

Assignment 1: Least Squares

Name: Barun Parua

Roll Number: 21CS10014

Github Link: <https://github.com/Baron105/ML-Assignments> for the Notebook and Reports. Please refer to AI61003/Assignment folder for assignment related files.

Imports

Here is the list of all the libraries used in this assignment.

```
In [ ]: # imports

# torchvision for MNIST dataset
from torchvision.datasets import MNIST

# numpy for mathematical operations
import numpy as np
# matplotlib for plotting graphs and images
import matplotlib.pyplot as plt
# pandas for data Loading and printing the confusion matrix
import pandas as pd
```

MNIST Dataset

Here, the MNIST dataset is downloaded and stored separately for training and test data, hence no further splitting is required in questions 1 and 2.

```
In [ ]: # fetching and Loading the MNIST dataset
mnist_data_train = MNIST('/', train=True, download=True)
mnist_data_test = MNIST('/', train=False, download=True)
```

Least Squares Function

Main function of the assignment.

This is the function that calculates the least squares solution for a given matrix A and vector b. It returns the vector x that minimizes the 2-norm of Ax-b.

x is based on the formula: $x = \text{inv}(A.T @ A) @ A.T @ b$

Note that here I have used the @ operator to denote matrix multiplication. Also, the transpose of a matrix A is denoted by A.T. Finally, inv(A) denotes the inverse of a matrix A. Here I have used the pinv function from numpy.linalg to calculate the pseudo-inverse of A, rather than the inverse of A so that the function can handle non-square and singular matrices.

A corresponding function for regularized least squares is also defined, which takes in an additional parameter lambda_. The matrix is then stacked with $\sqrt{\lambda}$ times the identity matrix, while the vector is stacked with zeros. The rest of the function remains the same.

```
In [ ]: def least_squares(A, b):
    x = np.linalg.pinv(A.T @ A) @ A.T @ b
    return x

def least_squares_regularized(A, b, lambda_):
    A = np.concatenate((A, np.sqrt(lambda_) * np.identity(A.shape[1])), axis=0)
    b = np.concatenate((b, np.zeros(A.shape[1])), axis=0)
    x = np.linalg.pinv(A.T @ A) @ A.T @ b
    return x
```

Question 1

The MNIST dataset is loaded already above. Now, we have to filter the dataset to only include classes 1 and 7 and assign them labels 1 and -1 respectively. After this the images are flattened from matrices of size 28x28 to vectors of length 784. Then, using the least squares function we

train the model on train set and report the accuracy and the confusion matrix on the test set.

It can be seen that the accuracy is very high, and the confusion matrix shows that the model is able to distinguish between 1 and 7 very well.

```
In [ ]: # function to filter the dataset to only include 1s and 7s as per the assignment
def filter_dataset(dataset):
    indices = (dataset.targets == 1) | (dataset.targets == 7)
    images = dataset.data[indices]
    labels = dataset.targets[indices]
    return images, labels

print("Shapes of the train and test data before filtering:", mnist_data_train.data.shape, mnist_data_test.data.shape)

# applying the filter to get the test and train data
X_train, y_train = filter_dataset(mnist_data_train)
X_test, y_test = filter_dataset(mnist_data_test)

# assigning the Labels to 1 if class is 1 and -1 if class is 7
y_train = np.where(y_train == 1, 1, -1)
y_test = np.where(y_test == 1, 1, -1)

# making X_train and X_test numpy arrays
X_train = np.array(X_train)
X_test = np.array(X_test)

print(f"Shapes of X_train and X_test before flattening: {X_train.shape}, {X_test.shape}")

# vectorizing the images
X_train = X_train.reshape(-1, 28 * 28) / 255.0
X_test = X_test.reshape(-1, 28 * 28) / 255.0
print(f"Shapes of X_train and X_test after flattening: {X_train.shape}, {X_test.shape}\n")

# adding a column of 1s to the data for the bias term
X_train = np.hstack((np.ones((X_train.shape[0], 1)), X_train))
X_test = np.hstack((np.ones((X_test.shape[0], 1)), X_test))

# training the model to get the weights
weights = least_squares(X_train, y_train)

# predicting the labels for the test data
```

```

y_pred = np.sign(X_test @ weights)

# reporting the accuracy of the model
accuracy = np.mean(y_pred == y_test)
print(f"Accuracy: {accuracy:.4f}\n\nConfusion Matrix:")

# plotting the confusion matrix
confusion_matrix = pd.crosstab(y_test, y_pred, rownames=['Actual'], colnames=['Predicted'])
print(confusion_matrix)

```

Shapes of the train and test data before filtering: torch.Size([60000, 28, 28]) torch.Size([10000, 28, 28])

Shapes of X_train and X_test before flattening: (13007, 28, 28), (2163, 28, 28)

Shapes of X_train and X_test after flattening: (13007, 784), (2163, 784)

Accuracy: 0.9894

Confusion Matrix:

	-1.0	1.0
Actual	1007	21
-1	2	1133
1		

Question 2

The MNIST dataset is loaded already above. Now, the images are flattened from matrices of size 28x28 to vectors of length 784.

For each class, we have to train a separate model. For this, we select samples of the current class as positive samples and randomly select equal number of samples from the other classes as negative samples to make the dataset balanced. Now using the least squares function we train the model on train set for each of the classes. Auxiliary functions are used so that repeated code is not written.

Now, we have to predict the class of each test sample by using the model that gives the highest score of prediction for that sample and assigning the sample to that class. Finally we report the accuracy and the confusion matrix on the test set.

```

In [ ]: # define the training and test sets as numpy arrays
X_train, y_train = np.array(mnist_data_train.data), np.array(mnist_data_train.targets)
X_test, y_test = np.array(mnist_data_test.data), np.array(mnist_data_test.targets)

# vectorizing the images

```

```
X_train = X_train.reshape(-1, 28 * 28) / 255.0
X_test = X_test.reshape(-1, 28 * 28) / 255.0

# function to filter the dataset to only include class i images for i in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
def filter_dataset(train_x, train_y, class_label):
    indices = (train_y == class_label)
    images = train_x[indices]
    labels = train_y[indices]
    return images, labels

# function to select all images except those of class i for i in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
def filter_dataset_not(train_x, train_y, class_label):
    indices = (train_y != class_label)
    images = train_x[indices]
    labels = train_y[indices]
    return images, labels

# function to train the model for a given class label
def train_wrt_class(class_label):
    # class_label is the label of the class for which the model is being trained
    # take the images of class and randomly sample the same number of images from the rest of the dataset
    X_train_0, y_train_0 = filter_dataset(X_train, y_train, class_label)
    X_train_0_not, y_train_0_not = filter_dataset_not(X_train, y_train, class_label)

    # randomly sample the same number of images from the rest of the dataset
    indices = np.random.choice(np.arange(len(X_train_0_not)), len(X_train_0))
    X_train_0_not = X_train_0_not[indices]
    y_train_0_not = y_train_0_not[indices]

    # add the images of class 0 and the randomly sampled images from the rest of the dataset
    X_train_0 = np.concatenate((X_train_0, X_train_0_not))
    y_train_0 = np.concatenate((y_train_0, y_train_0_not))

    print(f"Shapes of X_train_0 and y_train_0 after filtering and sampling: {X_train_0.shape}, {y_train_0.shape}")

    # assign the labels to 1 if class is class_label and -1 if class is not class_label
    y_train_0 = np.where(y_train_0 == class_label, 1, -1)
    y_test_0 = np.where(y_test == class_label, 1, -1)

    # add a column of 1s to the data for the bias term
    X_train_0 = np.hstack((np.ones((X_train_0.shape[0], 1)), X_train_0))
```

```
X_test_0 = np.hstack((np.ones((X_test.shape[0], 1)), X_test))

# train the model to get the weights
weights_0 = least_squares(X_train_0, y_train_0)

# predict the labels for the test data
y_pred_0 = np.sign(X_test_0 @ weights_0)

# report the accuracy of the model
accuracy_0 = np.mean(y_pred_0 == y_test_0)
print(f"\nAccuracy for class {class_label}: {accuracy_0:.4f}\n\nConfusion Matrix for class {class_label}:")

# plot the confusion matrix
confusion_matrix_0 = pd.crosstab(y_test_0, y_pred_0, rownames=['Actual'], colnames=['Predicted'])
print(confusion_matrix_0)

return weights_0
```

Class 0

Training of the model for class 0.

```
In [ ]: # class 0
weights_0 = train_wrt_class(0)

Shapes of X_train_0 and y_train_0 after filtering and sampling: (11846, 784), (11846,)

Accuracy for class 0: 0.9751

Confusion Matrix for class 0:
Predicted -1.0  1.0
Actual
-1        8781  239
 1         10   970
```

Class 1

Training of the model for class 1.

```
In [ ]: # class 1  
weights_1 = train_wrt_class(1)  
  
Shapes of X_train_0 and y_train_0 after filtering and sampling: (13484, 784), (13484,)  
  
Accuracy for class 1: 0.9669  
  
Confusion Matrix for class 1:  
Predicted -1.0 1.0  
Actual  
-1 8545 320  
1 11 1124
```

Class 2

Training of the model for class 2.

```
In [ ]: # class 2  
weights_2 = train_wrt_class(2)  
  
Shapes of X_train_0 and y_train_0 after filtering and sampling: (11916, 784), (11916,)  
  
Accuracy for class 2: 0.9508  
  
Confusion Matrix for class 2:  
Predicted -1.0 1.0  
Actual  
-1 8559 409  
1 83 949
```

Class 3

Training of the model for class 3.

```
In [ ]: # class 3  
weights_3 = train_wrt_class(3)
```

```
Shapes of X_train_0 and y_train_0 after filtering and sampling: (12262, 784), (12262,)
```

```
Accuracy for class 3: 0.9256
```

```
Confusion Matrix for class 3:
```

	Predicted -1.0	Predicted 1.0
Actual -1	8305	685
Actual 1	59	951

Class 4

Training of the model for class 4.

```
In [ ]: # class 4  
weights_4 = train_wrt_class(4)
```

```
Shapes of X_train_0 and y_train_0 after filtering and sampling: (11684, 784), (11684,)
```

```
Accuracy for class 4: 0.9384
```

```
Confusion Matrix for class 4:
```

	Predicted -1.0	Predicted 1.0
Actual -1	8428	590
Actual 1	26	956

Class 5

Training of the model for class 5.

```
In [ ]: # class 5  
weights_5 = train_wrt_class(5)
```

```
Shapes of X_train_0 and y_train_0 after filtering and sampling: (10842, 784), (10842,)
```

```
Accuracy for class 5: 0.9043
```

```
Confusion Matrix for class 5:
```

	Predicted -1.0	Predicted 1.0
Actual -1	8197	911
Actual 1	46	846

Class 6

Training of the model for class 6.

```
In [ ]: # class 6  
weights_6 = train_wrt_class(6)
```

```
Shapes of X_train_0 and y_train_0 after filtering and sampling: (11836, 784), (11836,)
```

```
Accuracy for class 6: 0.9624
```

```
Confusion Matrix for class 6:
```

	Predicted -1.0	Predicted 1.0
Actual -1	8695	347
Actual 1	29	929

Class 7

Training of the model for class 7.

```
In [ ]: # class 7  
weights_7 = train_wrt_class(7)
```

```
Shapes of X_train_0 and y_train_0 after filtering and sampling: (12530, 784), (12530,)
```

```
Accuracy for class 7: 0.9526
```

```
Confusion Matrix for class 7:
```

	Predicted -1.0	Predicted 1.0
Actual -1	8547	425
Actual 1	49	979

Class 8

Training of the model for class 8.

```
In [ ]: # class 8  
weights_8 = train_wrt_class(8)
```

```
Shapes of X_train_0 and y_train_0 after filtering and sampling: (11702, 784), (11702,)
```

```
Accuracy for class 8: 0.9009
```

```
Confusion Matrix for class 8:
```

	Predicted -1.0	Predicted 1.0
Actual -1	8134	892
Actual 1	99	875

Class 9

Training of the model for class 9.

```
In [ ]: # class 9  
weights_9 = train_wrt_class(9)
```

```
Shapes of X_train_0 and y_train_0 after filtering and sampling: (11898, 784), (11898,)
```

```
Accuracy for class 9: 0.8943
```

```
Confusion Matrix for class 9:
```

	Predicted -1.0	Predicted 1.0
Actual -1	8004	987
Actual 1	70	939

Using the models to get the confidence scores and assigning the class with highest score

Now, we have to predict the class of each test sample by using the model that gives the highest score of prediction for that sample and assigning the sample to that class. Finally we report the accuracy and the confusion matrix on the test set.

```
In [ ]: # function to predict the class of an image by calculating the confidence scores for each class
def predict_class(image):
    # add a column of 1s to the image for the bias term
    image = np.hstack((1, image))
    confidence_scores = np.array([image @ weights_0, image @ weights_1, image @ weights_2, image @ weights_3, image @ weights_4])

    # return the class with the highest confidence score
    return np.argmax(confidence_scores)

# predicting the classes of the test images
y_pred = np.array([predict_class(image) for image in X_test])

# reporting the accuracy of the model
accuracy = np.mean(y_pred == y_test)
print(f"Accuracy of the model: {accuracy:.4f}\n\nConfusion Matrix:")

# plotting the confusion matrix
confusion_matrix = pd.crosstab(y_test, y_pred, rownames=['Actual'], colnames=['Predicted'])
print(confusion_matrix)
```

Accuracy of the model: 0.8811

Confusion Matrix:

Predicted \ Actual	0	1	2	3	4	5	6	7	8	9
0	924	1	1	1	3	24	15	3	6	2
1	0	1098	3	3	1	4	4	0	22	0
2	12	28	834	30	22	12	14	15	56	9
3	7	3	18	875	3	47	7	10	24	16
4	1	8	3	4	898	6	6	1	6	49
5	12	11	0	39	8	755	17	5	26	19
6	8	8	5	12	22	36	849	0	17	1
7	4	24	18	8	20	11	5	862	2	74
8	6	22	6	21	14	35	11	9	828	22
9	11	8	4	8	47	8	0	21	14	888

Question 3

Here first of all, we generate two random vectors p and q of length 100 sampled from uniform random distribution between [-10,10].

Then the target variable b is defined as $b_i = +1$ if $p_i * q_i > 1$ and $b_i = -1$ otherwise, for $i = 1, 2, \dots, 100$.

Now we generate 6 basis functions as provided. Then we generate the matrix A of basis functions.

Now, we use the least squares method to find the optimal weights vector x. The optimal weights vector x is given by the formula $(A^T A)^{-1} A^T b$, where A is the matrix of basis functions and b is the target variable vector.

We get the weight vector x in this way and report x as well as the accuracy of the model.

```
In [ ]: # generate vectors p and q
p = np.random.uniform(-10, 10, 100)
q = np.random.uniform(-10, 10, 100)

# generate the target variable b
b = np.where(p*q > 1, 1, -1)

# defining the 6 basis functions
f0 = np.ones(100)
```

```

f1 = p
f2 = q
f3 = p**2
f4 = q**2
f5 = p * q

# creating matrix A from the basis functions
A = np.column_stack((f0, f1, f2, f3, f4, f5))

# calling the least squares function to calculate the coefficient vector x
x = least_squares(A, b)
print(f'Coefficient vector x: {x}')

# calculating the predicted values
y_pred = np.sign(A @ x)
accuracy = np.mean(y_pred == b)
print(f'Accuracy: {accuracy}')

```

Coefficient vector x: [-0.19582371 -0.00420142 -0.00624551 0.00215698 0.00141465 0.02263778]
 Accuracy: 0.9

Question 4

Here first of all, we generate two random vectors p and q of length 100 sampled from uniform random distribution between [-1,1].

Then the target variable b is defined as pq + p^2 + q^2.

Now we generate 6 basis functions as provided. Then we generate the matrix A of basis functions.

Now, we use the least squares method to find the optimal weights vector x. The optimal weights vector x is given by the formula $(A^T A)^{-1} A^T b$, where A is the matrix of basis functions and b is the target variable vector.

As we can see very easily that the target function is a linear combination of the basis functions, the mean squared error is very close to zero and the weights vector x is very close to [0,0,0,1,1,1] as expected.

```
In [ ]: # generate vectors p and q
p = np.random.uniform(-1, 1, 100)
q = np.random.uniform(-1, 1, 100)
```

```
# generate the target variable b
b = p * q + p**2 + q**2

# defining the 6 basis functions
f0 = np.ones(100)
f1 = p
f2 = q
f3 = p**2
f4 = q**2
f5 = p * q

# creating matrix A from the basis functions
A = np.column_stack((f0, f1, f2, f3, f4, f5))

# calling the Least squares function to calculate the coefficient vector x
x = least_squares(A, b)

# calculating the mse and printing the results
mse = np.mean((A @ x - b)**2)
print("Coefficient vector x:", x)
print("Mean Squared Error (MSE):", mse)
```

```
Coefficient vector x: [1.44328993e-15 5.55111512e-17 4.16333634e-16 1.00000000e+00
1.00000000e+00 1.00000000e+00]
Mean Squared Error (MSE): 7.539364647584204e-30
```

Question 5

Triangular Basis Function

In this question we deal with the triangular basis function. The function to generate the triangular basis function is defined below, so as to reduce repeated code and make the code more readable.

p is again a vector of length 100 sampled from uniform random distribution between [0,1]. b is defined as $b = 7p - 3p^2$ and we have to generate the coefficients of the triangular basis function.

```
In [ ]: # generate random vector p of size 100 ranging from 0 to 1
p = np.random.uniform(0, 1, 100)

# define b = 7p - 3p^2
b = 7*p - 3*(p**2)

# basis function matrix generating function
def gen_basis_matrix(p, c):
    A = [[0 for _ in range(c)] for _ in range(100)]
    for i in range(100):
        for j in range(c):
            # if p[i] is in the interval [j/c, (j+1)/c), then A[i][j] = (2j+1)/(2c) - |p[i] - (2j+1)/(2c)|
            # else A[i][j] = 0
            if j / c <= p[i] and p[i] < (j + 1) / c:
                A[i][j] = ((2 * j + 1) / (2 * c) - np.abs(p[i] - (2 * j + 1) / (2 * c))) * 2 * c
            else:
                A[i][j] = 0
    return A
```

Part A (Normal Least Squares)

Normal least squares method is used to find the optimal weights vector x . The optimal weights vector x is given by the formula $(A^T A)^{-1} A^T b$, where A is the matrix of basis functions and b is the target variable vector. The coefficient vector x and the mean squared error are reported.

$n = 10$

```
In [ ]: # number of basis functions = 10
A = np.array(gen_basis_matrix(p, 10))

# Least squares solution
x = least_squares(A, b)

# calculate the mse and print the results
mse = np.mean((A @ x - b)**2)
print("Results for 10 basis functions:")
```

```
print("Coefficient vector x:", x)
print("Mean Squared Error (MSE):", mse)
```

Results for 10 basis functions:

```
Coefficient vector x: [0.53071111 0.3607081  0.36154405 0.31365102 0.30271326 0.27601811
0.27326449 0.24342083 0.22862182 0.21268347]
```

```
Mean Squared Error (MSE): 0.02821275956091188
```

n = 50

```
In [ ]: # number of basis functions = 50
A = np.array(gen_basis_matrix(p, 50))

# Least squares solution
x = least_squares(A, b)

# calculate the mse and print the results
mse = np.mean((A @ x - b)**2)
print("Results for 50 basis functions:")
print("Coefficient vector x:", x)
print("Mean Squared Error (MSE):", mse)
```

Results for 50 basis functions:

```
Coefficient vector x: [0.12019467 0.12087572 0.08395354 0.0734853  0.07274823 0.06789204
0.06638316 0.06651617 0.        0.06658391 0.06386817 0.
0.06390027 0.06423656 0.06231816 0.06109013 0.0609512  0.05964094
0.05891572 0.0594205  0.05771872 0.05723884 0.05690927 0.0562588
0.05591247 0.05604779 0.05506206 0.05360234 0.        0.05256495
0.        0.05122908 0.        0.05099484 0.05005479 0.04898657
0.04827662 0.04770154 0.04697906 0.04659594 0.04607065 0.04531666
0.04489111 0.        0.04367732 0.04273106 0.04250461 0.04171692
0.04109652 0.04045242]
```

```
Mean Squared Error (MSE): 0.0005012488497474136
```

Part B (Regularized Least Squares)

Here we do the same thing as in part A, but using regularized least squares instead of least squares. The lambda_ parameter is randomly sampled from a uniform distribution between [0,0.2].

The values of n in this cases are 150 and 200. The least squares regularized method is used to find the optimal weights vector x. The optimal weights vector x is given by the formula $(A^T A + \lambda I)^{-1} A^T b$, where A is the matrix of basis functions and b is the target variable vector. Note that here x is commented out as it takes too much space to print and only the mse is reported. If required, please uncomment the print statement to see the value of x.

n = 150

```
In [ ]: # number of basis functions = 150
A = np.array(gen_basis_matrix(p, 150))

# pick 5 random values of Lambda from 0 to 0.2
lambdas = np.random.uniform(0, 0.2, 5)
for lambda_ in lambdas:
    # Least squares regularized solution
    x = least_squares_regularized(A, b, lambda_)

    # calculate the mse and print the results
    mse = np.mean((A @ x - b)**2)
    print(f"Results for 150 basis functions and lambda = {lambda_}:")
    # print("Coefficient vector x:", x)
    print("Mean Squared Error (MSE):", mse)
    print()
```

Results for 150 basis functions and lambda = 0.08085823988890153:

Mean Squared Error (MSE): 1.7901231776072764e-05

Results for 150 basis functions and lambda = 0.09241014685447242:

Mean Squared Error (MSE): 1.7903562938248887e-05

Results for 150 basis functions and lambda = 0.14698953333144302:

Mean Squared Error (MSE): 1.7918628585235414e-05

Results for 150 basis functions and lambda = 0.1647082076923416:

Mean Squared Error (MSE): 1.7924934499000023e-05

Results for 150 basis functions and lambda = 0.07877177577154944:

Mean Squared Error (MSE): 1.790084309354731e-05

n = 200

```
In [ ]: # number of basis functions = 200
A = np.array(gen_basis_matrix(p, 200))

# pick 5 random values of Lambda from 0 to 0.2
lambdas = np.random.uniform(0, 0.2, 5)
for lambda_ in lambdas:
    # Least squares regularized solution
    x = least_squares_regularized(A, b, lambda_)

    # calculate the mse and print the results
    mse = np.mean((A @ x - b)**2)
    print(f"Results for 200 basis functions and lambda = {lambda_}:")
    # print("Coefficient vector x:", x)
    print("Mean Squared Error (MSE):", mse)
    print()
```

Results for 200 basis functions and lambda = 0.04508453844221956:

Mean Squared Error (MSE): 8.689569951041144e-06

Mean Squared Error (MSE): 8.689569951041144e-06

Results for 200 basis functions and lambda = 0.17145078757001347:

Mean Squared Error (MSE): 8.695470584804921e-06

Results for 200 basis functions and lambda = 0.07668703474305001:

Mean Squared Error (MSE): 8.690402664809776e-06

Results for 200 basis functions and lambda = 0.023020961400530938:

Mean Squared Error (MSE): 8.689244280638514e-06

Results for 200 basis functions and lambda = 0.01985400795095431:

Mean Squared Error (MSE): 8.689214828383487e-06

In general it can be seen that the mean squared error decreases as we increase the value of n, which is expected as the model becomes more complex and is able to fit the data better.

Question 6

Here we have a csv file with various kinds of data. However according to the assignment, we only have to consider the consumption column as the data and build a auto-regressive model for it. We take the last 8 values of the consumption column as the lagged values and the current value as the target variable.

This way, we build the matrix A of data points and the vector b of target variables. Then we use the least squares method to find the optimal weights vector x. As we know that the last 8 values of the consumption column are the lagged values, we can use the last 8 values of the weights vector x to predict the next value of the consumption column.

Finally, the weights vector x and the next day's predicted value of the consumption column are reported.

```
In [ ]: # Loading the dataset
data = pd.read_csv('q6.csv')

# extracting the consumption column and converting it to a numpy array
y = np.array(data['Consumption'])
print("Mean of Consumption:", np.mean(y))
# print("Shape of Consumption:", y.shape)

# creating a new dataset where lag_1 to lag_8 are the features and the next value of consumption is the target variable
X = np.zeros((len(y) - 8, 8))
for i in range(len(y) - 8):
    X[i] = y[i:i+8]
print("Lagged data:", X)
y = y[8:]
print("Target variable:", y)

# fit the model using least squares
weights = least_squares(X, y)

# y_new is the last 8 values of the consumption column
# y_pred is the predicted value for the next day
y_new = y[-8:]
y_pred = y_new @ weights

# print the results
```

```

print("\nWeights:", weights)
print("Predicted value for the next day:", y_pred)

Mean of Consumption: 1338.6758355897787
Lagged data: [[1069.184   1380.521   1442.533   ... 1403.427   1300.287   1207.985   ]
 [1380.521   1442.533   1457.217   ... 1300.287   1207.985   1529.323   ]
 [1442.533   1457.217   1477.131   ... 1207.985   1529.323   1576.911   ]
 ...
 [1520.37206 1423.23782 1272.17085 ... 1130.11683 1263.94091 1299.86398]
 [1423.23782 1272.17085 1141.7573  ... 1263.94091 1299.86398 1295.08753]
 [1272.17085 1141.7573 1111.28338 ... 1299.86398 1295.08753 1215.44897]]
Target variable: [1529.323   1576.911   1577.176   ... 1295.08753 1215.44897 1107.11488]

Weights: [-0.48031652  0.79451371  0.0741064   0.00317865 -0.04130348  0.08092347
 -0.08570714  0.65400841]
Predicted value for the next day: 1093.2953728795715

```

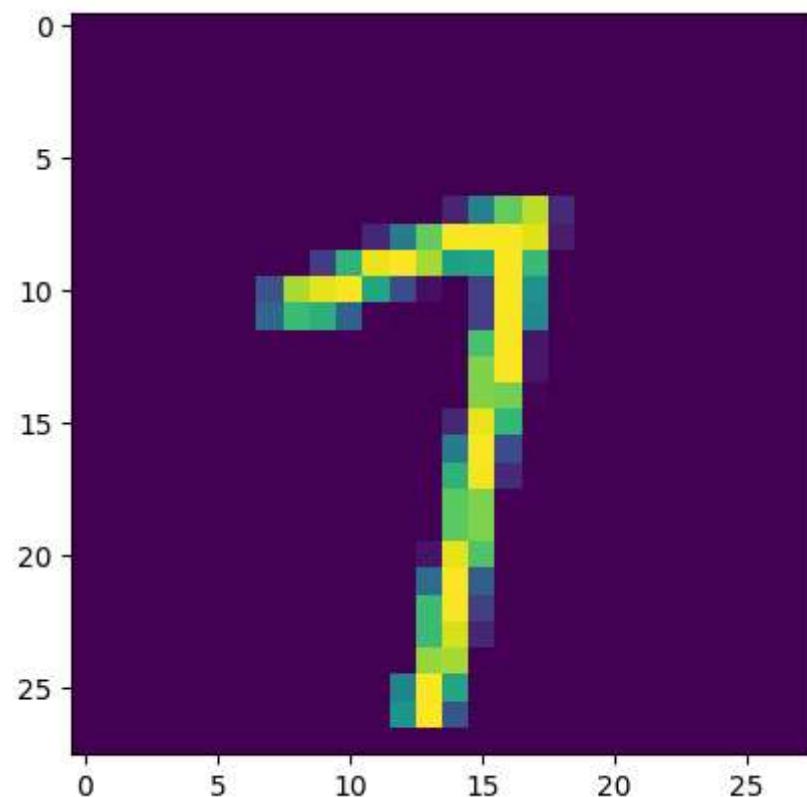
Question 7

This question involves blurring and then deblurring an image. The image is loaded and then blurred using a Gaussian kernel. After this, the blurred image is deblurred using the least squares method. We also use some parameters to keep the difference between consecutive rows and columns of the image as small as possible so that the image is smooth.

Now first of all, we pick the 42nd image from the dataset. It can be seen that it represents the digit 7.

```
In [ ]: # get any image from MNIST dataset, here we are getting the 42nd image
image = np.array(mnist_data_train.data[42])
print("Initial image:")
plt.imshow(image)
flat_image = image.flatten()
```

Initial image:



Kernel and Toeplitz Matrix for Blurring

Now, using the Gaussian kernel, we blur the image. This is achieved by getting the Toeplitz matrix of the kernel and then multiplying it with the image matrix. The Toeplitz matrix of the kernel is generated using `np.roll` function at each row wrt to the flattened kernel.

While this Toeplitz matrix is not very accurate at the edges, we can still use it to get very good results as most of our data is concentrated at the center (true for any image).

After this, the blurred image is printed, and we can see the blurred 7.

```
In [ ]: # define the kernel vector as a 784x1 vector instead of a 3x3 matrix
# this is so that we can generate the toeplitz matrix of the kernel
kernel = np.zeros(784)
```

```
kernel[0], kernel[1], kernel[2] = 1/16, 1/8, 1/16
kernel[28], kernel[29], kernel[30] = 1/8, 1/4, 1/8
kernel[56], kernel[57], kernel[58] = 1/16, 1/8, 1/16

# get the toeplitz matrix of the kernel
# np.roll is used to shift the kernel by i places to the right
toeplitz_kernel = np.zeros((784, 784))
for i in range(784):
    toeplitz_kernel[i] = np.roll(kernel, i)
print("Shape of toeplitz kernel:", toeplitz_kernel.shape)
print("Toeplitz kernel:\n", toeplitz_kernel)

# get the blurred image by multiplying the toeplitz matrix with the flattened image
# equivalent to convolving the image with the kernel matrix
blurred_image = toeplitz_kernel @ flat_image

# reshape the blurred image to 28x28 to get the image
blurred_image = np.reshape(blurred_image, (28, 28))
print("\nBlurred image:")
plt.imshow(blurred_image)
```

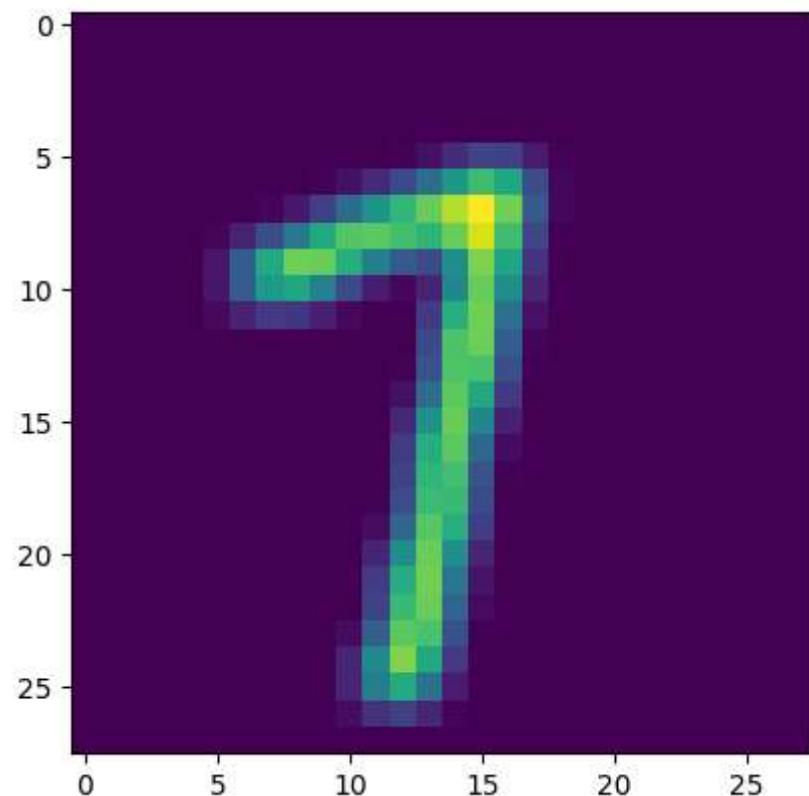
Shape of toeplitz kernel: (784, 784)

Toeplitz kernel:

```
[[0.0625 0.125 0.0625 ... 0.      0.      0.      ]
 [0.      0.0625 0.125  ... 0.      0.      0.      ]
 [0.      0.      0.0625  ... 0.      0.      0.      ]
 ...
 [0.      0.      0.      ... 0.0625 0.125  0.0625]
 [0.0625 0.      0.      ... 0.      0.0625 0.125 ]
 [0.125  0.0625 0.      ... 0.      0.      0.0625]]
```

Blurred image:

Out[]: <matplotlib.image.AxesImage at 0x20380092ad0>



Deblurring the Image Using Least Squares

Now we have to deblur the image using the least squares method. First of all, we generate the constraint matrices D_v and D_h using identity matrices and the smaller D matrix. For generating the matrices, the `np.kron` function is used which basically places the matrix in the diagonal of the larger matrix. Then using some manipulations like adding columns of zeroes and adding the matrices, we obtain the required matrices.

Now, we stack the toeplitz matrix of the kernel with the constraint matrices (multiplied with λ) to get the matrix A . Correspondingly, we stack the blurred image with zeroes to get the vector b .

The λ parameter is set to 0.0007 initially and the deblurred image is obtained. We can see that this λ value is good enough to get a deblurred image.

```
In [ ]: # deblurring the image using the least squares solution

# generating identity matrices of size 27 and 28 along with their negative counterparts for ease of use
I27 = np.identity(27)
MI27 = -I27
I28 = np.identity(28)
MI28 = -I28

# stack an additional column of zeros to the right of MI27 and an additional row of zeros to the left of I27 and add them to g
# we can see that this is equivalent to the matrix D in the assignment
D = np.hstack((np.zeros((27, 1)), I27)) + np.hstack((MI27, np.zeros((27, 1))))
print("Shape of matrix D:", D.shape)

# matrix Dv using D as the diagonal block element
Dv = np.kron(np.eye(28), D)
print("Shape of matrix Dv:", Dv.shape)

# get the stacked matrices I28*27 and MI28*27
I28_27 = np.kron(np.eye(27), I28)
# print("Shape of matrix I28_27:", I28_27.shape)
MI28_27 = np.kron(np.eye(27), MI28)

# stack an additional column of zeros to the right of MI28_27 and an additional row of zeros to the left of I28_27 and add the
Dh = np.hstack((np.zeros((756, 28)), I28_27)) + np.hstack((MI28_27, np.zeros((756, 28))))
print("Shape of matrix Dh:", Dh.shape)

# stack matrices toeplitz_kernel, Dv and Dh to get the matrix A
lambda_ = 0.0007
Dv_, Dh_ = lambda_* Dv, lambda_* Dh
A = np.vstack((toeplitz_kernel, Dv_, Dh_))
print("Shape of matrix A:", A.shape)

# similarly stack the blurred image, zeros and zeros to get the vector b
b = np.hstack((blurred_image.flatten(), np.zeros(756), np.zeros(756)))
print("Shape of vector b:", b.shape)

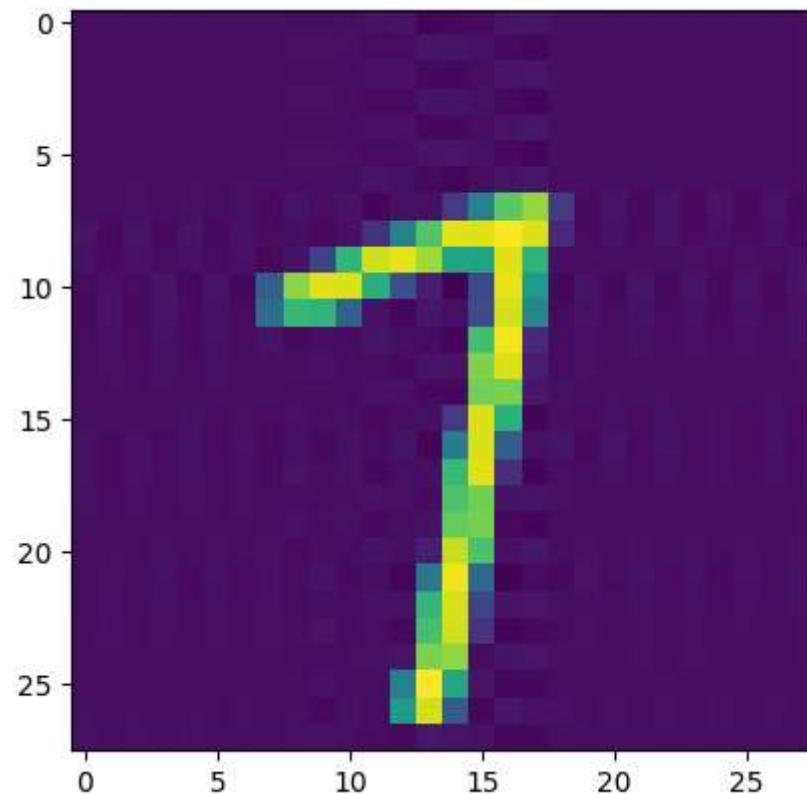
# calculate the least squares solution
x = least_squares(A, b)
print("Shape of vector x:", x.shape)
```

```
# deblur the image using the Least squares solution  
deblurred_image = np.reshape(x, (28, 28))  
print("\nDeblurred image with lambda = 0.0007:")  
plt.imshow(deblurred_image)
```

Shape of matrix D: (27, 28)
Shape of matrix Dv: (756, 784)
Shape of matrix Dh: (756, 784)
Shape of matrix A: (2296, 784)
Shape of vector b: (2296,)
Shape of vector x: (784,)

Deblurred image with lambda = 0.0007:

Out[]: <matplotlib.image.AxesImage at 0x20380172ad0>



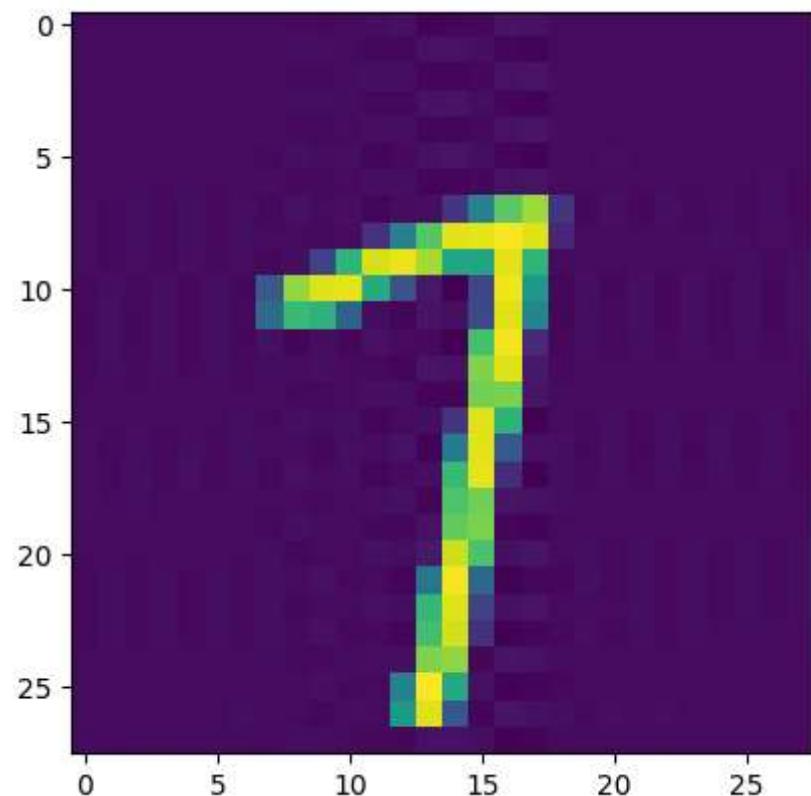
Variation of Lambda_

Now, we vary the hyperparameters a bit to have a look at the effect of lambda_ on the deblurred image. It can be seen that for large values of lambda_, the image is not deblurred properly but it is very smooth. For small values of lambda_, the image is deblurred properly but it is not very smooth. Hence intermediate values should be chosen appropriately.

```
In [ ]: # Lambda = 0.0001
lambda_ = 0.0001
Dv_, Dh_ = lambda_ * Dv, lambda_ * Dh
A = np.vstack((toeplitz_kernel, Dv_, Dh_))
x = least_squares(A, b)
deblurred_image = np.reshape(x, (28, 28))
print("\nDeblurred image with lambda = 0.0001:")
plt.imshow(deblurred_image)

Deblurred image with lambda = 0.0001:
```

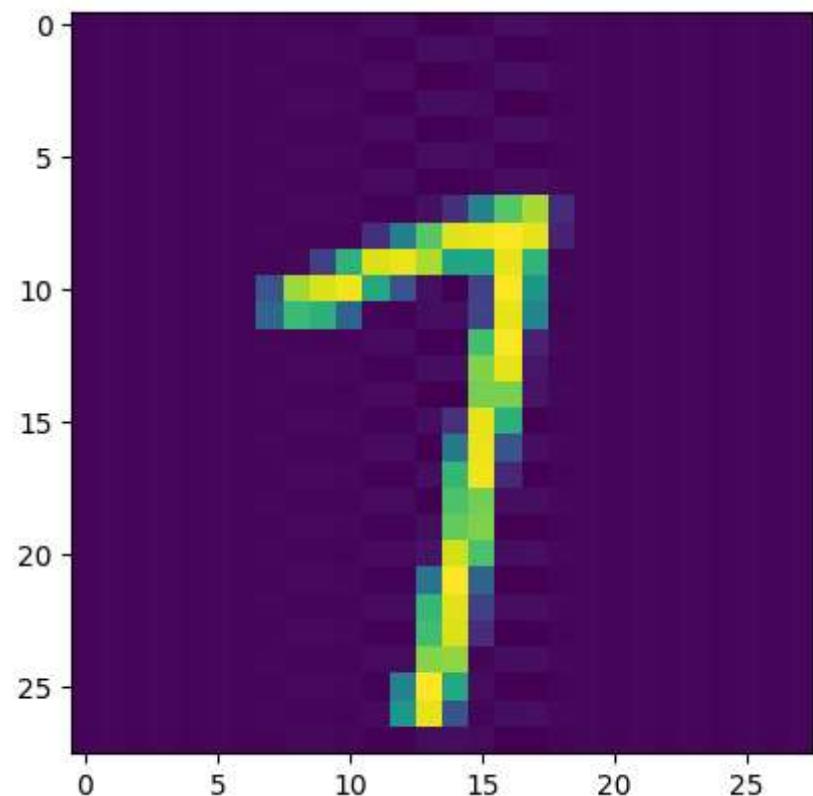
```
Out[ ]: <matplotlib.image.AxesImage at 0x203801baad0>
```



```
In [ ]: # Lambda = 0
lambda_ = 0
Dv_, Dh_ = lambda_* Dv, lambda_* Dh
A = np.vstack((toeplitz_kernel, Dv_, Dh_))
x = least_squares(A, b)
deblurred_image = np.reshape(x, (28, 28))
print("\nDeblurred image with lambda = 0:")
plt.imshow(deblurred_image)
```

Deblurred image with lambda = 0:

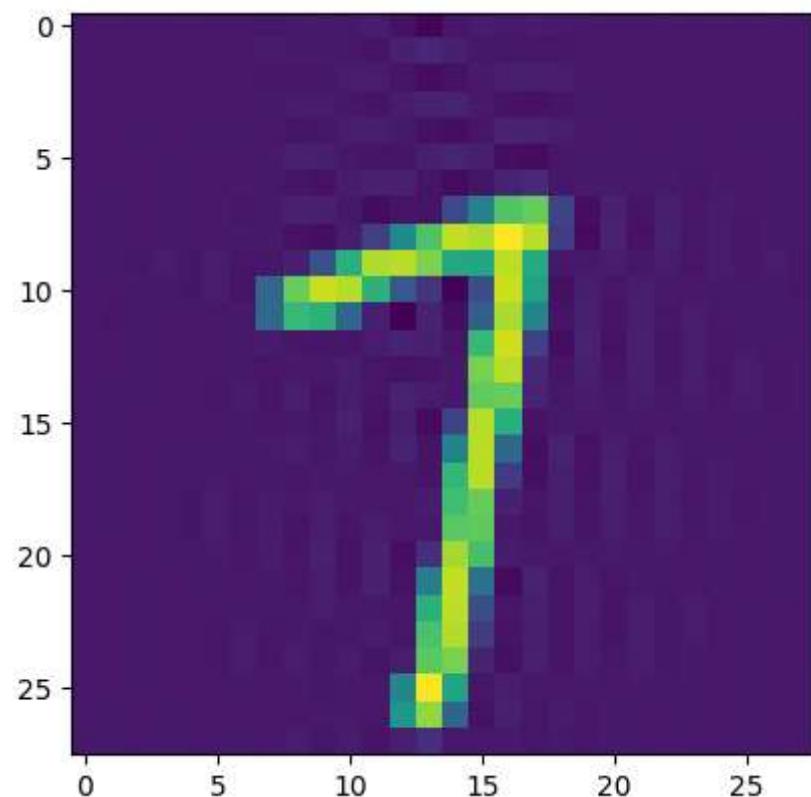
Out[]: <matplotlib.image.AxesImage at 0x20380172e10>



```
In [ ]: # Lambda = 0.01
lambda_ = 0.01
Dv_, Dh_ = lambda_* Dv, lambda_* Dh
A = np.vstack((toeplitz_kernel, Dv_, Dh_))
x = least_squares(A, b)
deblurred_image = np.reshape(x, (28, 28))
print("\nDeblurred image with lambda = 0.01:")
plt.imshow(deblurred_image)
```

Deblurred image with lambda = 0.01:

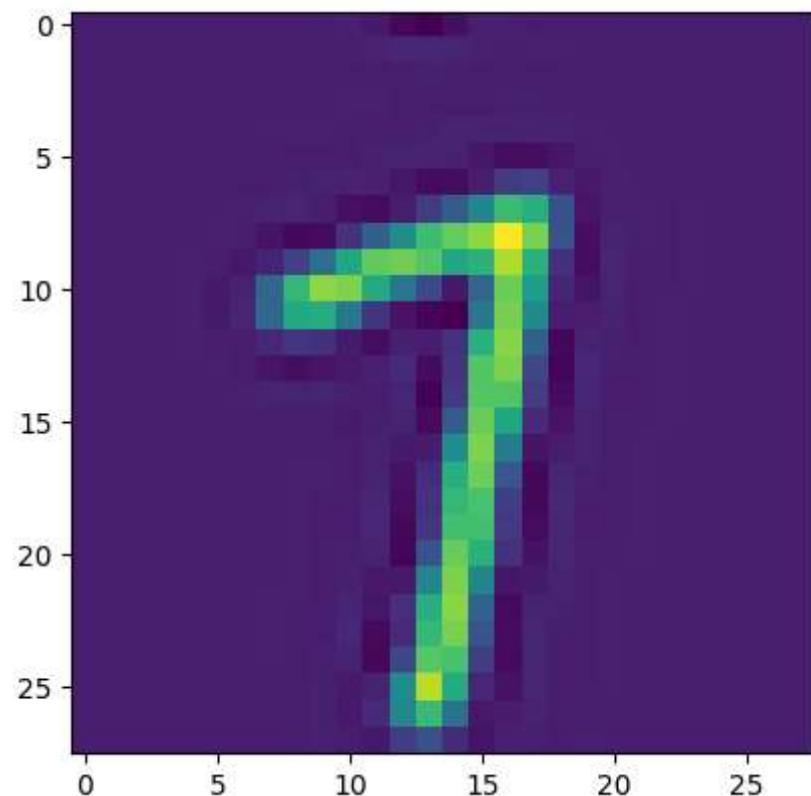
Out[]: <matplotlib.image.AxesImage at 0x2038263aad0>



```
In [ ]: # Lambda = 0.1
lambda_ = 0.1
Dv_, Dh_ = lambda_* Dv, lambda_* Dh
A = np.vstack((toeplitz_kernel, Dv_, Dh_))
x = least_squares(A, b)
deblurred_image = np.reshape(x, (28, 28))
print("\nDeblurred image with lambda = 0.1:")
plt.imshow(deblurred_image)
```

Deblurred image with lambda = 0.1:

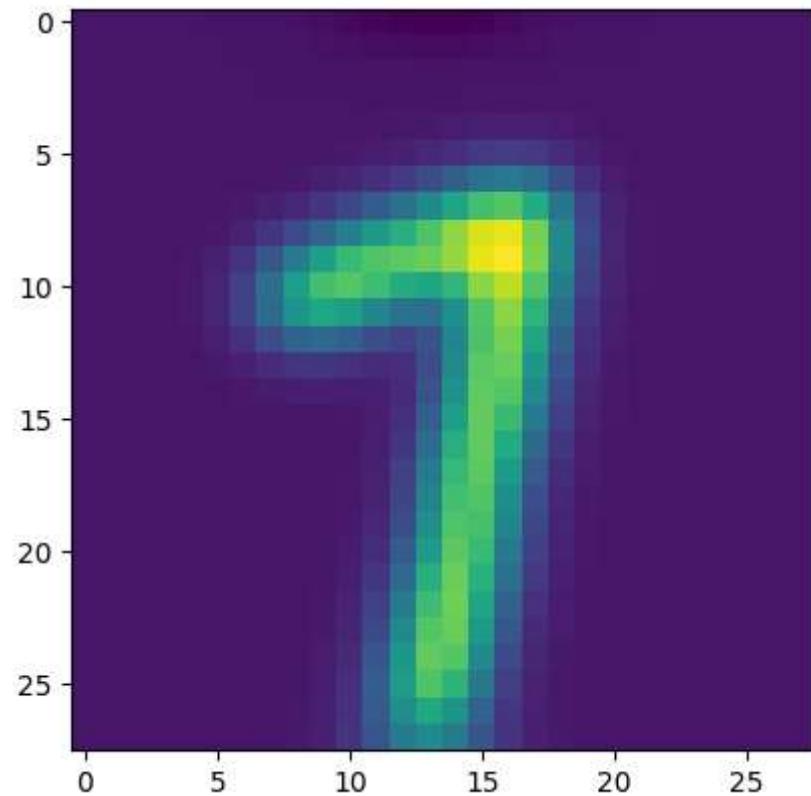
Out[]: <matplotlib.image.AxesImage at 0x203826778d0>



```
In [ ]: # Lambda = 1
lambda_ = 1
Dv_, Dh_ = lambda_* Dv, lambda_* Dh
A = np.vstack((toeplitz_kernel, Dv_, Dh_))
x = least_squares(A, b)
deblurred_image = np.reshape(x, (28, 28))
print("\nDeblurred image with lambda = 1:")
plt.imshow(deblurred_image)
```

Deblurred image with lambda = 1:

```
Out[ ]: <matplotlib.image.AxesImage at 0x2038246f050>
```



Question 8

Note: This is the same question as 4, just that we use the analytical solution there and here we use the gradient descent algorithm.

Here first of all, we generate two random vectors p and q of length 100 sampled from uniform random distribution between [-1,1].

Then the target variable b is defined as $pq + p^2 + q^2$.

Now we generate 6 basis functions as provided in order to implement the gradient descent algorithm to find the optimal weights vector x . Then we generate the matrix A of basis functions.

Gradient descent algorithm is implemented using the cost function and the update weights function. x (here theta) is initialized as a random vector of length 6.

The learning rate is set as 0.01 so that the algorithm converges to the optimal weights vector and the number of iterations is set as 50000. The cost graph is plotted to see how the cost function decreases with the number of iterations.

As we can see very easily that the target function is a linear combination of the basis functions, the mean squared error is very close to zero and the weights vector x (or theta) is very close to [0,0,0,1,1,1] as expected.

```
In [ ]: # generate vectors p and q
p = np.random.uniform(-1, 1, 100)
q = np.random.uniform(-1, 1, 100)

# generate the target variable b
b = p * q + p**2 + q**2

# defining the 6 basis functions
f0 = np.ones(100)
f1 = p
f2 = q
f3 = p**2
f4 = q**2
f5 = p * q

# creating matrix A from the basis functions
A = np.column_stack((f0, f1, f2, f3, f4, f5))

# gradient descent functions
# cost function
def cost(x,y,theta):
    return np.sum((x @ theta - y)**2)/(2*len(y))

# gradient descent function which updates theta
def gradient_descent(x,y,theta,learning_rate):
    return theta - learning_rate * x.T @ (x @ theta - y) / len(y)

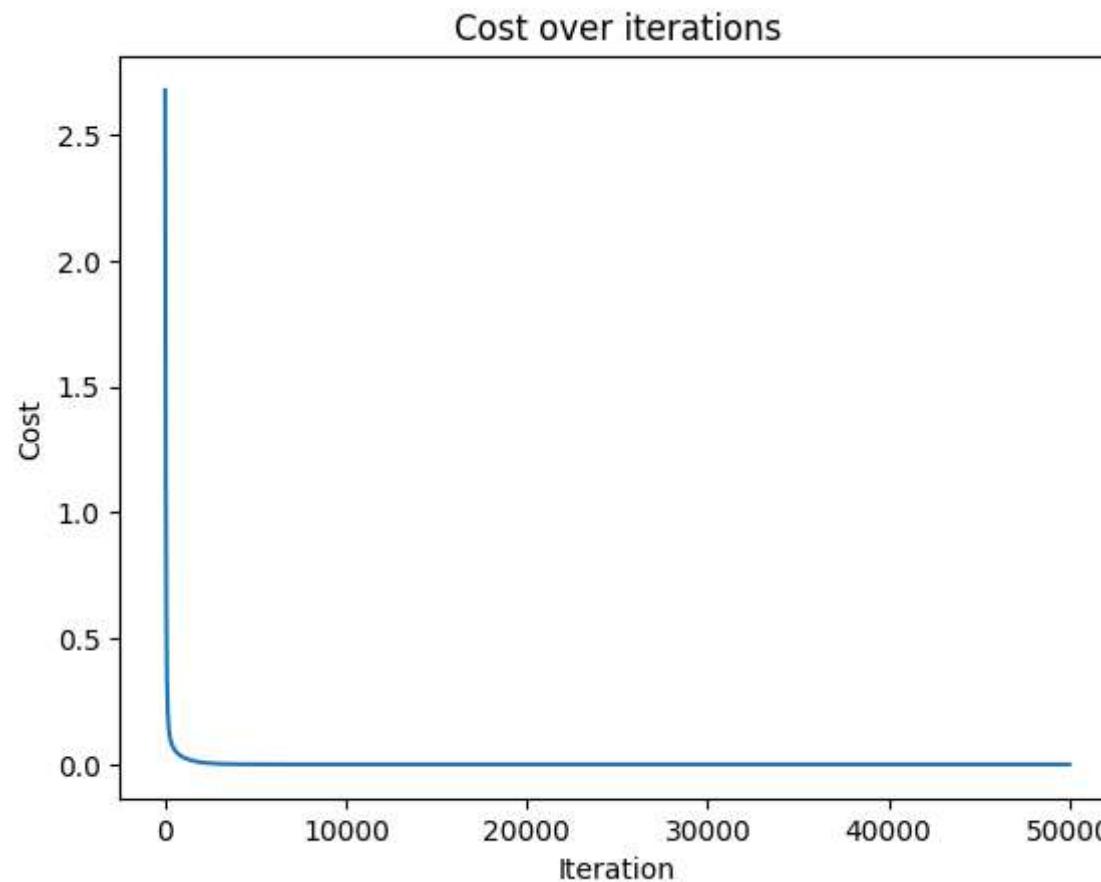
# doing the gradient descent and plotting the cost function over iterations
# random initial theta
theta = np.random.randn(6)
```

```
# print("Initial theta:", theta)

learning_rate = 0.01
num_iterations = 50000
cost_history = np.zeros(num_iterations)

for i in range(num_iterations):
    theta = gradient_descent(A,b,theta,learning_rate)
    cost_history[i] = cost(A,b,theta)
plt.plot(cost_history)
plt.title("Cost over iterations")
plt.xlabel("Iteration")
plt.ylabel("Cost")
plt.show()

# calculating the mse and printing the results
mse = np.mean((A @ theta - b)**2)
print("Coefficient vector x:", theta)
print("Mean Squared Error (MSE):", mse)
```



Coefficient vector x : [2.40646575e-13 -2.02928184e-14 2.29866015e-14 1.00000000e+00
1.00000000e+00 1.00000000e+00]

Mean Squared Error (MSE): 1.9757127476428388e-26

Question 9

Here, based on the given data, we have to fit 3 piece wise polynomial functions on the interval $[0,1]$ and $[1,2]$ and $[2,3]$ respectively for f_1 , f_2 and f_3 .

It is given that f_1 is degree 2 polynomial, f_2 is degree 3 polynomial and f_3 is degree 2 polynomial. Further, to impose continuity and differentiability at the boundaries, some additional constraints are added. The matrix A and the vector b are generated accordingly. The matrix

C and the vector d are generated to impose the constraints. Finally they are stacked to get the matrix A and the vector b. Note that a heavy penalty is imposed on the constraints so that they are satisfied. ($\lambda = 100$)

$$f_1(1) = f_2(1) \text{ and } f_2(2) = f_3(2) \text{ and } f_2'(2) = f_3'(2)$$

We get the optimal parameters and divide them into 3 parts to get the parameters for the 3 polynomials.

The polynomials are printed as follows:

$$f_1(x) = 4.00 + -3.00x + 1.00x^2$$

$$f_2(x) = 1.00 + 0.00x + -2.00x^2 + 3.00x^3$$

$$f_3(x) = -35.00 + 24.00x + 1.00x^2$$

We see that the required polynomials also satisfy the given constraints. (Note that $f_1'(1) \neq f_2'(1)$, but it is not our constraint.)

Finally, the data points and the fitted polynomials are plotted to see how well the polynomials fit the data, it can be seen that our polynomials fit the data very well.

```
In [ ]: # Loading the data
data = pd.read_csv("q9.csv")
data = data.values

# dividing the data into three parts
part_1 = data[(data[:, 0] >= 0) & (data[:, 0] <= 1)]
part_2 = data[(data[:, 0] > 1) & (data[:, 0] <= 2)]
part_3 = data[(data[:, 0] > 2) & (data[:, 0] <= 3)]

# define the vandermonde matrices for each part with corresponding degrees
vander_1 = np.vander(part_1[:, 0], 3, increasing=True)
vander_2 = np.vander(part_2[:, 0], 4, increasing=True)
vander_3 = np.vander(part_3[:, 0], 3, increasing=True)

# matrix A is the main matrix for least squares with data
# A1, A2 and A3 are the three segments wrt the vander matrices which are joined appropriately to get the matrix A
A1 = np.concatenate((vander_1, np.zeros((len(part_1), 4)), np.zeros((len(part_1), 3))), axis=1)
A2 = np.concatenate((np.zeros((len(part_2), 3)), vander_2, np.zeros((len(part_2), 3))), axis=1)
```

```

A3 = np.concatenate((np.zeros((len(part_3), 3)), np.zeros((len(part_3), 4)), vander_3), axis=1)
A = np.concatenate((A1, A2, A3), axis=0)
print("Shape of matrix A:", A.shape)

# vector b is similarly the data points stacked together
b = np.concatenate((part_1[:, 1], part_2[:, 1], part_3[:, 1]), axis=0)
print("Shape of vector b:", b.shape)

# matrix C for constraints f1(1) = f2(1) and f2(2) = f3(2) and f2'(2) = f3'(2)
# C1, C2 and C3 are the constraint vectors with appropriate coefficients
C1 = np.array([1, 1, 1, -1, -1, -1, 0, 0, 0])    # f1(1) = f2(1)
C2 = np.array([0, 0, 0, 1, 2, 4, 8, -1, -2, -4])    # f2(2) = f3(2)
C3 = np.array([0, 0, 0, 0, 1, 4, 12, 0, -1, -4])    # f2'(2) = f3'(2)
C = np.vstack((C1, C2, C3))
print("Shape of matrix C:", C.shape)

# vector d is just zeros to make the constraints valid
d = np.zeros(3)
print("Shape of vector d:", d.shape)

# define Lambda_ to be the regularization parameter
# here we have taken Lambda_ = 100 as it should be large enough to make the constraints the dominant factor
lambda_ = 100

# stack A and C to get the matrix A and stack b and d to get the vector b and call the least squares to get the params
A = np.vstack((A, lambda_ * C))
b = np.hstack((b, d))
print("Shape of final matrix A and vector b:", A.shape, b.shape)
params = least_squares(A, b)

# get the params for each part
params_1, params_2, params_3 = params[:3], params[3:7], params[7:]

# print the polynomials
print("\nPolynomial for part 1 = {:.2f} + {:.2f}x + {:.2f}x^2".format(params_1[0], params_1[1], params_1[2]))
print("Polynomial for part 2 = {:.2f} + {:.2f}x + {:.2f}x^2 + {:.2f}x^3".format(params_2[0], params_2[1], params_2[2], params_2[3]))
print("Polynomial for part 3 = {:.2f} + {:.2f}x + {:.2f}x^2".format(params_3[0], params_3[1], params_3[2]))

print("\nConstraints:")
# reverse the order of the coefficients to get the polynomial in the correct order for the polyval function
rev_params_1 = params_1[::-1]

```

```
rev_params_2 = params_2[::-1]
rev_params_3 = params_3[::-1]

# check continuity
print("f1(1) = {:.2f}".format(np.polyval(rev_params_1, 1)))
print("f2(1) = {:.2f}".format(np.polyval(rev_params_2, 1)))
print("f2(2) = {:.2f}".format(np.polyval(rev_params_2, 2)))
print("f3(2) = {:.2f}".format(np.polyval(rev_params_3, 2)))

# check differentiability
print("f2'(2) = {:.2f}".format(np.polyval(np.polyder(rev_params_2), 2)))
print("f3'(2) = {:.2f}".format(np.polyval(np.polyder(rev_params_3), 2)))

# plotting the data and the polynomials
print("\nPlotting the data and the polynomials:")
plt.scatter(data[:, 0], data[:, 1], label="Data", color="darkblue")

x = np.linspace(0, 1, 100)
plt.plot(x, np.polyval(rev_params_1, x), label="Polynomial for part 1")
x = np.linspace(1, 2, 100)
plt.plot(x, np.polyval(rev_params_2, x), label="Polynomial for part 2")
x = np.linspace(2, 3, 100)
plt.plot(x, np.polyval(rev_params_3, x), label="Polynomial for part 3")

plt.title("Data and Polynomials")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.show()
```

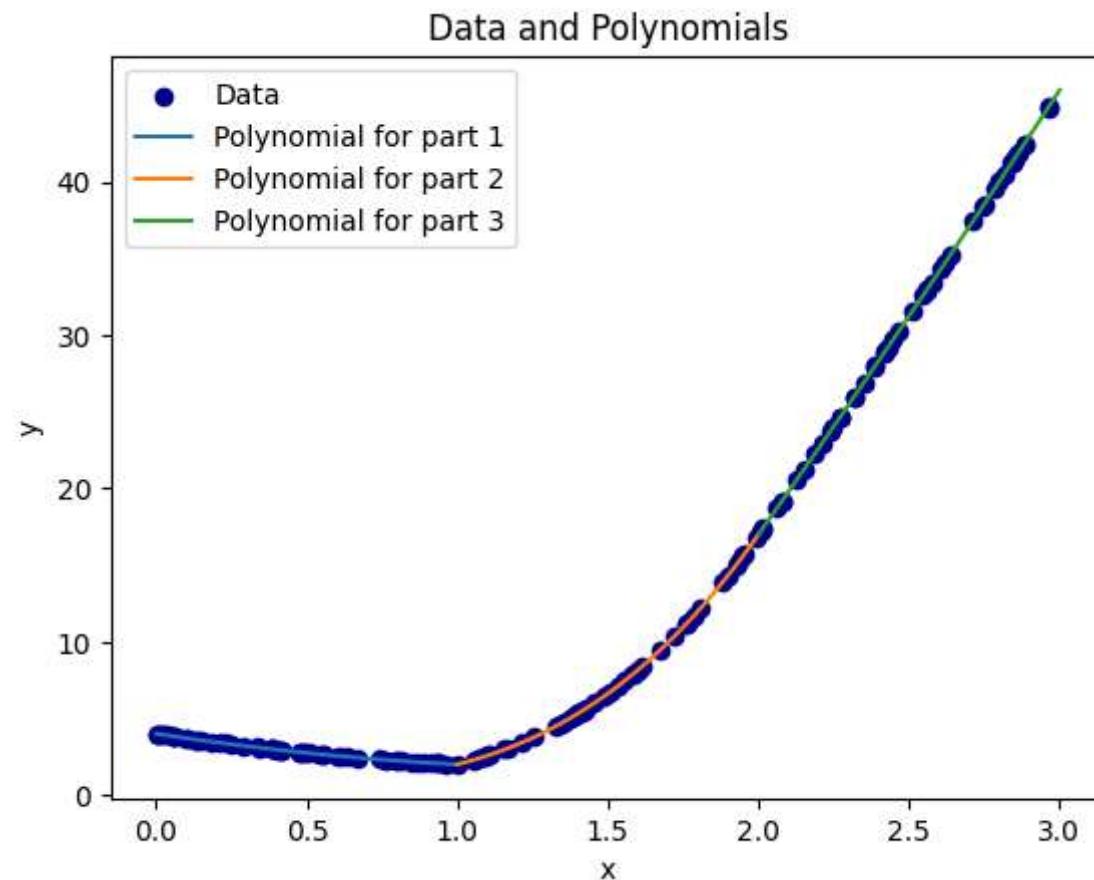
```
Shape of matrix A: (150, 10)
Shape of vector b: (150,)
Shape of matrix C: (3, 10)
Shape of vector d: (3,)
Shape of final matrix A and vector b: (153, 10) (153,)
```

```
Polynomial for part 1 = 4.00 + -3.00x + 1.00x^2
Polynomial for part 2 = 1.00 + -0.00x + -2.00x^2 + 3.00x^3
Polynomial for part 3 = -35.00 + 24.00x + 1.00x^2
```

Constraints:

```
f1(1) = 2.00
f2(1) = 2.00
f2(2) = 17.00
f3(2) = 17.00
f2'(2) = 28.00
f3'(2) = 28.00
```

Plotting the data and the polynomials:



Conclusion

This assignment was based on least squares method and its applications. We saw how to use least squares method to solve regression problems and classification problems. We also saw how to use least squares method to solve polynomial fitting problems.

Thank you.