Programming Machine Learning Lab

Exercise 3

General Instructions:

1. You need to submit the PDF as well as the filled notebook file.
2. Name your submissions by prefixing your matriculation number to the filename. Example, if your MR is 12345 then rename the files as **"12345_Exercise_3.xxx"**
3. Complete all your tasks and then do a clean run before generating the final pdf. (*Clear All Ouputs* and *Run All* commands in Jupyter notebook)

Exercise Specific instructions::

1. You are allowed to use only NumPy and Pandas (unless stated otherwise). You can use any library for visualizations.

Part 1

You are required to optimize the booth function using gradient descent. The Code below provides a visual representation of the booth function in a 3D plot. Mathematically, the function can be defined as follows.

$$f(x, y) = (x + 2y)^2 + (2x + y - 5)^2$$

Your tasks are to:

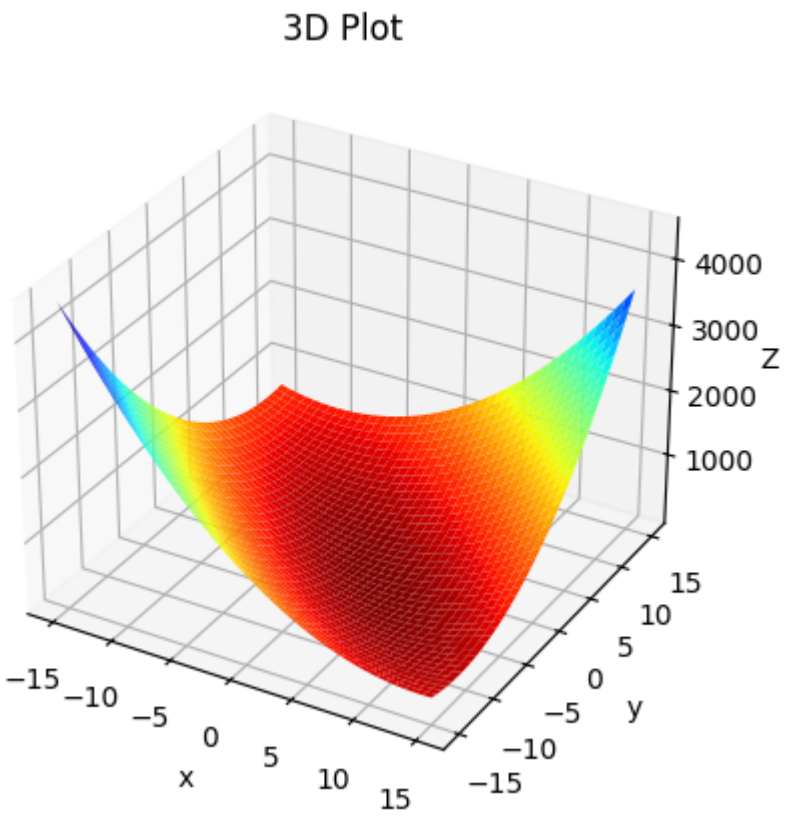
- Add proper axis/figure labels to the 3d plot provided.
- Derive the partial gradients.
- Create a function to perform Gradient Descent on the given equation, the function should take the initial values of (x, y) and α as inputs. Initial value of (x,y) would be (-10,5) and the steplength α needs to be determined through trial and error. Once the α is finalized, plot the value of Z over time as it is minimized using the gradient descent.
- Visualize the trajectory on a 3D plot. This trajectory should ideally lead to the function minimum. Try to plot the trajectory in a for loop so that the path taken is visible.

The GD algorithm is given below, where ϵ is a small float value like 10^{-3} or 10^{-5} and i_{max} is the max number of iterations to perform.

1. $\boldsymbol{\theta}_0 = \text{randomInitialization}()$
2. for $i = 0, \ldots, i_{max}$:
3. $\quad \boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta \nabla L(\boldsymbol{\theta}_i)$
4. $\quad$ if $L(\boldsymbol{\theta}_i) - L(\boldsymbol{\theta}_{i+1}) < \epsilon$:
5. $\quad\quad$ return $\boldsymbol{\theta}_{i+1}$
6. raise Exception("Not converged in i_{max} iterations")

(Note: If you prefer, you can use you a differnt code for plotting the 3d figure)

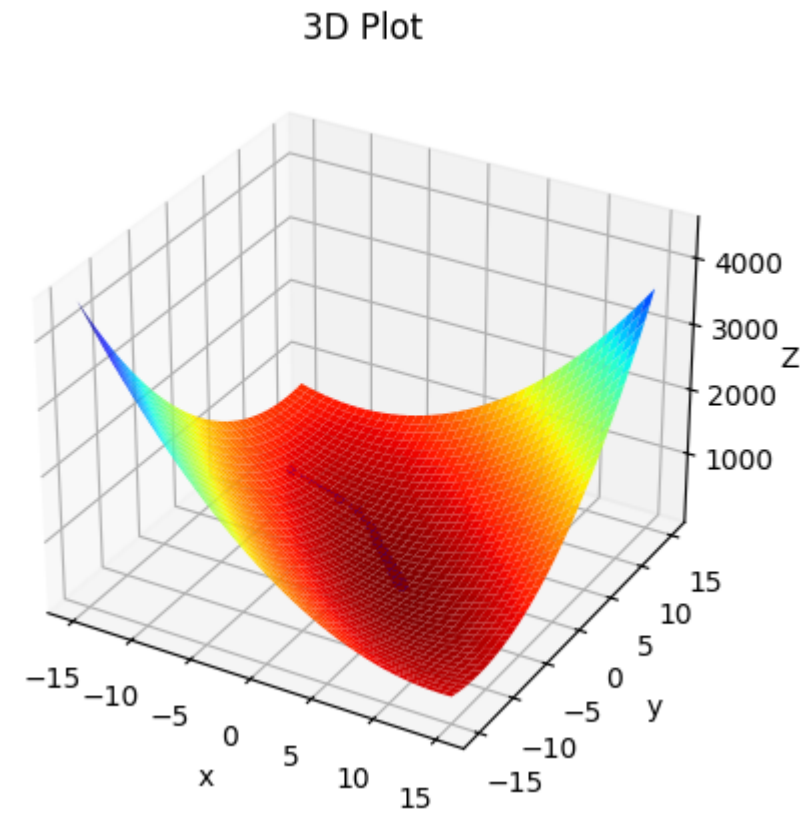
In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
reso = 42
X = np.linspace(-15, 15, reso)
Y = np.linspace(-15, 15, reso)
X, Y = np.meshgrid(X, Y)
Z = (X+2*Y)**2 + (2*X+Y-5)**2

surf = ax.plot_surface(X, Y, Z, cmap='jet_r',
    linewidth=0, antialiased=True)
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel('Z', labelpad=1)
ax.set_title('3D Plot')
plt.show()
```



In [ ]:

```
### write your code here

def f(x, y):
    return (x + 2*y)**2 + (2*x + y - 5)**2

def gradient_f_x(x, y):
    return 2*(x + 2*y) + 2*(2*x + y - 5)*2

def gradient_f_y(x, y):
    return 2*(x + 2*y)*2 + 2*(2*x + y - 5)

def draw_progress(Xs, Ys):
    fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
    reso = 42
    X = np.linspace(-15, 15, reso)
    Y = np.linspace(-15, 15, reso)
    X, Y = np.meshgrid(X, Y)
    Z = (X+2*Y)**2 + (2*X+Y-5)**2
    Zs = f(Xs, Ys)
    surf = ax.plot_surface(X, Y, Z, cmap='jet_r',
        linewidth=0, antialiased=True, label="search space")
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_zlabel('Z', labelpad=1)
    ax.set_title('3D Plot')
    ax.plot(
        Xs, Ys, Zs, '.-b',
        label = 'gradient descent'
    )
    plt.show()
```

```
def gradient_decent_booth(x,y,alpha):
    epochs = 100
    eps = 0.0001
    value_before = f(x, y)
    Xs = [x]
    Ys = [y]
    for i in range(epochs):
        x = x - alpha*gradient_f_x(x,y)
        y = y - alpha*gradient_f_y(x,y)
        value = f(x, y)
        Xs.append(x)
        Ys.append(y)
        if (abs(
            value_before - value
        ) < eps):
            draw_progress(np.array(Xs), np.array(Ys))
            return (x, y)
        value_before = value
    draw_progress(np.array(Xs), np.array(Ys))
    raise Exception(f'Did not converge in {epochs} iterations')

gradient_decent_booth(-10, 5, 0.05)
```



Out[ ]: (3.3205936905126063, -1.6547570507776936)

Part 2

Regression

Implement learn a linear regression model using Normal Equations using the algorithm below.

learn-linreg-NormEq $(D\_train := \{(x_1, y_1), \ldots, (x_N, y_N)\})$:
$X := (x_1, x_2, \ldots, x_N)^T$
$y := (y_1, y_2, \ldots, y_N)^T$
$A := X^T X$
$b := X^T y$
$\hat{\beta} := \text{solve-SLE}(A, b)$
return $\hat{\beta}$

For solving the Linear Equations (the function solve-SLE), implement either:

- Gaussian Elimination (implemented in pure python i.e., only NumPy).
- QR decomposition (implemented in pure python i.e., only NumPy)

(Note: you are not allowed to use numpy.linalg.qr or np.linalg.lstsq for this)

We will apply this function on dummy data to verify correctness of the code, your function should provide beta's close to the original ones.

β = (7.10844396, 6.30577555, 4.26871516, 6.97082463, 1.65779631, 9.14148068, 5.45284651, 4.40990826, 7.87749723, 6.9317578)

In [ ]:
```
## Ground truth beta values
Beta = np.array([7.10844396, 6.30577555, 4.26871516, 6.97082463, 1.65779631,
    9.14148068, 5.45284651, 4.40990826, 7.87749723, 6.9317578 ])

## Setting the seed for ensuring deterministic behaviour
np.random.seed(1408)

X = np.random.normal(loc=2,scale=1,size=(100,10))*np.random.normal(loc=4,scale=0.1,

Y = X@Beta + np.random.normal(loc=1,scale=1,size=(100))
```

In [ ]:
```
### Write your code


def gaussian_elimination(A, b):
    # Augmenting the matrix A with the vector b
    augmented_matrix = np.column_stack((A, b))

    # Performing Gaussian Elimination
    n = len(b)
    for i in range(n):
        # Partial Pivoting
        max_row = np.argmax(np.abs(augmented_matrix[i:, i])) + i
        augmented_matrix[[i, max_row]] = augmented_matrix[[max_row, i]]

        # Elimination
        for j in range(i + 1, n):
```

```
            factor = augmented_matrix[j, i] / augmented_matrix[i, i]
            augmented_matrix[j, i:] -= factor * augmented_matrix[i, i:]

    # Back-substitution
    x = np.zeros(n)
    for i in range(n - 1, -1, -1):
        x[i] = (augmented_matrix[i, -1] - np.dot(augmented_matrix[i, i+1:n], x[i+1:

    return x

def gaussian_elimination2(A, B):
    aug = np.hstack((A, B.reshape(B.size, 1)))
    n = aug.shape[0]
    m = aug.shape[1]
    for i in range(n):
        max_row = max(range(i, n), key=lambda j: abs(aug[j][i]))
        aug[i], aug[max_row] = aug[max_row], aug[i]
        if aug[i,i] != 0:
            for j in range(m):
                aug[i,j] = (aug[i,j] / aug[i,i])
        for k in range(i+1, n):
            factor = aug[k][i]
            for j in range(i, m):
                aug[k][j] -= (factor * aug[i][j])
    return solve_from_REF(aug)

def solve_from_REF(aug):
    n = aug.shape[0]
    m = aug.shape[1]
    solution = np.zeros(n)
    for i in range(n - 1, -1, -1):
        solution[i] = aug[i][m-1]
        for j in range(i + 1, n):
            solution[i] -= aug[i][j] * solution[j]
    return solution

def solve_SLE(x, y):
    x = np.column_stack((x, np.ones(len(x))))
    A = x.T @ x
    B = x.T @ y
    sol = gaussian_elimination(A, B)
    return sol[:len(sol)-1]

betha_star = solve_SLE(X, Y)
print(betha_star)
```

```
[7.1087686  6.3154711  4.29642522 6.97030561 1.69272245 9.12520527
 5.41588803 4.42472208 7.8469587  6.85139092]
```

Evaluation

1. Compare your Learned Beta values to the ground truth beta values. A visual comparison is required.
2. Use the numpy implementation of linear equation solver (np.linalg.lstsq) as solve-SLE and compare the results to your Learned Beta values.

In [ ]:
```
MSE = np.mean(np.sum(Beta - betha_star)**2)
print(MSE)
```

```
0.005958003290036681
```

In [ ]:
```
### Write your code here
# MSE = np.mean(np.sum(Beta - betha_star)**2)
# print(MSE)

def solving_using_np(x, y):
    A = x.T @ x
    B = x.T @ y
    sol = np.linalg.lstsq(A, B, rcond=None)[0]
    return sol

sol = solving_using_np(X, Y)
print(sol)
```

```
[7.11902953 6.33663098 4.3042112  6.97657093 1.71656095 9.13381132
 5.42917403 4.430442   7.86111635 6.86502871]
```

Part 3

1. **Time Series Exploration**

Try to understand the **"time series.csv"** dataset uploaded along with the exercise. This involves (but is not limited to) plotting the multivariate time series (time on x-axis) clearly labelled and formatted, understanding how the multivariate time series interacts, understanding the correlation between the different variables, plots for variable density functions, identifying inherent seasonality or trend etc.

(Note: if the different variable are of different scales, it would be better to plot them on different subplots)

2. **Train/Test split**

As a next step, try to split the data into train and test on 2017-10-24. All days till 2017-10-24 would be in train dataset and the rest would be in test. Draw plots for both time series with appropriate labels.

3. **Scaling data**

Perform a standard scaling to scale each variable independently i.e.,

$$z = \frac{(x - \text{mean}(x))}{\text{std}(x)}$$

The scaling needs to be implement to both train and test data, however the mean and std are only calculated from training data. The training mean and std are then used to scale the

test data as well. Explain why only the train dataset should be used to calculate the scaling factors.

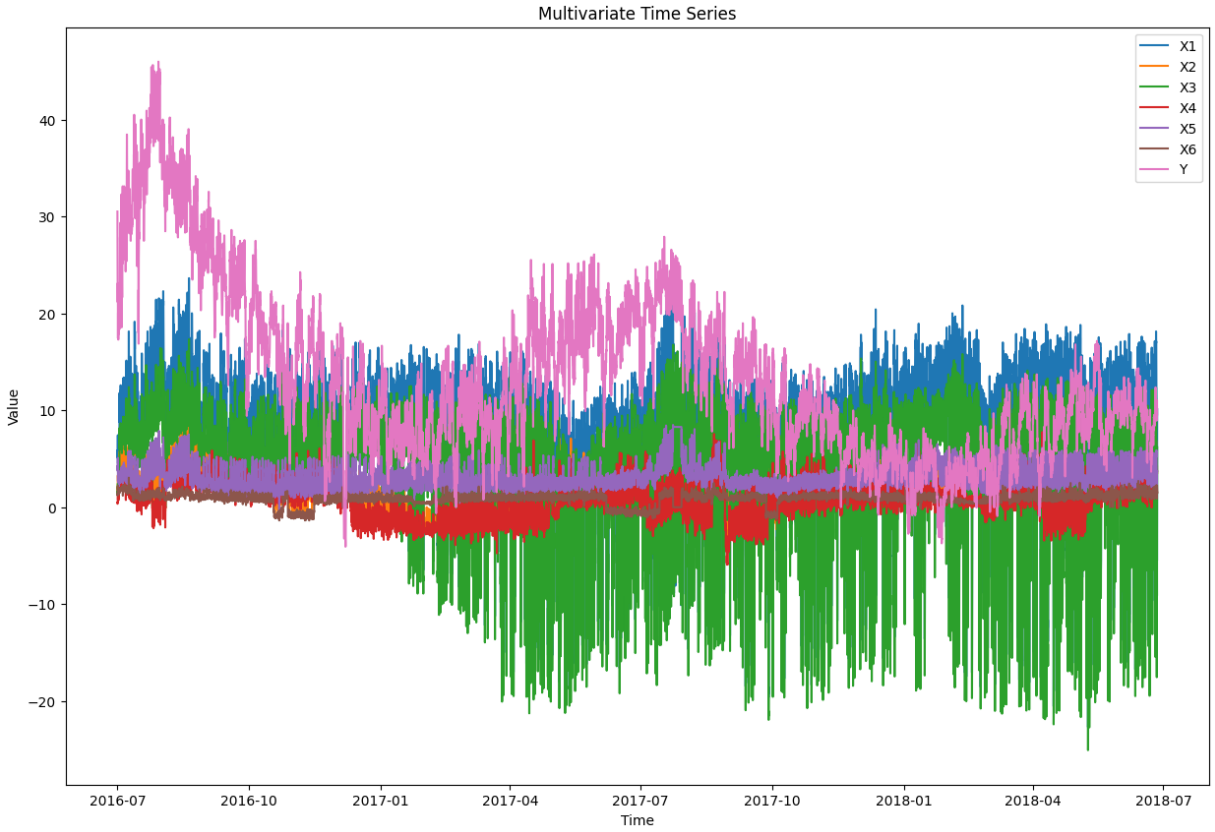
4. **Perform Regression**

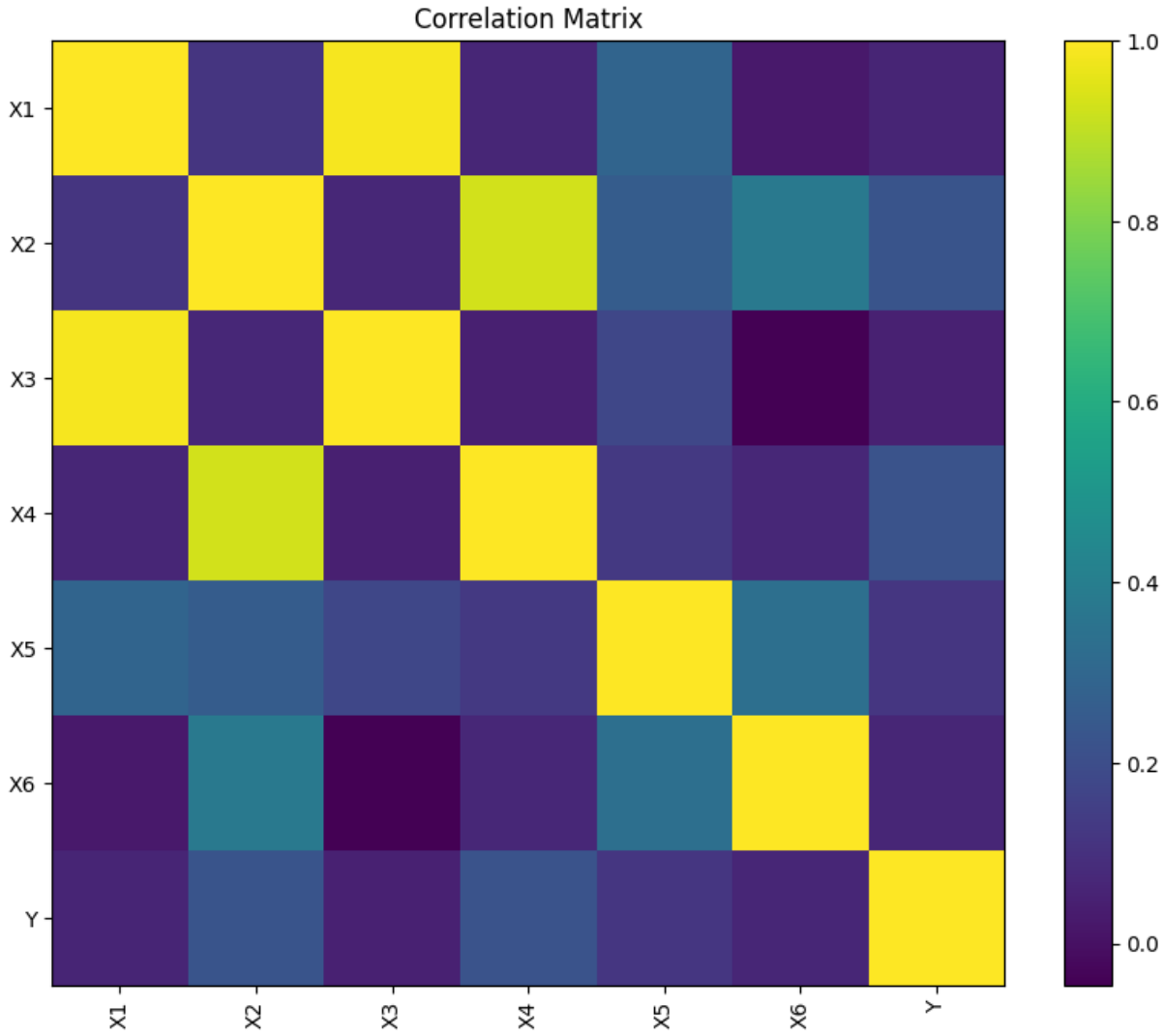
Use the **learn-linreg-NormEq** function form Part 2 here to calculate the beta values using the train dataset.

5. **Evaluation**

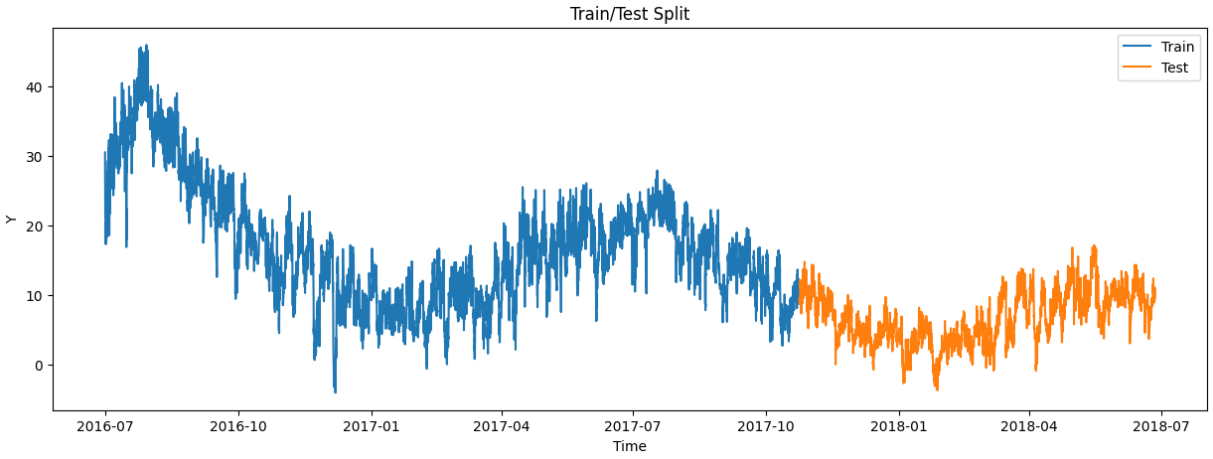
Use the learned parameters ($\hat{\beta}$) to calculate the Mean Absolute Error (MAE) on train and test split for the dataset. Additionally, plot the true target and the predicted target for the test split as a time-series.

In [ ]:
```
### Write your code here
import pandas as pd
import matplotlib.pyplot as plt

# Load the dataset
data = pd.read_csv('time_series.csv', parse_dates=['date'], index_col='date')

# Plot multivariate time series
plt.figure(figsize=(15, 10))
for col in data.columns:
    plt.plot(data.index, data[col], label=col)

plt.xlabel('Time')
plt.ylabel('Value')
plt.title('Multivariate Time Series')
plt.legend()
plt.show()

# Understanding correlation
correlation_matrix = data.corr()
plt.figure(figsize=(10, 8))
plt.imshow(correlation_matrix, cmap='viridis', interpolation='none', aspect='auto')
plt.colorbar()
plt.xticks(range(len(correlation_matrix)), correlation_matrix.columns, rotation='ve
plt.yticks(range(len(correlation_matrix)), correlation_matrix.columns)
plt.title('Correlation Matrix')
plt.show()
```


Multivariate Time Series
X1
X2
X3
X4
X5
X6
Y
Value
40
30
20
10
0
−10
−20
2016-07
2016-10
2017-01
2017-04
2017-07
2017-10
2018-01
2018-04
2018-07
Time



In [ ]:
```
# Split data into train and test
train_data = data[data.index <= '2017-10-24']
test_data = data[data.index > '2017-10-24']

# Plot time series for train and test
plt.figure(figsize=(15, 5))
plt.plot(train_data.index, train_data['Y'], label='Train')
plt.plot(test_data.index, test_data['Y'], label='Test')
plt.xlabel('Time')
plt.ylabel('Y')
plt.title('Train/Test Split')
plt.legend()
plt.show()
```



In [ ]:
```
# we use only test data mean and std because they are the only data that model have
# Calculate mean and standard deviation for each feature in the training set
mean_train = train_data.mean()
std_train = train_data.std()

# Manually scale the training set
train_scaled = (train_data - mean_train) / std_train

# Manually scale the test set using mean and std from the training set
test_scaled = (test_data - mean_train) / std_train

train_scaled_df = pd.DataFrame(train_scaled, columns=data.columns, index=train_data
test_scaled_df = pd.DataFrame(test_scaled, columns=data.columns, index=test_data.in
```

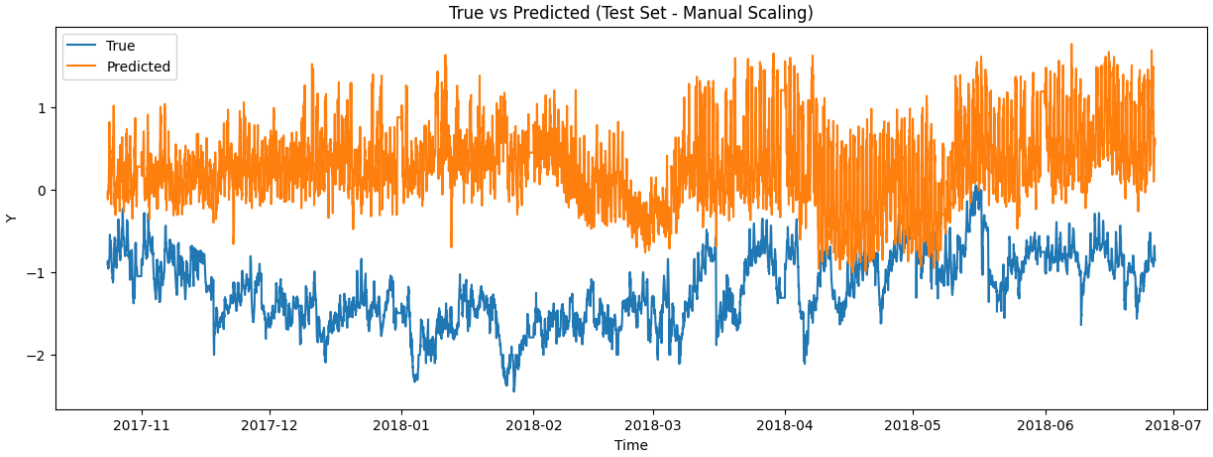
In [ ]:
```
X_train = train_scaled_df.drop(columns='Y')
y_train = train_scaled_df['Y']
X_test = test_scaled_df.drop(columns='Y')
y_test = test_scaled_df['Y']

# Perform linear regression using Normal Equations
beta_hat = solving_using_np(X_train.values, y_train.values)
print(beta_hat)
```

```
[ 0.43888442  0.1544392  -0.38010452  0.29573298  0.1903921   0.07856963]
```

In [ ]:
```
# Predict on the training set using normalized values
y_train_pred_normalized = np.dot(X_train.values, beta_hat)

# Calculate Mean Squared Error on the training set using normalized values
mse_train_normalized = np.mean((y_train.values - y_train_pred_normalized)**2)

# Predict on the test set using normalized values
y_test_pred_normalized = np.dot(X_test.values, beta_hat)

# Calculate Mean Squared Error on the test set using normalized values
mse_test_normalized = np.mean((y_test.values - y_test_pred_normalized)**2)

# Print MSE on both sets
print(f'MSE on Train Set (Normalized): {mse_train_normalized:.2f}')
print(f'MSE on Test Set (Normalized): {mse_test_normalized:.2f}')
```

```
# Plot true target and predicted target for the test set
plt.figure(figsize=(15, 5))
plt.plot(test_data.index, y_test, label='True')
plt.plot(test_data.index, y_test_pred_normalized, label='Predicted')
plt.xlabel('Time')
plt.ylabel('Y')
plt.title('True vs Predicted (Test Set - Manual Scaling)')
plt.legend()
plt.show()
```

```
MSE on Train Set (Normalized): 0.68
MSE on Test Set (Normalized): 2.69
```



In [ ]: