagged DT': 69.03, 'Scaled Lagged DT': 89.68, 'Eliminated Scaled Lagged DT': 69.03
Avg_Good2_Lag1:  {'Lagged DT': 75.95, 'Eliminated Lagged DT': 78.46, 'Scaled Lagged DT': 76.73, 'Eliminated Scaled Lagged DT': 78.46
Avg_Mid_Lag1:  {'Lagged DT': 83.16, 'Eliminated Lagged DT': 60.49, 'Scaled Lagged DT': 83.16, 'Eliminated Scaled Lagged DT': 60.49,
Avg_Bad1_Lag1:  {'Lagged DT': 45.29, 'Eliminated Lagged DT': 39.8, 'Scaled Lagged DT': 48.92, 'Eliminated Scaled Lagged DT': 39.8, '
Avg_Bad2_Lag1:  {'Lagged DT': -9.99, 'Eliminated Lagged DT': -0.06, 'Scaled Lagged DT': -8.48, 'Eliminated Scaled Lagged DT': -0.06,
Best_Good1_Lag1:  Scaled Lagged DT 89.68
Best_Good2_Lag1:  Scaled Lagged Lasso 97.9
Best_Mid_Lag1:  Lagged RFR 89.28
Best_Bad1_Lag1:  Scaled Lagged Lasso 85.63
Best_Bad2_Lag1:  Eliminated Lagged MLP 83.81
Best of the Best:
{'Good1': 'Scaled Lagged DT', 'Good2': 'Scaled Lasso', 'Mid': 'Lagged RFR', 'Bad1': 'Scaled Lasso', 'Bad2': 'Eliminated LR'}
```

Summary:

Non-lagged data:

Decision Tree

ID	Normal Data	Data After Elimination	Data After Normalization	Data After Elimination and Normalization
542236	**84.79%**	73.8%	78.99%	73.8%
67321	**96.47%**	79.27%	**96.47%**	79.27%
549295	**79.45%**	68.55%	**79.45%**	68.55%
41108	44.27%	**78.44%**	40.38%	**78.44%**
541982	27.04%	**58.26%**	28.62%	**58.26%**

Linear Regression

ID	Normal Data	Data After Elimination	Data After Normalization	Data After Elimination and Normalization
542236	-183.68%	**75.34%**	-29.59%	**75.34%**
67321	70.49%	78.44%	**90.63%**	78.44%
549295	-212.78%	**79.06%**	-59.98%	**79.06%**
41108	-734.57%	**84.21%**	-348.39%	**84.21%**
541982	-1504.31%	**84.06%**	-561.86%	**84.06%**

Lasso

ID	Normal Data	Data After Elimination	Data After Normalization	Data After Elimination and Normalization
542236	13.16%	**75.34%**	21.87%	75.33%
67321	96.77%	78.45%	**97.91%**	79.22%

549295	69%	78.92%	**82.59%**	74.55%
41108	41.87%	84.2%	**87.06%**	83.24%
541982	-23.08%	**84.06%**	72.89%	83.33%

Support Vector Regression

ID	Normal Data	Data After Elimination	Data After Normalization	Data After Elimination and Normalization
542236	61.08%	**61.09%**	61.08%	61.08%
67321	79.18%	80.73%	**82.31%**	81.76%
549295	69.69%	74.91%	71.16%	**76.04%**
41108	70.8%	71.22%	68.79%	**73.47%**
541982	61.08%	68.05%	58.64%	**70.42%**

Multi-Layer Perceptron Regressor

ID	Normal Data	Data After Elimination	Data After Normalization	Data After Elimination and Normalization
542236	-277.74%	**73.22%**	56.01%	71.91%
67321	-92192.1%	**88.59%**	-3969.56%	76.22%
549295	-1474537.93%	**83.9%**	64.89%	81.17%
41108	-117778.35%	81.73%	59.1%	**82.25%**
541982	-749339.64%	78.94%	-379.61%	**79.78%**

Gradient Boosting Regressor

ID	Normal Data	Data After Elimination	Data After Normalization	Data After Elimination and Normalization
542236	**85.11%**	82.37%	85.1%	82.37%

67321	**97.76%**	78.98%	**97.76%**	78.98%
549295	**86.44%**	82.91%	**86.44%**	82.91%
41108	71%	**80.23%**	66.53%	**80.23%**
541982	51.23%	**82.05%**	50.52%	**82.05%**

Random Forest Regressor

ID	Normal Data	Data After Elimination	Data After Normalization	Data After Elimination and Normalization
542236	**88.61%**	78.66%	88.05%	78.65%
67321	**96.26%**	79.65%	96.13%	79.65%
549295	**89.14%**	82.88%	88.92%	82.88%
41108	74.72%	**82.97%**	74.18%	**82.97%**
541982	60.57%	79.17%	59.59%	**79.18%**

Ridge Regression

ID	Normal Data	Data After Elimination	Data After Normalization	Data After Elimination and Normalization
542236	10.36%	75.34%	**84.06%**	74.24%
67321	84.61%	78.44%	**93.22%**	78.54$
549295	-13.42%	**79.06%**	68.39%	78.67%
41108	-282.54%	**84.21%**	60.62%	83.48%
541982	-444.67%	**84.06%**	32.71%	83.75%

Lagged data:

Decision Tree

ID	Normal Data	Data After Elimination	Data After Normalization	Data After Elimination and Normalization
542236	88.72%	69.03%	**89.68%**	69.03%
67321	75.95%	**78.46%**	76.73%	**78.46%**
549295	**83.16%**	60.49%	**83.16%**	60.49%
41108	45.29%	39.8%	**48.92%**	39.8%
541982	-9.99%	**-0.06%**	-8.48%	**-0.06%**

Linear Regression

ID	Normal Data	Data After Elimination	Data After Normalization	Data After Elimination and Normalization
542236	23.92%	**75.07%**	73.34%	**75.07%**
67321	58.03%	73.48%	**89.67%**	73.48%
549295	-9.89%	**79.54%**	64.33%	**79.54%**
41108	-178.0%	**84.14%**	59.51%	**84.14%**
541982	-555.13%	**83.02%**	40.42%	**83.02%**

Lasso

ID	Normal Data	Data After Elimination	Data After Normalization	Data After Elimination and Normalization
542236	58.26%	**75.07%**	72.09%	75.05%
67321	94.07%	73.58%	**97.9%**	75.22%
549295	36.47%	79.76%	**82.67%**	74.57%
41108	-10.27%	84.08%	**85.63%**	83.01%

541982	-68.31%	83%	74.28%	**83.15%**

Support Vector Regression

ID	Normal Data	Data After Elimination	Data After Normalization	Data After Elimination and Normalization
542236	61.08%	**61.09%**	61.08%	61.08%
67321	79.09%	**82.64%**	82.04%	81.12%
549295	69.88%	77.32%	71.15%	**82.75%**
41108	70.99%	**78.99%**	68.89%	70.23%
541982	61.38%	**73.98%**	58.77%	66.37%

Multi-Layer Perceptron Regressor

ID	Normal Data	Data After Elimination	Data After Normalization	Data After Elimination and Normalization
542236	-460.51%	**72.39%**	54.32%	53.2%
67321	-66991.14%	**83.05%**	-44.44%	75.68%
549295	-1449880.09%	**86.09%**	-172.85%	78.74%
41108	-185318.76%	**81.13%**	42.62%	79.82%
541982	-300988.1%	**83.81%**	30.21%	77.93%

Gradient Boosting Regressor

ID	Normal Data	Data After Elimination	Data After Normalization	Data After Elimination and Normalization
542236	**89.38%**	63.38%	89.37%	63.03%
67321	**96.8%**	78.92%	**96.8%**	78.92%
549295	**87.45%**	60.68%	**87.45%**	60.68%

41108	69.91%	41.46%	**70.01%**	41.46%
541982	**34.61%**	4.44%	32.99%	4.44%

Random Forest Regressor

ID	Normal Data	Data After Elimination	Data After Normalization	Data After Elimination and Normalization
542236	88.05%	71.43%	**88.12%**	71.45%
67321	94.23%	79.33%	**94.24%**	79.33%
549295	**89.28%**	74.35%	89.14%	74.37%
41108	**75.31%**	62.1%	74.38%	62.1%
541982	62.75%	45.79%	**64.24%**	45.8%

Ridge Regression

ID	Normal Data	Data After Elimination	Data After Normalization	Data After Elimination and Normalization
542236	5.5%	**75.07%**	74.64%	74.02%
67321	52.34%	73.48%	**89.8%**	73.77%
549295	-4.14%	**79.54%**	56.18%	79.12%
41108	-234.62%	**84.14%**	62.52%	83.41%
541982	-724.83%	**83.02%**	46.73%	83.01%

After comparison, we can summarize in the below table:

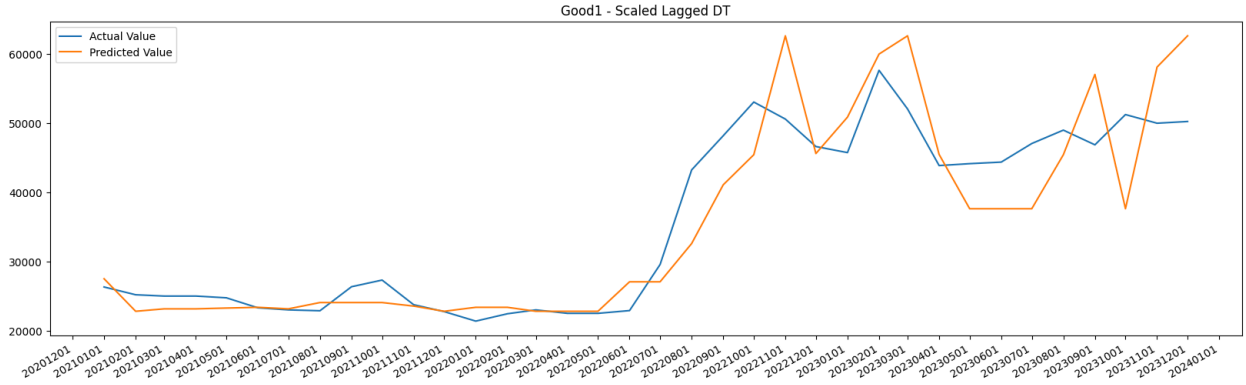
ID	Best Model	Best Accuracy
542236	**Decision Tree + Normalization + Lag**	**89.68%**

67321	**Lasso + Normalization**	**97.91%**
549295	**Random Forest Regressor + Lag**	**89.28%**
41108	**Lasso + Normalization**	**87.06%**
541982	**Linear Regression + Elimination**	**84.06%**

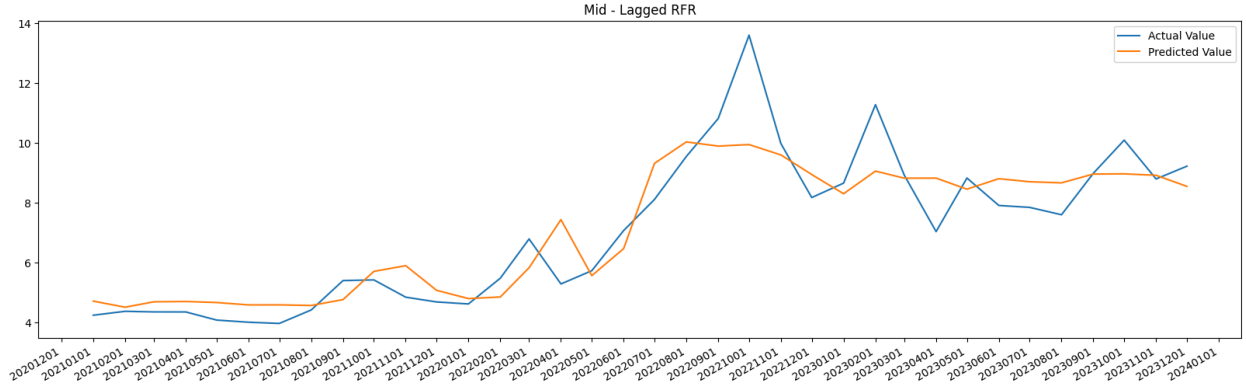
Step 8: Plot Diagrams

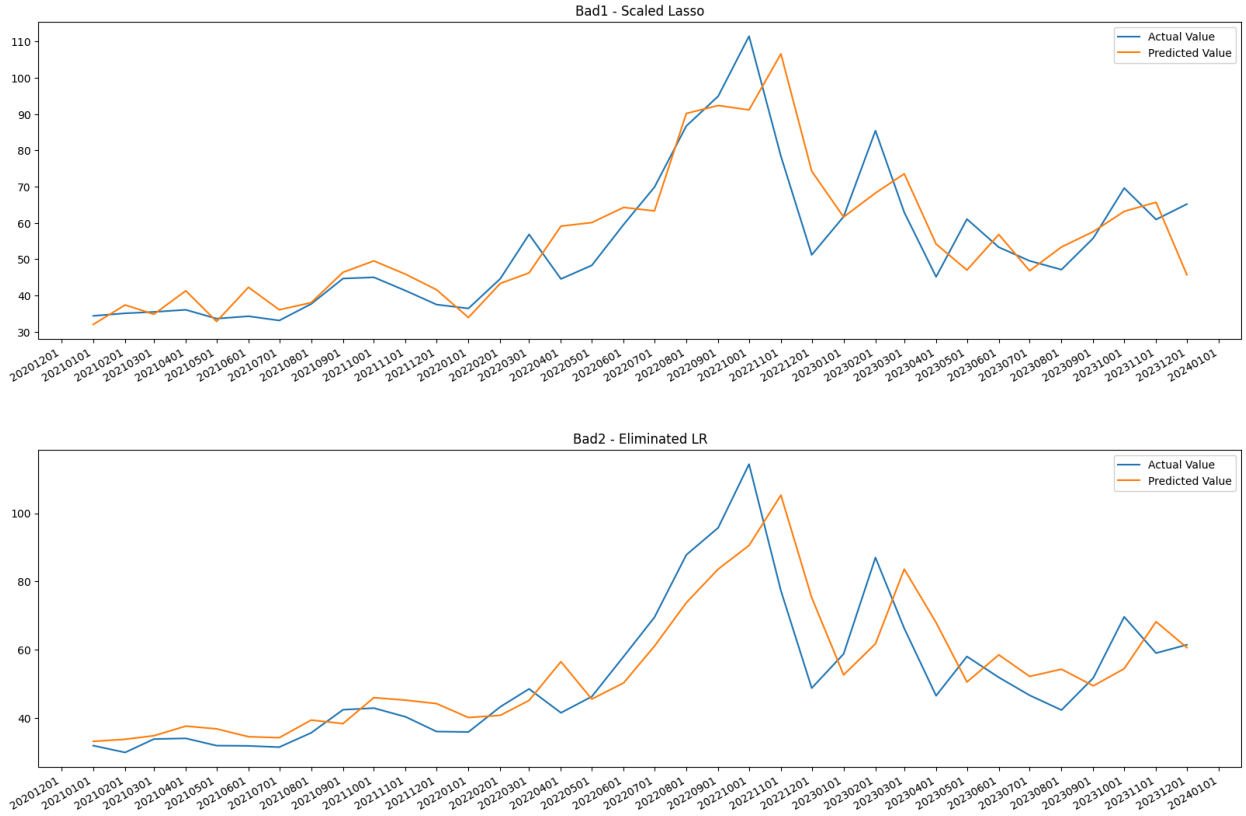
Using the Matplotlib library for plotting the performance of prediction models over time. The script defines a function called plot_best_model.These plots are critical for visually assessing how well each model's predictions align with the actual data, which is an essential part of the model evaluation process.

```
def plot_best_model(best_model, y_test, predictions, title):
    X_test['Datetime'] = pd.to_datetime(X_test['Unnamed: 0'], format='%Y%m%d')
    plt.figure(figsize=(20,6))
    plt.plot(X_test['Datetime'], y_test, label='Actual Value')
    plt.plot(X_test['Datetime'], predictions[best_model], label='Predicted Value')
    plt.title(title)
    plt.legend()
    # Format the dates on the x-axis
    plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y%m%d'))
    plt.gca().xaxis.set_major_locator(mdates.MonthLocator())

    # Rotate and align the tick labels so they look better
    plt.gcf().autofmt_xdate()
    plt.show()

#plot the best of the best model for each target column
#if the best model is from non-lagged data, plot the non-lagged data
#if the best model is from lagged data, plot the lagged data
if best_of_the_best['Good1'] in good1_predictions:
    plot_best_model(best_of_the_best['Good1'], good1_y_test, good1_predictions, f"Good1 - {best_of_the_best['Good1']}")
else:
    plot_best_model(best_of_the_best['Good1'], good1_y_lag1_test, good1_lag1_predictions, f"Good1 - {best_of_the_best['Good1']}")

if best_of_the_best['Good2'] in good2_predictions:
    plot_best_model(best_of_the_best['Good2'], good2_y_test, good2_predictions, f"Good2 - {best_of_the_best['Good2']}")
else:
    plot_best_model(best_of_the_best['Good2'], good2_y_lag1_test, good2_lag1_predictions, f"Good2 - {best_of_the_best['Good2']}")

if best_of_the_best['Mid'] in mid_predictions:
    plot_best_model(best_of_the_best['Mid'], mid_y_test, mid_predictions, f"Mid - {best_of_the_best['Mid']}")
else:
    plot_best_model(best_of_the_best['Mid'], mid_y_lag1_test, mid_lag1_predictions, f"Mid - {best_of_the_best['Mid']}")

if best_of_the_best['Bad1'] in bad1_predictions:
    plot_best_model(best_of_the_best['Bad1'], bad1_y_test, bad1_predictions, f"Bad1 - {best_of_the_best['Bad1']}")
else:
    plot_best_model(best_of_the_best['Bad1'], bad1_y_lag1_test, bad1_lag1_predictions, f"Bad1 - {best_of_the_best['Bad1']}")

if best_of_the_best['Bad2'] in bad2_predictions:
    plot_best_model(best_of_the_best['Bad2'], bad2_y_test, bad2_predictions, f"Bad2 - {best_of_the_best['Bad2']}")
else:
    plot_best_model(best_of_the_best['Bad2'], bad2_y_lag1_test, bad2_lag1_predictions, f"Bad2 - {best_of_the_best['Bad2']}")
```


Good1 - Scaled Lagged DT
Actual Value
Predicted Value
60000
50000
40000
30000
20000


Good2 - Scaled Lasso
Actual Value
Predicted Value
130
125
120
115
110
105
100
95
90


Mid - Lagged RFR
Actual Value
Predicted Value
14
12
10
8
6
4



Step 9: Export csv files

we devised the write_csv function. This function takes in the actual values, the predicted values, the individual prediction accuracies, the average accuracy, and the desired filename for output.

For every model, we used the function to write a CSV file containing the time series from our test set, along with the corresponding actual and predicted values, and their accuracies. It neatly places the average accuracy at the top of the column, avoiding repetition, for a cleaner look in the data file.

```python
import csv

def write_csv(y_test, y_pred, ind_acc, avg_acc, filename):
    with open(filename, 'w', newline='') as file:
        X_test_local = X_test.set_index('Unnamed: 0')
        writer = csv.writer(file)
        writer.writerow(["Time Series", "Target Value", "Predicted Value", "Accuracy", "Average Accuracy"])
        for i in range(len(y_test)):
            if i == 0:
                writer.writerow([X_test_local.index[i], y_test.iloc[i], y_pred[i], ind_acc.loc[361+i], avg_acc])
            else:
                writer.writerow([X_test_local.index[i], y_test.iloc[i], y_pred[i], ind_acc.loc[361+i], ""])

if best_of_the_best['Good1'] in good1_predictions:
    write_csv(good1_y_test, good1_predictions[best_of_the_best['Good1']], good1_ind_acc[best_of_the_best['Good1']], good1_avg_acc[best_of_the_best['Good1']], 'good1.csv')
else:
    write_csv(good1_y_lag1_test, good1_lag1_predictions[best_of_the_best['Good1']], good1_lag1_ind_acc[best_of_the_best['Good1']], good1_lag1_avg_acc[best_of_the_best['Good1']], 'good1.csv')

if best_of_the_best['Good2'] in good2_predictions:
    write_csv(good2_y_test, good2_predictions[best_of_the_best['Good2']], good2_ind_acc[best_of_the_best['Good2']], good2_avg_acc[best_of_the_best['Good2']], 'good2.csv')
else:
    write_csv(good2_y_lag1_test, good2_lag1_predictions[best_of_the_best['Good2']], good2_lag1_ind_acc[best_of_the_best['Good2']], good2_lag1_avg_acc[best_of_the_best['Good2']], 'good2.csv')

if best_of_the_best['Mid'] in mid_predictions:
    write_csv(mid_y_test, mid_predictions[best_of_the_best['Mid']], mid_ind_acc[best_of_the_best['Mid']], mid_avg_acc[best_of_the_best['Mid']], 'mid.csv')
else:
    write_csv(mid_y_lag1_test, mid_lag1_predictions[best_of_the_best['Mid']], mid_lag1_ind_acc[best_of_the_best['Mid']], mid_lag1_avg_acc[best_of_the_best['Mid']], 'mid.csv')

if best_of_the_best['Bad1'] in bad1_predictions:
    write_csv(bad1_y_test, bad1_predictions[best_of_the_best['Bad1']], bad1_ind_acc[best_of_the_best['Bad1']], bad1_avg_acc[best_of_the_best['Bad1']], 'bad1.csv')
else:
    write_csv(bad1_y_lag1_test, bad1_lag1_predictions[best_of_the_best['Bad1']], bad1_lag1_ind_acc[best_of_the_best['Bad1']], bad1_lag1_avg_acc[best_of_the_best['Bad1']], 'bad1.csv')

if best_of_the_best['Bad2'] in bad2_predictions:
    write_csv(bad2_y_test, bad2_predictions[best_of_the_best['Bad2']], bad2_ind_acc[best_of_the_best['Bad2']], bad2_avg_acc[best_of_the_best['Bad2']], 'bad2.csv')
else:
    write_csv(bad2_y_lag1_test, bad2_lag1_predictions[best_of_the_best['Bad2']], bad2_lag1_ind_acc[best_of_the_best['Bad2']], bad2_lag1_avg_acc[best_of_the_best['Bad2']], 'bad2.csv')
```