1. Introduction The Problem

PROBLEM: Predict the foreign currency by using the other foreign currencies.

The given problem statement involves the task of predicting the value of a specific foreign currency using the values of other foreign currencies. This is a regression problem, and I need to build a model that predicts the value of a specific currency using input features that include the values of multiple currencies.

Code Phase 1

The instruction provided in Phase 1 involves the task of reading the training dataset for a model using the dataset named "Doviz_Satislari_20050101_20231205_Training_Set.xlsx." This dataset contains records for 8 different foreign currencies from January 3, 2005, to November 28, 2023. Seven of these currencies are marked as input features for the model, and the last one is marked as the output for the model. What is required is to write Python code that reads the training dataset. The code should be capable of reading inputs and outputs and have the ability to query a specific record, such as printing the 4512th record.

Code Phase 2

The instructions provided in Phase 2 involve writing code for specific functions related to training a machine learning model.

Cost/Loss Function Code:

In the Cost/Loss Function Code, we are required to measure the difference between the predicted values and the true values using this function. The Mean Squared Error (MSE) formula can be used for this purpose. MSE takes the deviation of each prediction from the true value, squares these values, takes the mean, and ultimately calculates a cost value. This process allows us to quantify how much the model's predictions deviate from the actual values, providing a measure of the overall model performance.

Derivative of Cost/Loss Function Code:

Here, the derivative of the desired cost function shows how the cost function changes according to the model parameters. This is crucial for optimization algorithms such as gradient descent. The derivative calculation shows the change in the cost function for each model parameter.

Optimizer Function Code:

This function requires that the model parameters be updated using the derivative of the cost function. The learning rate controls the size of these update steps.

Model Code:

This function expects us to train the model on given input data (X) and target output data (y). It requests the update of model parameters to minimize the cost over a specified number of epochs.

Code Phase 3

This task wants a hybrid regulation to be implemented, combining L2 and L1 penalties to prevent over-compliance**.** This is achieved by constraining the model parameters and adjusting the weights.

Code Test Phase

In this section, we are asked to write a test function that takes 7 input currency values and predicts the output of the model. This function will take the input features on which the model was trained and predict the output.

2. Solving The Problem

Code Phase 1

I implement phase 1 in my code you see in Figure **2.1.1.** If I were to explain what I did in this code; First of all, I converted the data in the Excel file to CSV file format because I believed that I could carry out my operations more easily.
We can see part of the view of the csv file in figure **2.1.2**. If I were to explain the procedures I did in more detail; I imported the pandas library I needed, read my csv file into a pandas dataframe called df, and printed the df to the console with the print function to check the accuracy of the file. We can see this output in figure **2.1.3.** I then took the input value from the user, converted it to an integer and had it select row(s) from the DataFrame whose 'No' column was equal to the entered value. For example, when the user types the value 5, information for data number 5 will appear. We can see this output in figure 2.1.4 and compare it with the data in our csv file to check its accuracy. At the same time, we can see 7 data as input features and the last 1 data as output features. In the rest of my code, I checked whether the selected row was empty. As I said, I selected the columns for the input and output features and printed them to the screen. If a matching row was not found, I put another print to print. In summary, this code allows the user to enter a value for the 'No' column, retrieves the relevant record from the dataset and displays the input properties and output properties of that record. If no matching row is found, it notifies the user.

Code Phase 2

First, we can see the code I wrote in the part of Code Phase 2 in figure **2.2.1.1**, figure **2.2.1.2**, figure **2.2.1.3.** I thought a multiple linear regression model would be appropriate for this problem because more than one independent variable affects one dependent variable. First of all, I want to start by explaining the general code. I imported the necessary libraries, opened a class for my multiple linear regression, and initially defined a property called weights. My "fit" function is used to train the model, here a bias term is added to the input data (X). Weights are initialized with zeros and training is performed for the specified number of epochs. At each iteration, predictions are calculated, mean square error (MSE) is derived, and weights are updated using gradient descent. MSE stands for mean square error. This is an error metric used in machine learning. MSE is the mean square of the differences between actual values and predicted values. MSE measures how accurate the model's predictions are. The smaller the MSE, the better the model. Gradient descent is an optimization method used in machine learning. This method aims to reach the global minimum value by starting with randomly taken variables. The global minimum value is the point where the cost or loss function is lowest. My “predict” method makes predictions on given input data. For this prediction, a bias term is added to the input data and the prediction is calculated using the weights. My "evaluate_performance" method is used to evaluate the performance of the model. Based on the actual and predicted values, the mean absolute error (MAE), mean square error (MSE), root mean square error (RMSE) and R-square score are calculated. Then, I read the data in the csv file, cleaned the lines with empty values because they were causing problems, I wrote

the cleaned version to another csv file and saved it (cleaned_dataset.csv). Then I determined the dependent and independent variables, and split the data into training and testing sets in my "train_test_split_custom" function. After training the model with some data sets, we can observe how it makes a prediction by giving the independent variables of the test data set. In the rest of the code, we create a model from the MyMultipleLinearRegression class, train it and make predictions on the test set, as I said.

Finally, we print the metrics that evaluate the performance of the model and the table comparing the actual values and predicted values at the output. We can see the output in figure **2.2.2.**

Cost/Loss Function:

I had Mean Squared Error (MSE) calculated for the cost/loss function, which was our first task for Phase 2. The cost or loss function measures the difference between the model's predictions and actual values. You can see where I did this in the evaluate_performance function in figure **2.2.1.1.**

y_pred - y_true: Calculates the difference between the model's predicted value and the actual value for each data point.
(y_pred - y_true)2**: Squares the differences.
np.mean((y_pred - y_true)2)**: Takes the average of these frames.

The result represents the squared average of the model's prediction errors over the entire data set. A lower MSE indicates that the model performs better because in this case the predictions are closer to the actual values.

Derivative Cost/Loss Function:

I added this part after the session 1 hackhathon. We can see the code I added in figure **2.2.3.** The cycle is started for the number of epochs specified here, predictions are made using the current weights of the model, **error = y_pred - y**: The error (difference) between the actual values and the predictions is calculated, **derivative = (2/len(y)) * X.T.dot(error):** Average is the derivative of the squared error (MSE). This is used to determine the gradient descent step, every 100 epochs the derivative of the MSE is printed to the screen which helps track the progress of gradient descent, **self.weights -= learning_rate * derivative**: Gradient descent step; The weights are subtracted from the previous weights by multiplying the learning rate and the derivative Gradient descent updates the variables step by step using the derivative of the cost or loss function. Epoch means training the model using all the training data once. The number of epochs determines how many times the model sees the training data. Increasing the number of epochs can help the model better fit the training data. However, increasing the number of epochs too much may cause the model to overfit and not generalize well to new data. Therefore, it is important to determine the number of epochs appropriately. After the training is completed, the final derivative value is printed on the screen. We can see the output in figure **2.2.4.**

Optimizer Function:

I added this part after the session 1 hackhathon. The optimization happens in my fit method. In this method, the weights are updated using the gradient descent step. We can see this part in figure **2.2.5.** In Figure **2.2.6**, we can see the **update_weights method**. This method is used to update the weights. Gradient descent step occurs with the following formula: updated_weights = weights − learning_rate ×derivative These steps are performed for each epoch, thus training the model. In the fit method, these steps are repeated continuously, and the weights of the model are optimized to represent the relationship between the characteristics of the data and the target variable.

Model Code:

In this section, we can see my entire model code that makes predictions based on input features. (Derivation cost/loss and optimization parts were added to the model after the session 1 hackhathon, the remaining part was done in session 1). I explained in detail what the model does in the fields above. We can see the final and complete version of the model in figures **2.2.7, 2.2.8** and **2.2.9**. I would like to say again in this section that I did it using the Multiple Linear Regression model.

Code Phase 3

I added this part after the session 1 hackhathon. In this section, I tried to solve the overfitting problem with the Hybrid L2/L1 penalty. Hybrid L2/L1 is a regularization technique used in machine learning. Regularization is a method used to prevent the model from overfitting. Overfitting means that the model fits the training data very well but does not generalize well to new data. Regularization imposes a penalty on model parameters, reducing the complexity of the model.
Hybrid L2/L1 is a mixture of L2 and L1 regularizations. L2 regularization attempts to minimize the sum of squares of the model parameters. This prevents model parameters from becoming too large. L1 regularization attempts to minimize the sum of the absolute value of the model parameters. This sets some of the model parameters equal to zero, ensuring that the model selects only the features that are important. Hybrid L2/L1 regularization is a weighted average of L2 and L1 regularizations. This ensures that the model avoids both large and unnecessary parameters. Since I thought I could see this part more clearly, I created a new class and used the codes and models I used in the previous parts. I put my alpha parameters in the init function. We can see this in figure **2.3.1.** The part we see in Figure **2.3.2** is where the derivative of the cost function of the multiple linear regression model is calculated with the hybrid L1 (LASSO) and L2 (Ridge) arrangement. **error = y_pred - y:** The error (residuals) between the actual values and the predicted values is calculated.
X.T.dot(error): The inner product of the transpose of the X matrix and the error vector is taken. This gives the sum of the independent variable values for each observation multiplied by the error.
self.alpha_l2 * self.weights: Applies L2 regularization. L2 regularization adds the sum of squares of the weights to the cost function. self.alpha_l2 determines the coefficient of this regularization term.
self.alpha_l1 * np.sign(self.weights): Applies L1 regularization. L1 regularization adds the sum of the absolute values of the weights to the cost function. self.alpha_l1 determines the coefficient of this regularization term.
(2/len(y)) * ...: Multiplied by the weights of the calculated derivative terms and divided by 2. This allows averaging and scaling of the derivative. Alpha and learning rate are two hyper-parameters used in machine learning. A hyper-parameter is a parameter that affects the performance of the model, but is not learned by the model, but is determined by the person who designed the model.

Alpha is a parameter that determines the effect of L2 regularization. L2 regularization is a method that tries to minimize the sum of the squares of the model parameters to prevent the model from overfitting. Alpha determines how much penalty is applied to this total. The larger the alpha, the smaller the model parameters and the simpler the model. The smaller the alpha, the larger the model parameters and the more complex the model. Choosing an appropriate value of alpha ensures that the model fits both training and testing data well.

Learning rate is a parameter that determines the step size of the gradient descent algorithm. Gradient reduction is an optimization method that uses model parameters to move closer to the point where the cost or loss function is lowest. The cost or loss function measures the difference between the model's predictions and the actual values. Gradient decay updates the model parameters step by step, using the derivative of the cost or loss function. The learning rate determines how big or small this step will be. The larger the learning rate, the faster the model parameters are updated, but the risk of missing the optimum point increases. The smaller the learning rate, the slower the model parameters are updated, but the greater the chance of approaching the optimum point more precisely. Choosing an appropriate value for the learning rate ensures that the model is trained both quickly and accurately.
As a result, the derivative variable is the derivative of the MSE cost function with respect to the weights, and this derivative is used to update the weights using the gradient descent algorithm. L1 and L2 regularization terms are added to the update of the weights, making the model resistant to overfitting. L1 editing can make feature selection by making some of the weights zero, while L2 editing can increase the generalization of the model by reducing the weights. We can see the entire code in figures **2.3.3** , **2.3.4** and **2.3.5.**

Code Test Phase

I added this part after the session 1 hackhathon. In this part, we try to make the model we trained make a prediction with the 7 input values we received from the user. We can see the code I added for this in figure **2.4.1.**
user_inputs = []: A list is created to hold the values entered by the user.
for i in range(2, 9): A loop is started. This loop rotates according to the number of features in the data set on which the model is trained. Since the number of features in the data set is 7, the loop continues from 2 to 8 using range(2, 9).
value = float(input(f'Enter the value for Feature {i-1}: ')): The input function is used to get the value of each feature from the user. An index of the form i-1 is used because we want to show the user sequential feature numbers starting from 1.
user_inputs.append(value): The value entered by the user is added to the user_inputs list.
user_prediction =model.predict(np.array(user_inputs).reshape(1, -1)): The list containing the values entered by the user is passed to the model's predict function. During this process, the input data is transformed (reshaped) into a format that the model can receive as expected. The result is assigned to the user_prediction variable.
print(f'\nUser Input: {user_inputs}'): The values entered by the user are printed on the screen.
print(f'Predicted Value: {user_prediction[0]:.2f}'): The prediction result made by the model based on these inputs is printed on the screen. The :.2f format is used to format a decimal number with two digits after the comma. We can see the output I added in figure **2.4.2.** When we entered these 7 values written by the user into our trained model, our model made a prediction.

Doviz.py > ...

```python
import pandas as pd

# Read the CSV file
df = pd.read_csv("Doviz_Satislari_20050101_20231205_Training_Set.csv")

# Display the entire dataset
print(df)

# Get user input for "No" column value
i = int(input("Enter the value for 'No' column: "))
selected_row = df[df["No"] == i]

# Check if the selected row is not empty
if not selected_row.empty:
    # Extract input features and output feature from the selected row
    input_features = selected_row.iloc[0, 2:9]
    output_feature = selected_row.iloc[0, 9]

    # Display the selected record
    print(f"\nInput Features (record {i}):")
    print(input_features)

    print("\nOutput Feature:")
    print(output_feature)
else:
    print(f"No matching row found for the given 'No' column value: {i}")
```

Figure 2.1.1

Doviz_Satislari_20050101_20231205_Training_Set.csv

```
"No","Tarih","TP DK USD S YTL","TP DK EUR S YTL","TP DK GBP S YTL","TP DK SEK S YTL","TP DK CHF S YTL","TP DK CAD S YTL","TP DK KWD S YTL","TP DK S
"1","03-01-2005","1.34","1.83","2.59","0.20","1.19","1.12","4.55","0.36"
"2","04-01-2005","1.34","1.82","2.57","0.20","1.18","1.11","4.61","0.36"
"3","05-01-2005","1.35","1.81","2.56","0.20","1.17","1.11","4.62","0.36"
"4","06-01-2005","1.38","1.83","2.60","0.20","1.18","1.13","4.74","0.37"
"5","07-01-2005","1.41","1.85","2.63","0.21","1.20","1.14","4.82","0.38"
"6","10-01-2005","1.38","1.82","2.59","0.20","1.18","1.12","4.72","0.37"
"7","11-01-2005","1.39","1.82","2.61","0.20","1.18","1.13","4.75","0.37"
"8","12-01-2005","1.38","1.82","2.59","0.20","1.17","1.13","4.73","0.37"
"9","13-01-2005","1.38","1.81","2.59","0.20","1.17","1.13","4.73","0.37"
"10","14-01-2005","1.36","1.80","2.56","0.20","1.16","1.13","4.65","0.36"
"11","17-01-2005","1.36","1.78","2.54","0.20","1.15","1.12","4.64","0.36"
"12","18-01-2005","1.34","1.75","2.50","0.19","1.13","1.10","4.58","0.36"
"13","19-01-2005","1.34","1.75","2.51","0.19","1.13","1.10","4.60","0.36"
"14","24-01-2005","1.34","1.75","2.51","0.19","1.13","1.10","4.60","0.36"
"15","25-01-2005","1.34","1.75","2.52","0.19","1.13","1.10","4.59","0.36"
"16","26-01-2005","1.33","1.74","2.50","0.19","1.11","1.09","4.57","0.36"
"17","27-01-2005","1.33","1.73","2.50","0.19","1.12","1.08","4.57","0.36"
"18","28-01-2005","1.34","1.75","2.52","0.19","1.13","1.08","4.59","0.36"
"19","31-01-2005","1.34","1.74","2.52","0.19","1.13","1.08","4.57","0.36"
"20","01-02-2005","1.34","1.74","2.52","0.19","1.12","1.08","4.57","0.36"
"21","02-02-2005","1.33","1.74","2.51","0.19","1.12","1.07","4.56","0.36"
"22","03-02-2005","1.33","1.73","2.50","0.19","1.12","1.08","4.54","0.35"
"23","04-02-2005","1.32","1.72","2.50","0.19","1.11","1.07","4.53","0.35"
"24","07-02-2005","1.32","1.73","2.48","0.19","1.10","1.06","4.51","0.35"
"25","08-02-2005","1.32","1.69","2.46","0.19","1.08","1.07","4.51","0.35"
"26","09-02-2005","1.33","1.70","2.47","0.19","1.09","1.06","4.55","0.35"
"27","10-02-2005","1.33","1.70","2.47","0.19","1.09","1.06","4.55","0.35"
"28","11-02-2005","1.34","1.71","2.49","0.19","1.10","1.07","4.59","0.36"
"29","14-02-2005","1.32","1.70","2.47","0.19","1.09","1.06","4.53","0.35"
"30","15-02-2005","1.32","1.71","2.48","0.19","1.10","1.07","4.50","0.35"
"31","16-02-2005","1.31","1.70","2.48","0.19","1.10","1.06","4.49","0.35"
```

Figure 2.1.2

3150661/Doviz.py

	No	Tarih	TP DK USD S YTL	TP DK EUR S YTL	TP DK GBP S YTL	TP DK SEK S YTL	TP DK CHF S YTL	TP DK CAD S YTL	TP DK KWD S YTL	TP DK SAR S YTL
0	1	03-01-2005	1.34	1.83	2.59	0.20	1.19	1.12	4.55	0.36
1	2	04-01-2005	1.34	1.82	2.57	0.20	1.18	1.11	4.61	0.36
2	3	05-01-2005	1.35	1.81	2.56	0.20	1.17	1.11	4.62	0.36
3	4	06-01-2005	1.38	1.83	2.60	0.20	1.18	1.13	4.74	0.37
4	5	07-01-2005	1.41	1.85	2.63	0.21	1.20	1.14	4.82	0.38
...	...	...	...	...	...	...	...	...	...	...
4752	4753	22-11-2023	28.80	31.53	36.15	2.78	32.65	21.02	94.01	7.68
4753	4754	23-11-2023	28.81	31.41	36.15	2.76	32.66	21.04	94.02	7.68
4754	4755	24-11-2023	28.81	31.47	36.20	2.77	32.73	21.11	94.09	7.69
4755	4756	27-11-2023	28.86	31.49	36.29	2.76	32.73	21.11	94.13	7.70
4756	4757	28-11-2023	28.91	31.66	36.54	2.78	32.90	21.22	94.28	7.71

[4757 rows x 10 columns]

Figure 2.1.3

```
Enter the value for 'No' column: 5

Input Features (record 5):
TP DK USD S YTL    1.41
TP DK EUR S YTL    1.85
TP DK GBP S YTL    2.63
TP DK SEK S YTL    0.21
TP DK CHF S YTL     1.2
TP DK CAD S YTL    1.14
TP DK KWD S YTL    4.82
Name: 4, dtype: object

Output Feature:
0.38
```

Figure 2.1.4

MultipleLinearRegression.py > ...

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

class MyMultipleLinearRegression:
    def __init__(self):
        self.weights = None

    def fit(self, X, y, learning_rate=0.01, epochs=1000):
        X = np.insert(X, 0, 1, axis=1)
        self.weights = np.zeros(X.shape[1])

        for _ in range(epochs):
            y_pred = X.dot(self.weights)

            # Compute the derivative of MSE
            error = y_pred - y
            derivative = (2/len(y)) * X.T.dot(error)

            # Update weights using gradient descent
            self.weights -= learning_rate * derivative

    def predict(self, X):
        X = np.insert(X, 0, 1, axis=1)
        return X.dot(self.weights)

    def evaluate_performance(self, y_true, y_pred):
        mae = np.mean(np.abs(y_pred - y_true))
        mse = np.mean((y_pred - y_true)**2)
        rmse = np.sqrt(mse)
        total_variance = np.sum((y_true - np.mean(y_true))**2)
        explained_variance = np.sum((y_pred - np.mean(y_true))**2)
        r2 = explained_variance / total_variance

        return {'MAE': mae, 'MSE': mse, 'RMSE': rmse, 'R2 Score': r2}
```

Figure 2.2.1.1

```python
# Read the file
df = pd.read_csv('Doviz_Satislari_20050101_20231205_Training_Set_without_quotes.csv')

# Remove rows containing empty values
df_cleaned = df.dropna(subset=['TP DK USD S YTL', 'TP DK EUR S YTL'])

# Write the cleaned data to 'cleaned_dataset.csv'
df_cleaned.to_csv('cleaned_dataset.csv', index=False)

# Define dependent and independent variables
X = df_cleaned.iloc[:, 2:7].values
y = df_cleaned.iloc[:, 6].values

# Split the dataset into training and test sets
def train_test_split_custom(X, y, test_size=0.2):
    split_index = int((1 - test_size) * len(X))
    X_train, X_test = X[:split_index], X[split_index:]
    y_train, y_test = y[:split_index], y[split_index:]
    return X_train, X_test, y_train, y_test

X_train, X_test, y_train, y_test = train_test_split_custom(X, y, test_size=0.2)

# Create and train the model
model = MyMultipleLinearRegression()
model.fit(X_train, y_train, learning_rate=0.01, epochs=1000)

# Make predictions on the test set
y_pred = model.predict(X_test)

# Print predictions and actual values
results_df = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred})
print(results_df)
```

Figure 2.2.1.2

```python
# Calculate performance metrics and print them
performance_metrics = model.evaluate_performance(y_test, y_pred)
for metric, value in performance_metrics.items():
    print(f'{metric}: {value}')
```

Figure 2.2.1.3

	Actual	Predicted
0	6.21	6.213183
1	6.19	6.199541
2	6.17	6.183444
3	6.20	6.207485
4	6.19	6.197658
..	...	...
942	32.65	31.892547
943	32.66	31.876697
944	32.73	31.932792
945	32.73	31.946301
946	32.90	32.090776

[947 rows x 2 columns]
MAE: 0.17138850324373106
MSE: 0.08179535575917396
RMSE: 0.28599887370263183
R2 Score: 0.9495923817471983

Figure 2.2.2

```python
def fit(self, X, y, learning_rate=0.01, epochs=1000):
    X = np.insert(X, 0, 1, axis=1)
    self.weights = np.zeros(X.shape[1])

    for epoch in range(epochs):
        y_pred = X.dot(self.weights)

        # Compute the derivative of MSE
        error = y_pred - y
        derivative = (2/len(y)) * X.T.dot(error)

        # Print the derivative of MSE for every 100 epochs
        if epoch % 100 == 0:
            print(f'Epoch {epoch}, Derivative of MSE: {derivative}')

        # Update weights using gradient descent
        self.weights -= learning_rate * derivative

    # Print the final derivative of MSE after training
    print(f'Final Derivative of MSE: {derivative}')
```

Figure 2.2.3

```
Epoch 0, Derivative of MSE: [ -4.85202324 -15.65531532 -18.43416519 -21.88979456  -1.87365283
 -15.90864786]
Epoch 100, Derivative of MSE: [ 0.10129696 -0.03962507  0.00097666  0.06471221  0.0015344  -0.08501238]
Epoch 200, Derivative of MSE: [ 0.04810725 -0.01732481  0.00168673  0.03738071 -0.00016725 -0.05245434]
Epoch 300, Derivative of MSE: [ 0.02046737 -0.00574448  0.00212021  0.02283863 -0.00102046 -0.03513316]
Epoch 400, Derivative of MSE: [ 0.00620175  0.00022378  0.00240197  0.01500961 -0.00143092 -0.02580451]
Epoch 500, Derivative of MSE: [-0.00106764  0.0032557   0.0025979   0.01070959 -0.00161109 -0.02067275]
Epoch 600, Derivative of MSE: [-0.0046816   0.00475288  0.00274296  0.00826986 -0.00167217 -0.017749  ]
Epoch 700, Derivative of MSE: [-0.00638984  0.00544943  0.00285586  0.00681555 -0.00167235 -0.01599088]
Epoch 800, Derivative of MSE: [-0.00710854  0.00572979  0.0029468   0.00588772 -0.00164204 -0.01485196]
Epoch 900, Derivative of MSE: [-0.00731721  0.00579521  0.00302149  0.00524539 -0.0015971  -0.01404558]
Final Derivative of MSE: [-0.00726795  0.00575177  0.0030827   0.00476606 -0.00154631 -0.01342687]
```

Figure 2.2.4

```python
def fit(self, X, y, learning_rate=0.01, epochs=1000):
    X = np.insert(X, 0, 1, axis=1)
    self.weights = np.zeros(X.shape[1])

    for epoch in range(epochs):
        y_pred = X.dot(self.weights)

        # Compute the derivative of MSE
        error = y_pred - y
        derivative = (2/len(y)) * X.T.dot(error)

        # Update weights using gradient descent
        old_weights = self.weights.copy()
        self.weights = self.update_weights(self.weights, derivative, learning_rate)

        # Print the derivative of MSE, old and current weights for every 100 epochs
        if epoch % 100 == 0:
            print(f'\nEpoch {epoch}, Derivative of MSE: {derivative}')
            print(f'Old Weights: {old_weights}')
            print(f'Current Weights: {self.weights}')

    # Print the final derivative of MSE and weights after training
    print(f'\nFinal Derivative of MSE: {derivative}')
    print(f'Final Weights: {self.weights}')
```

Figure 2.2.5

```python

    def update_weights(self, weights, derivative, learning_rate):
        # Update weights using gradient descent
        updated_weights = weights - learning_rate * derivative
        return updated_weights

```

Figure 2.2.6

LastMLR.py > MyMultipleLinearRegression > fit

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

class MyMultipleLinearRegression:
    def __init__(self):
        self.weights = None

    def fit(self, X, y, learning_rate=0.01, epochs=1000):
        X = np.insert(X, 0, 1, axis=1)
        self.weights = np.zeros(X.shape[1])

        for epoch in range(epochs):
            y_pred = X.dot(self.weights)

            # Compute the derivative of MSE
            error = y_pred - y
            derivative = (2/len(y)) * X.T.dot(error)

            # Print the derivative of MSE for every 100 epochs
            if epoch % 100 == 0:
                print(f'Epoch {epoch}, Derivative of MSE: {derivative}')

                |

            # Update weights using gradient descent
            self.weights -= learning_rate * derivative

        # Print the final derivative of MSE after training
        print(f'Final Derivative of MSE: {derivative}')

    def predict(self, X):
        X = np.insert(X, 0, 1, axis=1)
        return X.dot(self.weights)
```

Figure 2.2.7

```python

    def evaluate_performance(self, y_true, y_pred):
        mae = np.mean(np.abs(y_pred - y_true))
        mse = np.mean((y_pred - y_true)**2)
        rmse = np.sqrt(mse)
        total_variance = np.sum((y_true - np.mean(y_true))**2)
        explained_variance = np.sum((y_pred - np.mean(y_true))**2)
        r2 = explained_variance / total_variance

        return {'MAE': mae, 'MSE': mse, 'RMSE': rmse, 'R2 Score': r2}

    def update_weights(self, weights, derivative, learning_rate):
        # Update weights using gradient descent
        updated_weights = weights - learning_rate * derivative
        return updated_weights


# Read the file
df = pd.read_csv('Doviz_Satislari_20050101_20231205_Training_Set_without_quotes.csv')

# Remove rows containing empty values
df_cleaned = df.dropna(subset=['TP DK USD S YTL', 'TP DK EUR S YTL'])

# Write the cleaned data to 'cleaned_dataset.csv'
df_cleaned.to_csv('cleaned_dataset.csv', index=False)

# Define dependent and independent variables
X = df_cleaned.iloc[:, 2:7].values
y = df_cleaned.iloc[:, 6].values
```

Figure 2.2.8

```python

# Split the dataset into training and test sets
def train_test_split_custom(X, y, test_size=0.2):
    split_index = int((1 - test_size) * len(X))
    X_train, X_test = X[:split_index], X[split_index:]
    y_train, y_test = y[:split_index], y[split_index:]
    return X_train, X_test, y_train, y_test

X_train, X_test, y_train, y_test = train_test_split_custom(X, y, test_size=0.2)

# Create and train the model
model = MyMultipleLinearRegression()
model.fit(X_train, y_train, learning_rate=0.01, epochs=1000)

# Make predictions on the test set
y_pred = model.predict(X_test)

# Print predictions and actual values
results_df = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred})
print(results_df)

# Calculate performance metrics and print them
performance_metrics = model.evaluate_performance(y_test, y_pred)
for metric, value in performance_metrics.items():
    print(f'{metric}: {value}')
```

Figure 2.2.9

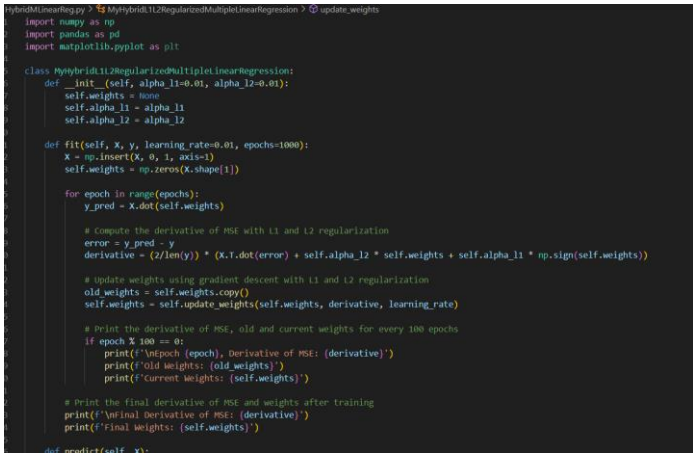
HybridMLinearReg.py > MyHybridL1L2RegularizedMultipleLinearRegression > fit

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

class MyHybridL1L2RegularizedMultipleLinearRegression:
    def __init__(self, alpha_l1=0.01, alpha_l2=0.01):
        self.weights = None
        self.alpha_l1 = alpha_l1
        self.alpha_l2 = alpha_l2

    def fit(self, X, y, learning_rate=0.01, epochs=1000):
        X = np.insert(X, 0, 1, axis=1)
        self.weights = np.zeros(X.shape[1])
```

Figure 2.3.1

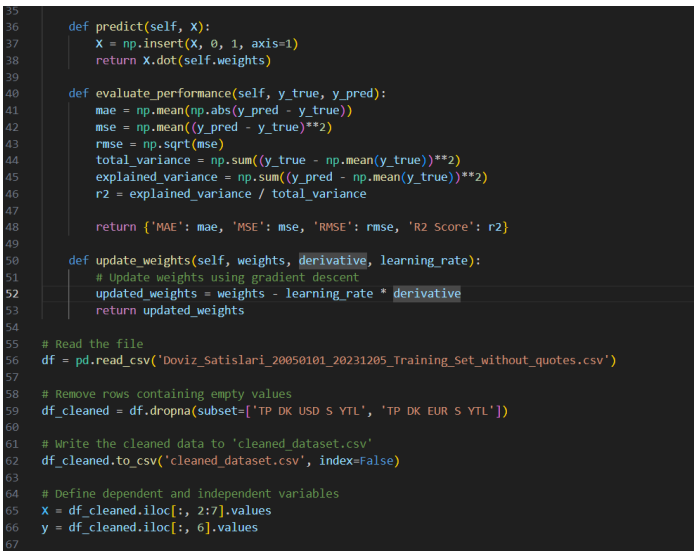
```python
def fit(self, X, y, learning_rate=0.01, epochs=1000):
    X = np.insert(X, 0, 1, axis=1)
    self.weights = np.zeros(X.shape[1])

    for epoch in range(epochs):
        y_pred = X.dot(self.weights)

        # Compute the derivative of MSE with L1 and L2 regularization
        error = y_pred - y
        derivative = (2/len(y)) * (X.T.dot(error) + self.alpha_l2 * self.weights + self.alpha_l1 * np.sign(self.weights))

        # Update weights using gradient descent with L1 and L2 regularization
        old_weights = self.weights.copy()
        self.weights = self.update_weights(self.weights, derivative, learning_rate)

        # Print the derivative of MSE, old and current weights for every 100 epochs
        if epoch % 100 == 0:
            print(f'\nEpoch {epoch}, Derivative of MSE: {derivative}')
            print(f'Old Weights: {old_weights}')
            print(f'Current Weights: {self.weights}')

    # Print the final derivative of MSE and weights after training
    print(f'\nFinal Derivative of MSE: {derivative}')
    print(f'Final Weights: {self.weights}')
```

Figure 2.3.2



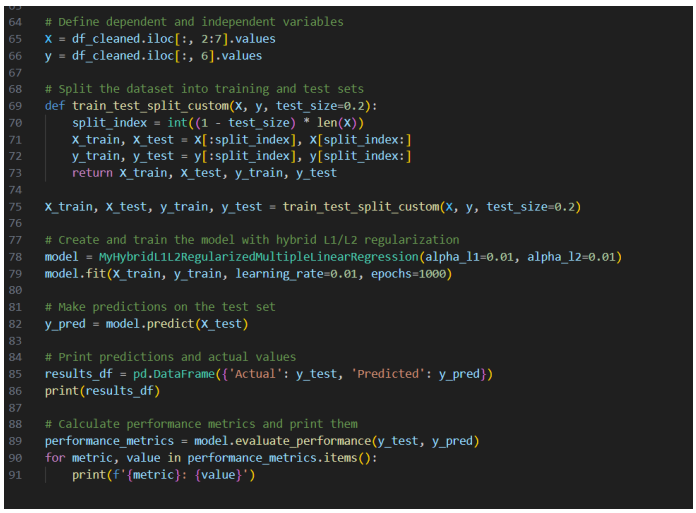
```python
HybridMLinearReg.py > MyHybridL1L2RegularizedMultipleLinearRegression > update_weights
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

class MyHybridL1L2RegularizedMultipleLinearRegression:
    def __init__(self, alpha_l1=0.01, alpha_l2=0.01):
        self.weights = None
        self.alpha_l1 = alpha_l1
        self.alpha_l2 = alpha_l2

    def fit(self, X, y, learning_rate=0.01, epochs=1000):
        X = np.insert(X, 0, 1, axis=1)
        self.weights = np.zeros(X.shape[1])

        for epoch in range(epochs):
            y_pred = X.dot(self.weights)

            # Compute the derivative of MSE with L1 and L2 regularization
            error = y_pred - y
            derivative = (2/len(y)) * (X.T.dot(error) + self.alpha_l2 * self.weights + self.alpha_l1 * np.sign(self.weights))

            # Update weights using gradient descent with L1 and L2 regularization
            old_weights = self.weights.copy()
            self.weights = self.update_weights(self.weights, derivative, learning_rate)

            # Print the derivative of MSE, old and current weights for every 100 epochs
            if epoch % 100 == 0:
                print(f'\nEpoch {epoch}, Derivative of MSE: {derivative}')
                print(f'Old Weights: {old_weights}')
                print(f'Current Weights: {self.weights}')

        # Print the final derivative of MSE and weights after training
        print(f'\nFinal Derivative of MSE: {derivative}')
        print(f'Final Weights: {self.weights}')

    def predict(self, X):
```

Figure 2.3.3



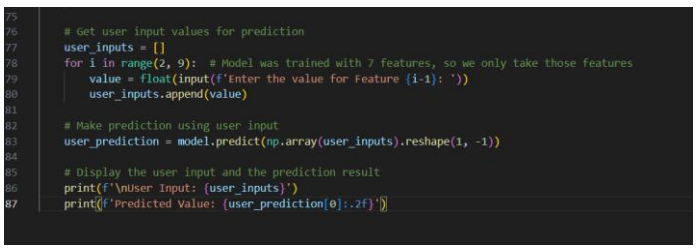
```python
    def predict(self, X):
        X = np.insert(X, 0, 1, axis=1)
        return X.dot(self.weights)

    def evaluate_performance(self, y_true, y_pred):
        mae = np.mean(np.abs(y_pred - y_true))
        mse = np.mean((y_pred - y_true)**2)
        rmse = np.sqrt(mse)
        total_variance = np.sum((y_true - np.mean(y_true))**2)
        explained_variance = np.sum((y_pred - np.mean(y_true))**2)
        r2 = explained_variance / total_variance

        return {'MAE': mae, 'MSE': mse, 'RMSE': rmse, 'R2 Score': r2}

    def update_weights(self, weights, derivative, learning_rate):
        # Update weights using gradient descent
        updated_weights = weights - learning_rate * derivative
        return updated_weights

# Read the file
df = pd.read_csv('Doviz_Satislari_20050101_20231205_Training_Set_without_quotes.csv')

# Remove rows containing empty values
df_cleaned = df.dropna(subset=['TP DK USD S YTL', 'TP DK EUR S YTL'])

# Write the cleaned data to 'cleaned_dataset.csv'
df_cleaned.to_csv('cleaned_dataset.csv', index=False)

# Define dependent and independent variables
X = df_cleaned.iloc[:, 2:7].values
y = df_cleaned.iloc[:, 6].values
```

Figure 2.3.4



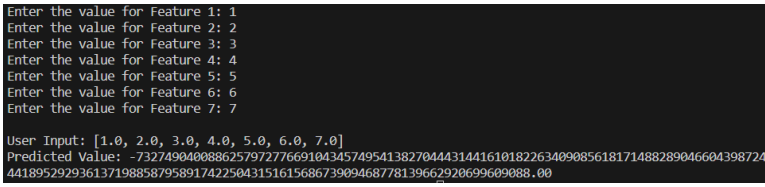
```python
# Define dependent and independent variables
X = df_cleaned.iloc[:, 2:7].values
y = df_cleaned.iloc[:, 6].values

# Split the dataset into training and test sets
def train_test_split_custom(X, y, test_size=0.2):
    split_index = int((1 - test_size) * len(X))
    X_train, X_test = X[:split_index], X[split_index:]
    y_train, y_test = y[:split_index], y[split_index:]
    return X_train, X_test, y_train, y_test

X_train, X_test, y_train, y_test = train_test_split_custom(X, y, test_size=0.2)

# Create and train the model with hybrid L1/L2 regularization
model = MyHybridL1L2RegularizedMultipleLinearRegression(alpha_l1=0.01, alpha_l2=0.01)
model.fit(X_train, y_train, learning_rate=0.01, epochs=1000)

# Make predictions on the test set
y_pred = model.predict(X_test)

# Print predictions and actual values
results_df = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred})
print(results_df)

# Calculate performance metrics and print them
performance_metrics = model.evaluate_performance(y_test, y_pred)
for metric, value in performance_metrics.items():
    print(f'{metric}: {value}')
```

Figure 2.3.5



```python
# Get user input values for prediction
user_inputs = []
for i in range(2, 9):  # Model was trained with 7 features, so we only take those features
    value = float(input(f'Enter the value for Feature {i-1}: '))
    user_inputs.append(value)

# Make prediction using user input
user_prediction = model.predict(np.array(user_inputs).reshape(1, -1))

# Display the user input and the prediction result
print(f'\nUser Input: {user_inputs}')
print(f'Predicted Value: {user_prediction[0]:.2f}')
```

Figure 2.4.1



```
Enter the value for Feature 1: 1
Enter the value for Feature 2: 2
Enter the value for Feature 3: 3
Enter the value for Feature 4: 4
Enter the value for Feature 5: 5
Enter the value for Feature 6: 6
Enter the value for Feature 7: 7

User Input: [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
Predicted Value: -732749040088625797277669104345749541382704443144161018226340908561817148828904660439872
4418952929361371988587958917422504315161568673909468778139662920699609088.00
```

Figure 2.4.2