Assignment 2-1

Problem 1: You're trying to figure out the price of apples, bananas, and cherries at the store. You go three days in a row, and bring this information.

- **Day 1:** You bought an apple, a banana, and a cherry, and paid $10.
- **Day 2:** You bought an apple, two bananas, and a cherry, and paid $15.
- **Day 3:** You bought an apple, a banana, and two cherries, and paid $12.

How much does each fruit cost?

Please solve this problem using the Matrix Technique.

Question 1: Solve Using the Matrix Technique

1. **Step 1:** Identify the number of equations and variables. Clearly define and describe the equation presented in the problem.
2. **Step 2:** Convert the system of equations into a matrix form.
3. **Step 3:** Utilize NumPy to calculate the inverse of the matrix. Check if the matrix is singular and assess its invertibility.
4. **Step 4:** Solve for the variables and substitute them back into the original equation to verify accuracy.

Step1:

- a = apple
- b = banana
- c = cherry

$$a + b + c = 10$$
$$a + 2b + c = 15$$
$$a + b + 2c = 12$$

Step2:

$$a + b + c = 10$$
$$a + 2b + c = 15$$
$$a + b + 2c = 12$$

$$\text{Matrix} \rightarrow \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 10 \\ 15 \\ 12 \end{bmatrix}$$

$$\text{let} \rightarrow A \quad X = B$$

Step3:

Calculation

$$AX = B$$
$$X = A^{-1}B$$

Step three: code in Python (using Numpy library)
code screenshot in assignment submission

solution result: $x = 3,\ y = 5,\ z = 2$

```python
# step 2: Convert into matrix form
A = np.array([[1,1, 1],[1,2,1],[1,1,2]])
B = np.array([[10],[15],[12]])

#step3
# calculate the inverse of A
A_inv = np.linalg.inv(A)

# print inverse of A
print("Inverse of A: ")
print(A_inv)
print("-"*30)

# check if x is singular and assess its invertibility
if np.linalg.det(A) == 0:
    print("Matrix A is singular and not invertible")
else:
    print("Matrix A is invertible")
print("-"*30)
```

```
Inverse of A:
[[ 3. -1. -1.]
 [-1.  1.  0.]
 [-1.  0.  1.]]
------------------------------
Matrix A is invertible
------------------------------
```

Step4:

```
# Step 4: Solve for X (the prices)
X = np.dot(A_inv, B)        # Matrix multiplication
print("\nStep 3: Solution Matrix X:\n", X)

# print the prices
apple_price = X[0][0]
banana_price = X[1][0]
cherry_price = X[2][0]

print("\nStep 4: Prices of Fruits")
print("Apple cost: $", apple_price)
print("Banana cost: $", banana_price)
print("Cherry cost: $", cherry_price)
```

```
Step 4: Prices of Fruits
Apple cost: $ 3.0
Banana cost: $ 5.0
Cherry cost: $ 2.0
```

$x = 3$

$y = 5$

$z = 2$

Verification

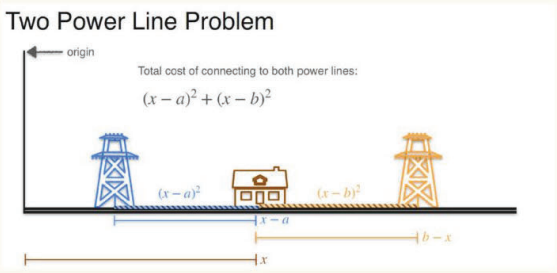
① $3 + 5 + 2 = 10$

② $3 + 10 + 2 = 15$

③ $3 + 5 + 4 = 12$

ASSIGNMENTS 2-2

Two Power Line Problem



Question 2: Differential Equation Analysis

- **Problem:** Solve the differential equation y=(x-a)**2+(x-b)**2, where a=1 and b=5. Find the value of x that minimizes y (lead to lowest y value).
- **Approach:** Experiment and calculate rigorously to determine the optimal value of x.

1.	$\frac{d}{dx}(c)=0$	$c'=0$
2.	$\frac{d}{dx}(x^n)=nx^{n-1}$	$(x^n)'=nx^{n-1}$
3.	$\frac{d}{dx}(f+g)=\frac{df}{dx}+\frac{dg}{dx}$	$(f+g)'=f'+g'$
4.	$\frac{d}{dx}(cf+kg)=c\frac{df}{dx}+k\frac{dg}{dx}$	$(cf+kg)'=cf'+kg'$
5.	$\frac{d}{dx}(fg)=\frac{df}{dx}g+\frac{dg}{dx}f$	$(fg)'=f'g+g'f$
6.	$\frac{d}{dx}(\frac{f}{g})=\frac{g(x)\frac{d}{dx}f(x)-f(x)\frac{d}{dx}g(x)}{g(x)^2}$	$(\frac{f}{g})'=\frac{gf'-fg'}{g^2}$
7.	$\frac{d}{dx}(u^n)=nu^{n-1}\frac{du}{dx}$	
8.	$\frac{d}{dx}(f\circ g)=\frac{d}{dg(x)}f(g(x))\frac{d}{dx}g(x)$	$(f\circ g)'(x)=f'(g(x))g'(x)$
9.	$\frac{d}{dx}\ln\lvert x\rvert=\frac{1}{x}$	10. $\frac{d}{dx}e^x=e^x$
11.	$\frac{d}{dx}a^x=a^x\ln a$	12. $\frac{d}{dx}\log_a\lvert x\rvert=\frac{1}{x\ln a}$
13.	$\frac{d}{dx}\sin x=\cos x$	14. $\frac{d}{dx}\cos x=-\sin x$
15.	$\frac{d}{dx}\tan x=\sec^2 x$	16. $\frac{d}{dx}\cot x=-\csc^2 x$
17.	$\frac{d}{dx}\sec x=\sec x\tan x$	18. $\frac{d}{dx}\csc x=-\csc x\cot x$
19.	$\frac{d}{dx}\arcsin x=\frac{1}{\sqrt{1-x^2}}$	20. $\frac{d}{dx}\arccos x=-\frac{1}{\sqrt{1-x^2}}$
21.	$\frac{d}{dx}\arctan x=\frac{1}{1+x^2}$	22. $\frac{d}{dx}\operatorname{arccot} x=-\frac{1}{1+x^2}$
23.	$\frac{d}{dx}\operatorname{arcsec} x=\frac{1}{\lvert x\rvert\sqrt{x^2-1}}$	24. $\frac{d}{dx}\operatorname{arccsc} x=-\frac{1}{\lvert x\rvert\sqrt{x^2-1}}$

DATE :
MON TUE WED THU FRI SAT SUN

SUBJECT :

(2)

$y = (x-a)^2 + (x-b)^2$

Now,

$\frac{dy}{dx} = 2(x-a) + 2(x-b)$

$= 2x - 2a + 2x - 2b$

$= 4x - 2a - 2b$

Now,

$4x - 2a - 2b = 0$

$\Rightarrow, x = \frac{2a+2b}{4}$

$\therefore x = \frac{a+b}{2}$

Now, when ~~x =~~ $a = 1$ and $b = 5$,

$x = \frac{1+5}{2} = 3$ //

House should be at 3 units //

y * minimum $= (3-1)^2 + (3-5)^2$

$= 4 + 4$

$= 8$ units cast //

Assignment 3: Maximum Likelihood Estimation

Instructions: This assignment engages you in the practical application of maximum likelihood estimation for linear regression modeling. Follow the steps carefully, and submit your work as a single PDF file that includes all required code and graphs. **Attach** supplementary code files (.py and .ipynb) as used in this assignment.

Objective: Learn how to fit a linear regression model using maximum likelihood estimation and understand the effects of model adjustments, such as adding a bias term.

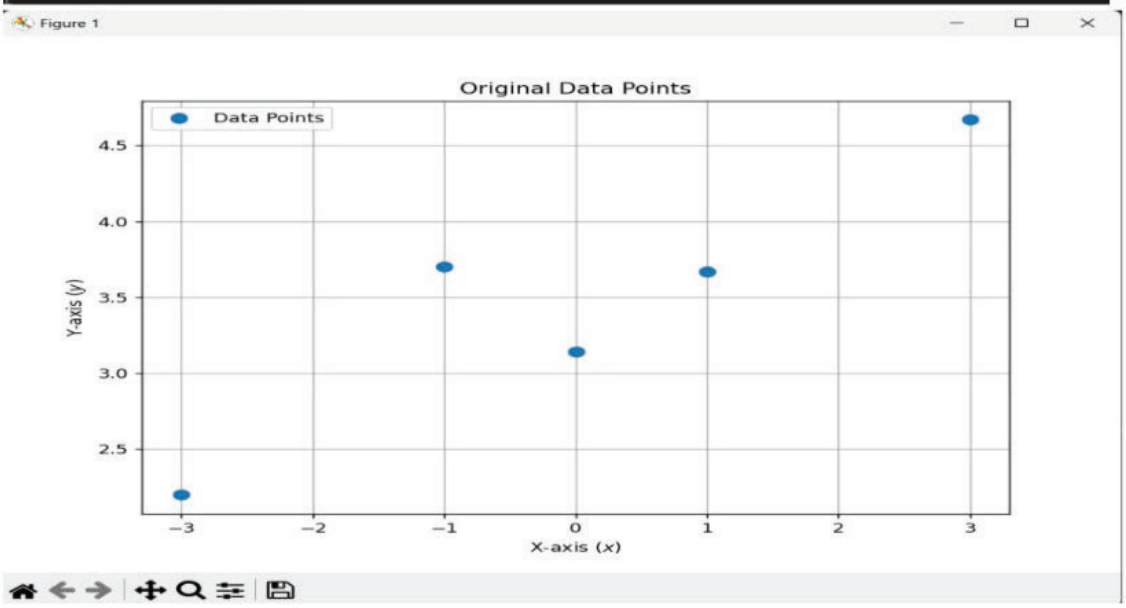
Step 1: Graphing the Relationship

```
import numpy as np
import scipy.linalg
import matplotlib.pyplot as plt

X = np.array([-3, -1, 0.0, 1, 3]).reshape(-1,1) # 5x1 vector, N=5, D=1
y = np.array([2.2, 3.7, 3.14, 3.67, 4.67]).reshape(-1,1) # 5x1 vector
```

- Using the provided code, plot the relationship between variables `x` and `y`.
- Attach the Python code used to generate the graph in your PDF.
- Display the graph you obtain.

Step 1: Graphing the relationship

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

# Step 1: Define Data
X = np.array([-3, -1, 0.0, 1, 3]).reshape(-1, 1)  # Training input (5x1)
y = np.array([2.2, 3.7, 3.14, 3.67, 4.67]).reshape(-1, 1)  # Training output (5x1)

# Step 1: Plot the data (without linear trend)
def step1plotData(X, y):
    plt.figure(figsize=(8, 6))
    plt.plot(X, y, 'o', markersize=8, label='Data Points')
    plt.xlabel("X-axis ($x$)")
    plt.ylabel("Y-axis ($y$)")
    plt.title("Original Data Points")
    plt.legend()
    plt.grid(True)
    plt.show()

# Call the function to plot
step1plotData(X, y)
```



Step 2: Applying the Derived Equation

$$\theta_{\text{ML}} = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y}$$

- Implement the final equation we derived in class to compute the parameter Theta (θ).
  - Refer to the Python file attached with this assignment for the implementation guide.
- Display the resulting Theta value and attach the code used to obtain it.
- Attach the Python code used in your PDF.

Step 2: Applying the Derived Equation:

```
22  # Step 2: Calculate Theta using MLE
23  def step2Equation(X, y):
24      inverse = np.linalg.inv(X.T @ X)
25      theta = inverse @ (X.T @ y)
26      return theta
27
28  # Step 2: Make Predictions
29  def step2Estimate(Xtest, theta):
30      return Xtest @ theta
31
32  theta_ml = step2Equation(X, y)# Calculate Theta
33  print("=" * 10)
34  print("theta without Bias" , theta_ml)
35  print("=" * 10)
```

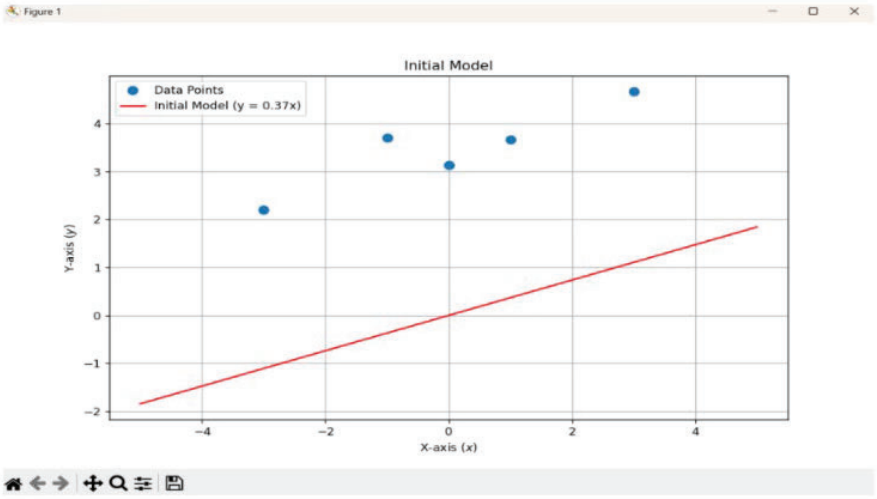
```
PS C:\Users\User\Desktop\AI Concepts>
op/AI Concepts/MLEassignment.py"
==========
theta without Bias [[0.369]]
```

Step 3: Plotting the Initial Model

- Apply the Theta obtained from Step 2 to a linear equation.
- Plot this equation on the same graph as in Step 1.
- Ensure the graph displays the original 5 data points and the linear model representing the obtained Theta.
- Attach the Python code used to generate the graph in your PDF.

Step 3: Plotting initial Model:

```
# Step 3: Initial Linear Model (Without Bias)
Xtest = np.linspace(-5, 5, 100).reshape(-1, 1)  # Test inputs
ml_prediction = step2Estimate(Xtest, theta_ml)# Predictions
plt.figure(figsize=(10, 6))

# Original Data Points
plt.plot(X, y, 'o', label='Data Points', markersize=8)

# Initial Model
plt.plot(Xtest, ml_prediction,
         label=f'Initial Model (y = {theta_ml[0][0]:.2f}x)',
         color='red')
```



Step 4: Improving the Model

- You will notice that the linear model does not fit well with the training points provided. This is because we did not include the bias of the linear equation when calculating Theta initially.
- Enhance your model by including a bias term. Stack a column of ones vertically with your **x** data (as shown in the provided code snippet).

```
# ================================================================================
print("="*10)
N, D = X.shape
X_aug = np.hstack([np.ones((N,1)), X]) # augmented training inputs of size N x (D+1)
print("X_aug = ")
print(X_aug)
print("="*10)
# ================================================================================
```

```
==========
X_aug =
[[ 1. -3.]
 [ 1. -1.]
 [ 1.  0.]
 [ 1.  1.]
 [ 1.  3.]]
==========
```

- Recalculate Theta using this new matrix configuration.
- The resulting Theta should now have two values (shape (2,1)).
- Attach your code and display both values of Theta.
- Explain how these values differ from those obtained in Step 2.

Step 4: Improving the model:

```
# Step 4: Improve Model by Adding Bias
X_aug = np.hstack((np.ones_like(X), X))# Add bias to training data
print("X_aug = ")
print(X_aug)
print("=" * 10)

theta_ml_bias = step2Equation(X_aug, y)# Recalculate Theta with bias
print("=" * 10)
print("theta with Bias" , theta_ml_bias.flatten())

Xtest_bias = np.hstack((np.ones_like(Xtest), Xtest))# Add bias to test inputs

ml_prediction_bias = step2Estimate(Xtest_bias, theta_ml_bias) # Predictions with bias
```

```
==========
X_aug =
[[ 1. -3.]
 [ 1. -1.]
 [ 1.  0.]
 [ 1.  1.]
 [ 1.  3.]]
==========
==========
theta with Bias [3.476 0.369]
==========
```

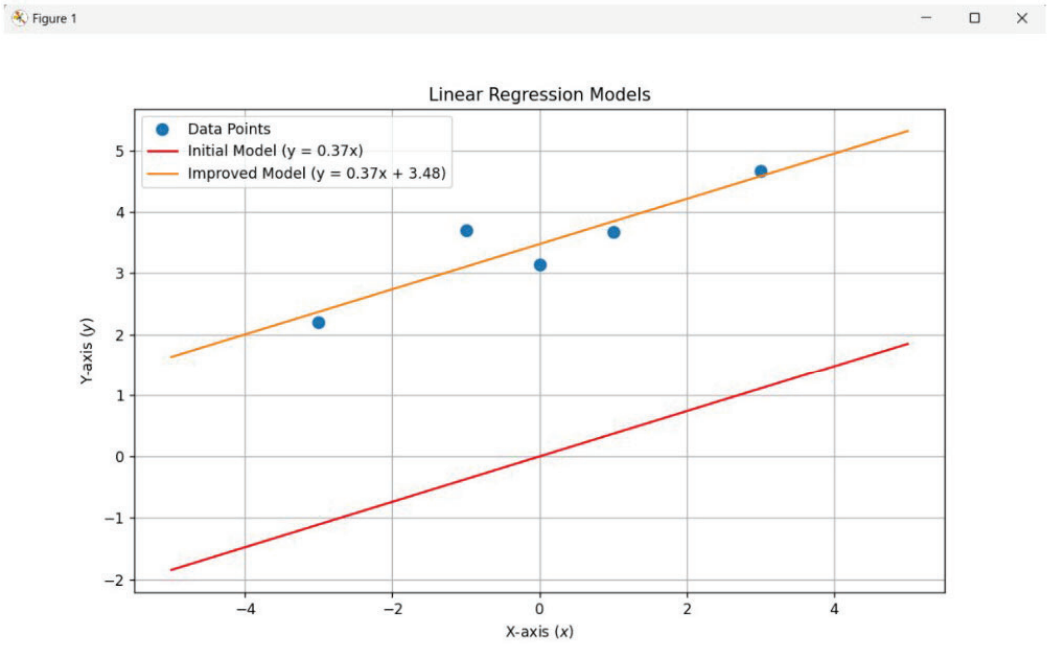
Here the values obtained differ from step 2, where only 0.369 (the slope was calculated. The first value theta [0] is the bias accounting for the vertical shift, no longer being constrained to pass through origin.

Step 5: Plotting the Improved Model

- Use the new Theta values from Step 4 in your linear equation.
- Plot this updated model on the same graph as in Step 1.
- The graph should still display the original 5 data points, along with the new linear model.
- Attach the Python code used to generate the graph in your PDF.

Step 5: Plotting improved Model:

```
# Plot Results: Step 5
plt.figure(figsize=(10, 6))

# Original Data Points
plt.plot(X, y, 'o', label='Data Points', markersize=8)

# Initial Model
plt.plot(Xtest, ml_prediction, label=f'Initial Model (y = {theta_ml[0][0]:.2f}x)', color='red')

# Improved Model with Bias
plt.plot(Xtest, ml_prediction_bias,
         label=f'Improved Model (y = {theta_ml_bias[1][0]:.2f}x + {theta_ml_bias[0][0]:.2f})')

# Add Labels and Styling
plt.xlabel("X-axis ($x$)")
plt.ylabel("Y-axis ($y$)")
plt.title("Linear Regression Models")
plt.legend()
plt.grid(True)
plt.show()
```



Step 6: Analytical Explanation

- Explain why the addition of a column of ones in Step 4 (bias term) and subsequent recalculation improves the fit of the model to the data.

Explanation Step 6:

The bias term acts as the y-intercept of the linear equation. Without the bias term, the regression line is constrained to pass through the origin (0,0)By introducing the bias term, the model can adjust vertically, allowing the regression line to better align with the central trend of the data points.

The bias theta(0) allows the line to account for a non-zero intercept, optimizing the least squares solution by minimizing the error across all data points.

3.467 is theta(0) accounting for vertical Shift

0.369 is theta(1) which is consistent and shows linear trend and slope is 0.369

Assignment 4: Maximum Likelihood Estimation with Features

Instructions: This assignment engages you in the practical application of maximum likelihood estimation with features for linear regression modeling. Follow the steps carefully, and submit your work as a single PDF file that includes all required code and graphs. **Attach** supplementary code files (.py and .ipynb) as used in this assignment.

Objective: Learn how to fit a linear regression model using maximum likelihood estimation with features polynomial and understand the effects of model adjustments, such as adding a degree of polynomial.

Step-by-Step Instructions

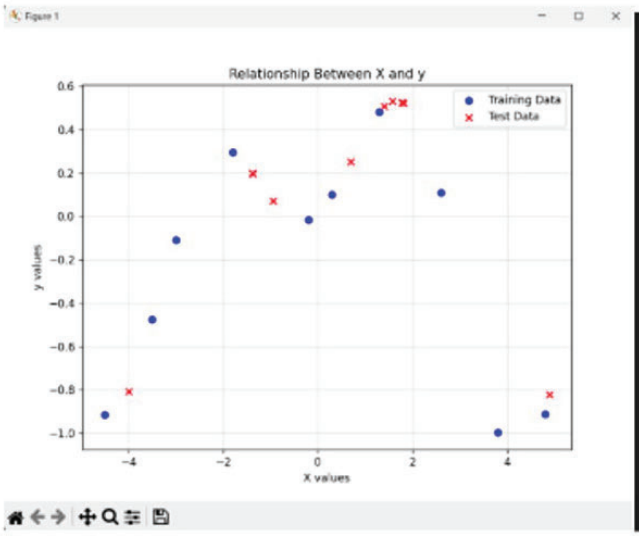
Q1. Maximum Likelihood Estimation with Features

Step 1: Understanding and Plotting the Data

```
import numpy as np
import matplotlib.pyplot as plt

# Data Initialization
X = np.array([-4.5, -3.5, -3, -1.8, -0.2, 0.3, 1.3, 2.6, 3.8, 4.8]).reshape(-1,1)
y = np.array([
    [-0.91650116],
    [-0.47546053],
    [-0.10972425],
    [0.29504095],
    [-0.01596218],
    [0.10014949],
    [0.48104303],
    [0.10979023],
    [-0.99742128],
    [-0.91221826]
]).reshape(-1,1)

X_test = np.array([-3.99, -1.38, -1.37, -0.94,
  0.69,  1.4,   1.57,  1.78,  1.81,  4.89]).reshape(-1,1)
y_test = np.array([
    [-0.80737607],
```

```
    [0.19813376],
    [0.19537639],
    [0.07185977],
    [0.24954213],
    [0.50662504],
    [0.52943298],
    [0.52406997],
    [0.51999057],
    [-0.82318288]
]).reshape(-1,1)
```

- Using the provided code, plot the relationship between variables `x` and `y`.
- Attach the Python code used to generate the graph in your PDF.
- Display the graph you obtain.




```
plt.figure(figsize=(8, 6))
plt.scatter(X, y, color='blue',
            label='Training Data', marker='o')
plt.scatter(X_test, y_test, color='red',
            label='Test Data', marker='x')
plt.title('Relationship Between X and y')
plt.xlabel('X values')
plt.ylabel('y values')
plt.legend()
plt.grid(alpha=0.4)
plt.show()
```

Step 2: Polynomial Feature Transformation

- Implement the poly_features function to transform input data X into polynomial features of degree K.

(Polynomial Regression)

$$\phi(x) = \begin{bmatrix} \phi_0(x) \\ \phi_1(x) \\ \vdots \\ \phi_{K-1}(x) \end{bmatrix} = \begin{bmatrix} 1 \\ x \\ x^2 \\ x^3 \\ \vdots \\ x^{K-1} \end{bmatrix}$$

(Feature Matrix for Second-order Polynomials)

$$\Phi = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & x_N & x_N^2 \end{bmatrix}$$

- Attach the Python code used for the implementation.
- Explain the purpose and the effect of this transformation.

Step 2: Polynomial Feature Transformation

```
#STEP 2

def poly_features(X, degree):
    return np.hstack([
        X**i for i in range(degree + 1)])
```

```
Transformed X (degree 4):
 [[ 1.000000e+00 -4.500000e+00  2.025000e+01 -9.112500e+01  4.100625e+02]
 [ 1.000000e+00 -3.500000e+00  1.225000e+01 -4.287500e+01  1.500625e+02]
 [ 1.000000e+00 -3.000000e+00  9.000000e+00 -2.700000e+01  8.100000e+01]
 [ 1.000000e+00 -1.800000e+00  3.240000e+00 -5.832000e+00  1.049760e+01]
 [ 1.000000e+00 -2.000000e-01  4.000000e-02 -8.000000e-03  1.600000e-03]
 [ 1.000000e+00  3.000000e-01  9.000000e-02  2.700000e-02  8.100000e-03]
 [ 1.000000e+00  1.300000e+00  1.690000e+00  2.197000e+00  2.856100e+00]
 [ 1.000000e+00  2.600000e+00  6.760000e+00  1.757600e+01  4.569760e+01]
 [ 1.000000e+00  3.800000e+00  1.444000e+01  5.487200e+01  2.085136e+02]
 [ 1.000000e+00  4.800000e+00  2.304000e+01  1.105920e+02  5.308416e+02]]
==========
```

Purpose:

Polynomial feature transformation introduces additional columns in the dataset, each representing a higher power of the original feature. This transformation allows the linear regression model to fit data that exhibits nonlinear relationships.

Effect:

Increased Flexibility: Higher polynomial degrees can model complex patterns and fit the training data better.

Risk of Overfitting: Excessively high-degree polynomials may overfit the training data, leading to poor generalization on unseen data.

Regularization Need: It may require additional techniques like regularization to balance model complexity and performance.

Step 3: Fitting the Model Using Maximum Likelihood

- Using the poly_features function from Step 2, transform both training and test datasets for a degree of polynomial 4.
- Apply the nonlinear_features_maximum_likelihood function to estimate model parameters (Theta) with Phi from training data.

$$\theta_{\mathrm{ML}} = (\Phi^\top \Phi)^{-1} \Phi^\top y$$

- Attach the Python code used for the implementation and show their result.

Step 3: Fitting the Model Using Maximum Likelihood:

```
#STEP 3
def fit_model_with_mle(Phi,y):
    return np.linalg.inv(Phi.T @ Phi) @ Phi.T @ y

Phi_train = poly_features(X, degree)
Phi_test = poly_features(X_test, degree)

theta_mle = fit_model_with_mle(Phi_train, y)
y_train_pred = Phi_train @ theta_mle
y_test_pred = Phi_test @ theta_mle
print("=" * 10)
print("Theta: ")
print(theta_mle)
print("=" * 10)
print("y-train : ")
print(y_train_pred)
print("=" * 10)
print("y-test: ")
print(y_test_pred)
```

```
==========
Theta:
[[ 2.89383888e-01]
 [-2.31470379e-02]
 [-5.87255114e-02]
 [ 1.12501954e-03]
 [-1.95778878e-05]]
==========
```

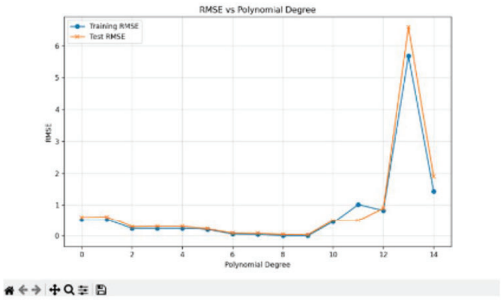
```
==========
y-train :
[[-0.90619161]
 [-0.40016211]
 [-0.20166594]
 [ 0.13401126]
 [ 0.29165524]
 [ 0.2771847 ]
 [ 0.16246238]
 [-0.14890419]
 [-0.58892142]
 [-1.06073227]]
==========
y-test:
[[-0.62960004]
 [ 0.2064623 ]
 [ 0.20791163]
 [ 0.25830253]
 [ 0.24581836]
 [ 0.14488788]
 [ 0.11252528]
 [ 0.06826454]
 [ 0.06155805]
 [-1.10770111]]
```

Step 4: Model Evaluation Using RMSE

- Implement the RMSE function to evaluate the accuracy of your model on both the training and test datasets.
- Calculate RMSE for models with polynomial degrees ranging from 0 to 15.
- Plot these RMSE values against the polynomial degree for both training and test datasets.
- Attach the Python code used for the implementation and generate the graph in your PDF.

```
def calculate_rmse(y_true, y_pred):
    return np.sqrt(np.mean((y_true - y_pred) ** 2))
```

```
degrees = range(0, 15)
rmse_train_list = []
rmse_test_list = []

for deg in degrees:
    # Transform datasets for the current degree
    Phi_train_deg = poly_features(X, deg)
    Phi_test_deg = poly_features(X_test, deg)

    # Estimate model parameters
    theta_deg = fit_model_with_mle(Phi_train_deg, y)

    # Predict on training and test datasets
    y_train_pred_deg = Phi_train_deg @ theta_deg
    y_test_pred_deg = Phi_test_deg @ theta_deg

    # Calculate RMSE
    rmse_train_list.append(calculate_rmse(y, y_train_pred_deg))
    rmse_test_list.append(calculate_rmse(y_test, y_test_pred_deg))

plt.figure(figsize=(10, 6))
plt.plot(degrees, rmse_train_list, label='Training RMSE', marker='o')
plt.plot(degrees, rmse_test_list, label='Test RMSE', marker='x')
plt.title('RMSE vs Polynomial Degree')
plt.xlabel('Polynomial Degree')
plt.ylabel('RMSE')
plt.legend()
plt.grid(alpha=0.4)
plt.show()

print("-" * 10)
print("Lowest RMSE of Train Data: ", min(rmse_train_list), "at degree", rmse_train_list.index(min(rmse_train_list)))
print("-" * 10)
print("Lowest RMSE of Test Data: ", min(rmse_test_list), "at degree", rmse_test_list.index(min(rmse_test_list)))
```




```
==========
Lowest RMSE of Train Data:  3.804822849281283e-10 at degree 9
==========
Lowest RMSE of Test Data:  0.0307246620421302 at degree 9
```

Step 5: Selecting the Best Model

- Identify the polynomial degree that results in the lowest RMSE for the test dataset.
- Using the optimal degree, plot the model's predictions against the original data points.

```
X_test_all = np.linspace(-5, 5, 100).reshape(-1,1)
Phi_test_all = poly_features(X_test_all, optimal_k)
ypred_test_all = Phi_test_all @ theta_optimal
```

- Discuss why this particular model degree was optimal and how model complexity affects overfitting and underfitting.
- Attach the Python code used for the implementation and generate the graph in your PDF.

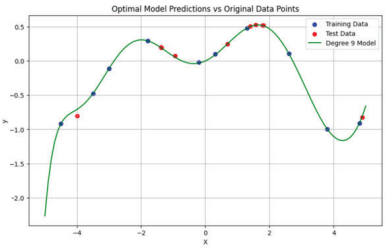
```
#step5
# Define the poly_features function
def poly_features(X, degree):
    return np.hstack([X**i for i in range(1, degree + 1)])
# Define the RMSE function
def rmse(y_true, y_pred):
    return np.sqrt(np.mean((y_true - y_pred) ** 2))
# Define the maximum likelihood function
def nonlinear_features_maximum_likelihood(Phi, y):
    return np.linalg.inv(Phi.T @ Phi) @ (Phi.T @ y)
# Data Initialization
X_train = np.array([-4.5, -3.5, -3, -1.8, -0.2, 0.3, 1.3, 2.6, 3.8, 4.8]).reshape(-1, 1)
y_train = np.array([[-0.91650818],[-0.47544859],[-0.18972425],[0.29584899],[-0.05196218],[0.10016949],[0.48104303],[0.18979023],[-0.99742128],[-0.91221826]]).reshape(-1, 1)

X_test = np.array([-3.99, -1.36, -1.37, -0.94, 0.69, 1.4, 1.57, 1.78, 1.81, 4.89]).reshape(-1, 1)
y_test = np.array([[-0.80737608],[0.19813376],[0.19537639],[0.07185977],[0.24954213],[0.58662584],[0.52943298],[0.52486997],[0.51999057],[-0.82318288]]).reshape(-1, 1)

# Compute RMSE for polynomial degrees 0 to 15
degrees = range(0, 16)
rmse_train_values = []
rmse_test_values = []

for d in degrees:
    # Transform data for the current degree
    Phi_train_d = poly_features(X_train, d) if d > 0 else np.ones_like(X_train)
    Phi_test_d = poly_features(X_test, d) if d > 0 else np.ones_like(X_test)
    # Estimate Theta
    Theta_d = nonlinear_features_maximum_likelihood(Phi_train_d, y_train)
    # Predictions
    y_train_pred_d = Phi_train_d @ Theta_d
    y_test_pred_d = Phi_test_d @ Theta_d
    # Compute RMSE
    rmse_train_values.append(rmse(y_train, y_train_pred_d))
    rmse_test_values.append(rmse(y_test, y_test_pred_d))
# Identify the optimal polynomial degree with the lowest RMSE on the test dataset
optimal_degree = degrees[np.argmin(rmse_test_values)]
print("Optimal Polynomial Degree:", optimal_degree)
# Transform data using the optimal polynomial degree
Phi_train_optimal = poly_features(X_train, optimal_degree)
Phi_test_all = poly_features(np.linspace(-5, 5, 100).reshape(-1, 1), optimal_degree)
# Estimate Theta for the optimal degree
theta_optimal = nonlinear_features_maximum_likelihood(Phi_train_optimal, y_train)
# Predictions for all test points
X_test_all = np.linspace(-5, 5, 100).reshape(-1, 1)
Phi_test_all = poly_features(X_test_all, optimal_degree)
y_pred_test_all = Phi_test_all @ theta_optimal
# Plot predictions against the original data points
plt.figure(figsize=(10, 6))
plt.scatter(X_train, y_train, color='blue', label="Training Data")
plt.scatter(X_test, y_test, color='red', label="Test Data")
plt.plot(X_test_all, y_pred_test_all, color='green', label="Degree {optimal_degree} Model")
plt.title("Optimal Model Predictions vs Original Data Points")
```

Why Degree 9 Was Optimal

- Lowest RMSE: This degree minimizes the test RMSE, indicating the best generalization to unseen data.
- Balanced Complexity: Degree 9 provides sufficient flexibility to capture the underlying patterns in the data without overfitting or underfitting.

Effect of Model Complexity

- Underfitting (Low Degrees): Models with a very low degree (e.g., 1 or 2) are too simple and fail to capture important data patterns, resulting in high RMSE for both training and test datasets.
- Overfitting (High Degrees): Very high-degree models (e.g., 15) fit the training data noise rather than the true relationship, leading to low training RMSE but high test RMSE.
- Optimal Degree: Degree 9 strikes a balance by fitting the data well without capturing unnecessary noise.



Step 6: Conclusion

- Summarize your findings regarding the polynomial regression model's performance.
- Discuss the trade-offs between model complexity (degree of the polynomial) and prediction accuracy.

Step 6: Conclusion

Summary of Findings:

1. **Polynomial Regression Performance**:
    a. Using polynomial features effectively captures the non-linear relationship between X and y.

    b. The optimal degree for the polynomial regression model is **9**, as it achieves the lowest test RMSE while maintaining reasonable generalization.
2. **Trade-offs Between Model Complexity and Prediction Accuracy**:
    a. **Underfitting**: Models with low polynomial degrees (e.g., 0-3) fail to capture the data's complexity, leading to high training and test RMSE.
    b. **Optimal Fit**: Degree 9 strikes a balance, accurately modeling the data while avoiding overfitting.
    c. **Overfitting**: Models with excessively high polynomial degrees (e.g., >10) overfit the training data, as evidenced by lower training RMSE but increased test RMSE.

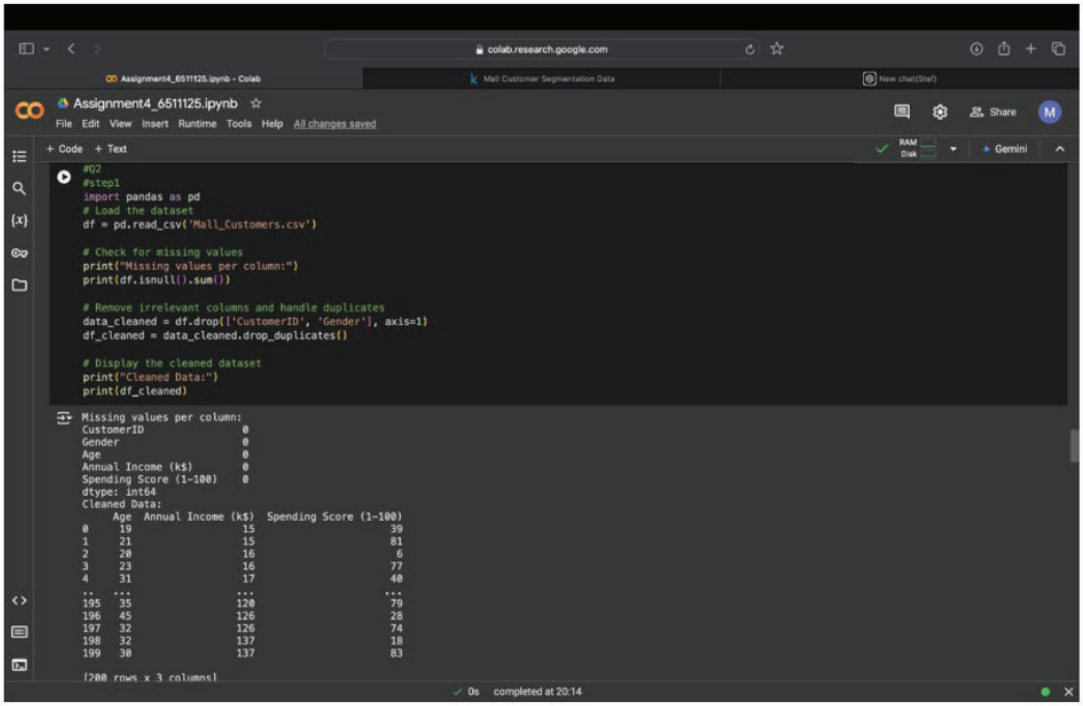
Q2. K-means

In this assignment, you will use a dataset from Kaggle to perform K-means clustering, commonly used for grouping similar data points together. This task will involve several clear steps:

Step 1: Prepare Your Dataset

- Choose a dataset from Kaggle that's good for grouping, like the Mall Customers Dataset (link)(https://www.kaggle.com/vjchoudhary7/customer-segmentation-tutorial-in-python). Clean up the dataset by removing any missing or unnecessary data.
- **Attach your data cleaning code** with your report.

Step 1: Prepare Your Dataset



```
#Q2
#step1
import pandas as pd
# Load the dataset
df = pd.read_csv('Mall_Customers.csv')

# Check for missing values
print("Missing values per column:")
print(df.isnull().sum())

# Remove irrelevant columns and handle duplicates
data_cleaned = df.drop(['CustomerID', 'Gender'], axis=1)
df_cleaned = data_cleaned.drop_duplicates()

# Display the cleaned dataset
print("Cleaned Data:")
print(df_cleaned)
```

```
Missing values per column:
CustomerID                0
Gender                    0
Age                       0
Annual Income (k$)        0
Spending Score (1-100)    0
dtype: int64
Cleaned Data:
     Age  Annual Income (k$)  Spending Score (1-100)
0     19                  15                      39
1     21                  15                      81
2     20                  16                       6
3     23                  16                      77
4     31                  17                      40
..   ...                 ...                     ...
195   35                 120                      79
196   45                 126                      28
197   32                 126                      74
198   32                 137                      18
199   30                 137                      83

[200 rows x 3 columns]
```

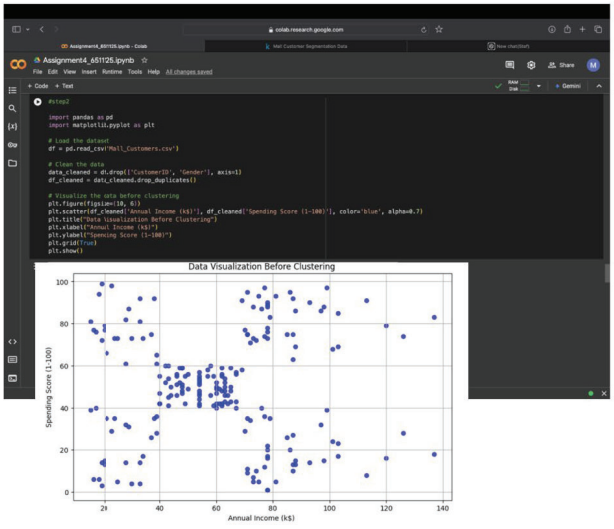
0s completed at 20:14

Clean and remove unnecessary data

Step 2: Visualize Data Before Clustering

Step 2: Visualize Data Before Clustering

- Create a graph to show how your data looks before you apply clustering. This helps you see what the data looks like normally.
- **Attach your plotting code** with your report.

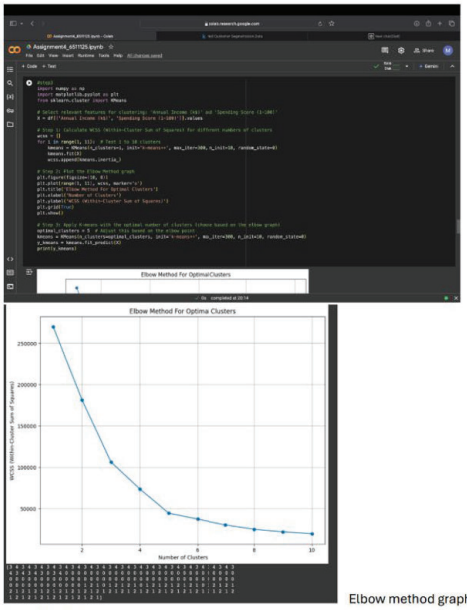


```
#step2
import pandas as pd
import matplotlib.pyplot as plt

# Load the dataset
df = pd.read_csv('Mall_Customers.csv')

# Clean the data
data_cleaned = df.drop(['CustomerID', 'Gender'], axis=1)
df_cleaned = data_cleaned.drop_duplicates()

# Visualize the data before clustering
plt.figure(figsize=(10, 6))
plt.scatter(df_cleaned['Annual Income (k$)'], df_cleaned['Spending Score (1-100)'], color='blue', alpha=0.7)
plt.title("Data Visualization Before Clustering")
plt.xlabel("Annual Income (k$)")
plt.ylabel("Spending Score (1-100)")
plt.grid(True)
plt.show()
```

Step 3: Implement K-means Clustering and Determine Optimal Clusters

- Use the K-means algorithm to find groups in your data. To figure out the best number of groups, use the elbow method: plot the within-cluster sum of squares (WCSS) against different numbers of clusters and look for the point where the line starts to flatten out.
- **Attach your K-means and elbow method code** with your report.

Step 3: Implement K-means Clustering and Determine Optimal Clusters

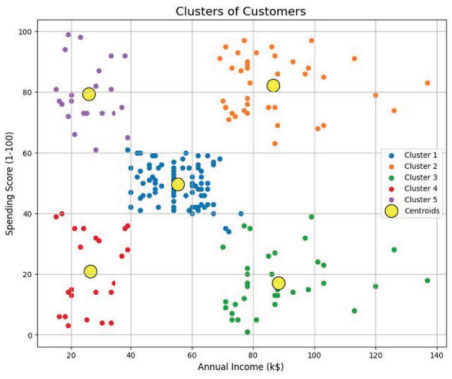


Elbow method graph with kmeans output

Step 4: Visualize Data After Clustering

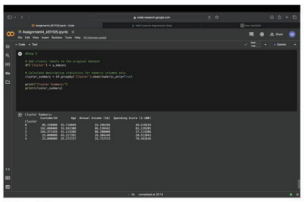
- Make another graph to show your data with the groups found by K-means. This shows how the algorithm has organized the data.
- **Attach your post-clustering plotting code** with your report.

```
#step4
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Select features for clustering
X = df[['Annual Income (k$)', 'Spending Score (1-100)']].values

# Apply K-means with the optimal number of clusters (based on elbow method)
optimal_clusters = 5  # Replace this with your chosen value
kmeans = KMeans(n_clusters=optimal_clusters, init='k-means++', max_iter=300, n_init=10, random_state=0)
y_kmeans = kmeans.fit_predict(X)

# Visualize the clusters
plt.figure(figsize=(10, 8))

# Plot each cluster
for cluster in range(optimal_clusters):
    plt.scatter(X[y_kmeans == cluster, 0], X[y_kmeans == cluster, 1], label=f'Cluster {cluster+1}')

# Mark cluster centroids
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
            s=300, c='yellow', label='Centroids', edgecolor='k')

# Add plot details
plt.title('Clusters of Customers', fontsize=16)
plt.xlabel('Annual Income (k$)', fontsize=12)
plt.ylabel('Spending Score (1-100)', fontsize=12)
plt.legend()
plt.grid(True)
plt.show()
```



Step 5: Interpret Your Results

- Explain what the results from the K-means clustering show about your data. Discuss how this information can help identify specific groups within the dataset.
- **Attach your interpretation and any additional code** you used to analyze the results.



Additional Code

Cluster 4 (Low Income, High Spending Score):

Customers with low annual income (~25k) but high spending scores (~79).

These could be impulsive shoppers who spend heavily despite limited income. Great candidates for discount and promotional offers.

How This Information Can Be Used:

1. **Personalized Marketing:**
   - Create targeted campaigns for high-spending clusters to encourage loyalty.
   - Offer discounts or budget-friendly promotions for low-income, low-spending groups.
2. **Product Placement:**
   - Tailor the placement of premium products to high-income, high-spending customers.
   - Focus on cost-effective products for budget-conscious groups.
3. **Improved Customer Engagement:**
   - Use insights to design events, offers, or memberships to retain and engage specific groups.

4. **Business Strategy:**
   - Allocate resources effectively by prioritizing profitable groups.
   - Analyze why certain groups (e.g., high income, low spending) are disengaged.

Step 5: Interpret Your Results

Interpretation of Results from K-means Clustering

The K-means algorithm grouped customers based on their Annual Income (k$) and Spending Score (1-100) into distinct clusters:

Cluster 0 (Average Income, Moderate Spending Score):

Customers with moderate annual income (~55k) and moderate spending scores (~49).

Likely represent typical mall visitors who neither overspend nor underspend.

Cluster 1 (High Income, High Spending Score):

Customers with high annual income (~86k) and high spending scores (~82).

These are wealthy, highly engaged shoppers, making them a profitable group for targeted marketing campaigns.

Cluster 2 (High Income, Low Spending Score):

Customers with high annual income (~88k) but low spending scores (~17).

Likely disengaged customers who have spending potential but are not utilizing mall services. Ideal for re-engagement campaigns.

Cluster 3 (Low Income, Low Spending Score):

Customers with low annual income (~26k) and low spending scores (~21).

Likely budget-conscious or infrequent shoppers.

Assignment 5: Practical Application of DB-SCAN and A* Algorithm.

Instructions: This assignment engages you in the practical application of DB-SCAN and A* Algorithm. Follow the steps carefully, and submit your work as a single PDF file that includes all required code and graphs. **Attach** supplementary code files (.py and .ipynb) as used in this assignment.

Objective: Learn DB-SCAN and A* Algorithm and understand the effects of DB-SCAN and A* Algorithm in any problem.

Step-by-Step Instructions

Q1. DB-SCAN

Step 1: Visualize Data Before Clustering

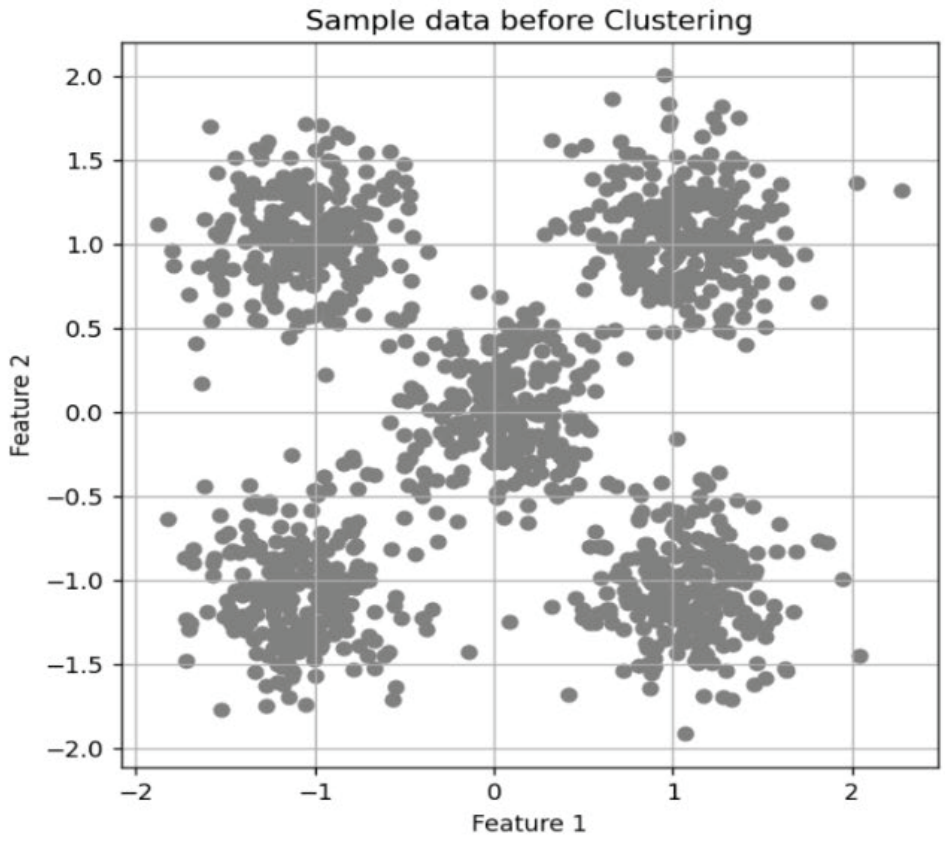
Using the provided code,

```
# ====================== Initial ================================
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler

# Generate sample data
centers = [[2, 2], [-2, -2], [2, -2], [-2, 2], [0, 0]]
X, _ = make_blobs(n_samples= 1250, centers=centers, cluster_std=0.55,
random_state=2)
X = StandardScaler().fit_transform(X)

# Plotting before clustering
plt.figure(figsize=(6, 6))
plt.scatter(X[:, 0], X[:, 1], color='gray', marker='o')
plt.title('Sample Data Before Clustering')
plt.grid(True)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
# ============================================================
```

- Create a graph to show how your data looks before you apply clustering. This helps you see what the data looks like normally.
- **Attach your plotting code** with your report.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler


centers=[[2,2],[-2,-2], [2,-2], [-2,2], [0,0]]

X,_ = make_blobs(
    n_samples=1250, centers=centers, cluster_std=0.55, random_state=2
)

X = StandardScaler().fit_transform(X)

plt.figure(figsize=(6,6))
plt.scatter(X[:,0], X[:,1], c='gray', marker='o')
plt.title('Sample data before Clustering')
plt.grid(True)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```

Figure 1



Step 2: Implement K-means Clustering and Determine Optimal Clusters

- Use the K-means algorithm to find groups in your data. To figure out the best number of groups, use the elbow method: **plot** the within-cluster sum of squares (WCSS) against different numbers of clusters and look for the point where the line starts to flatten out.
- **Attach your K-means and elbow method code** with your report.

Elbow Method Example Code:

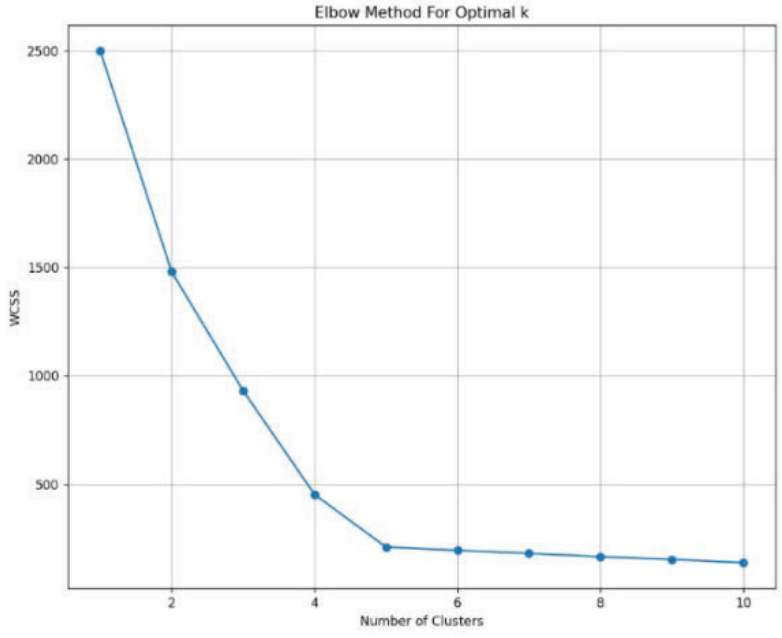
==========

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

# Generate synthetic data
X, _ = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)

# Calculate WCSS for different numbers of clusters
wcss = []
for i in range(1, 11):  # Testing 1 to 10 clusters
    kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=300, n_init=10,
random_state=0)
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)  # inertia_ is the WCSS for the model

# Plotting the WCSS to observe the 'Elbow'
plt.figure(figsize=(10, 8))
plt.plot(range(1, 11), wcss, marker='o')
plt.title('Elbow Method For Optimal k')
plt.xlabel('Number of Clusters')
plt.ylabel('WCSS')
plt.grid(True)
plt.show()
```

==========

```python
from sklearn.cluster import KMeans
plt.figure(figsize=(6,6))
plt.scatter(X[:,0], X[:,1], c='gray', marker='o')
plt.title('Sample data before Clustering')
plt.grid(True)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()

wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=300,
                    n_init=10, random_state=0)
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)


plt.figure(figsize=(10, 8))
plt.plot(range(1, 11), wcss, marker='o')
plt.title('Elbow Method For Optimal k')
plt.xlabel('Number of Clusters')
plt.ylabel('WCSS')
plt.grid(True)
plt.show()
```

Figure 1

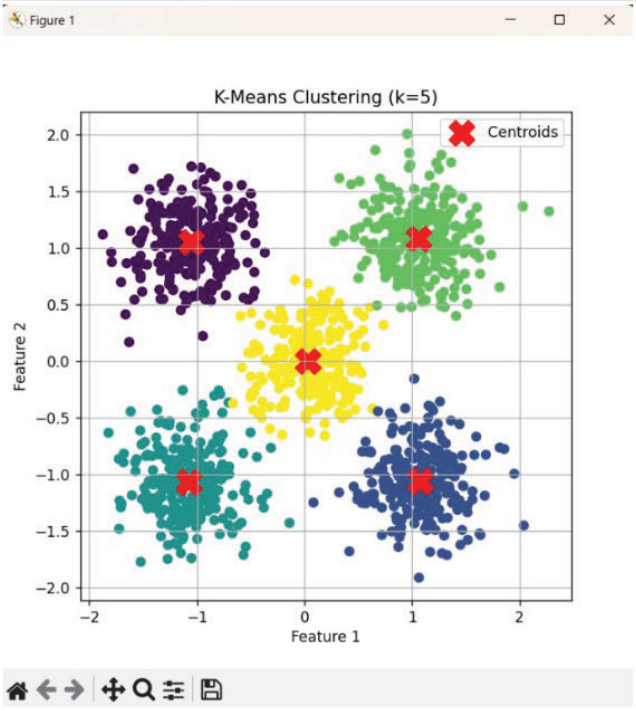


Optimal Clusters is 5

Step 3: Visualize Data After Clustering

- Make another graph to show your data with the groups found by best K-means with K value from step2. This shows how the algorithm has organized the data.
- **Attach your post-clustering plotting code** with your report.

```
optimal_clusters = 5

kmeans = KMeans(n_clusters=optimal_clusters, init='k-means++',
                max_iter=300, n_init=10, random_state=0)
y_kmeans = kmeans.fit_predict(X)


plt.figure(figsize=(6, 6))
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, cmap='viridis', marker='o')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
            s=300, c='red', marker='X', label='Centroids')
plt.title(f'K-Means Clustering (k={optimal_clusters})')
plt.grid(True)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.show()
```

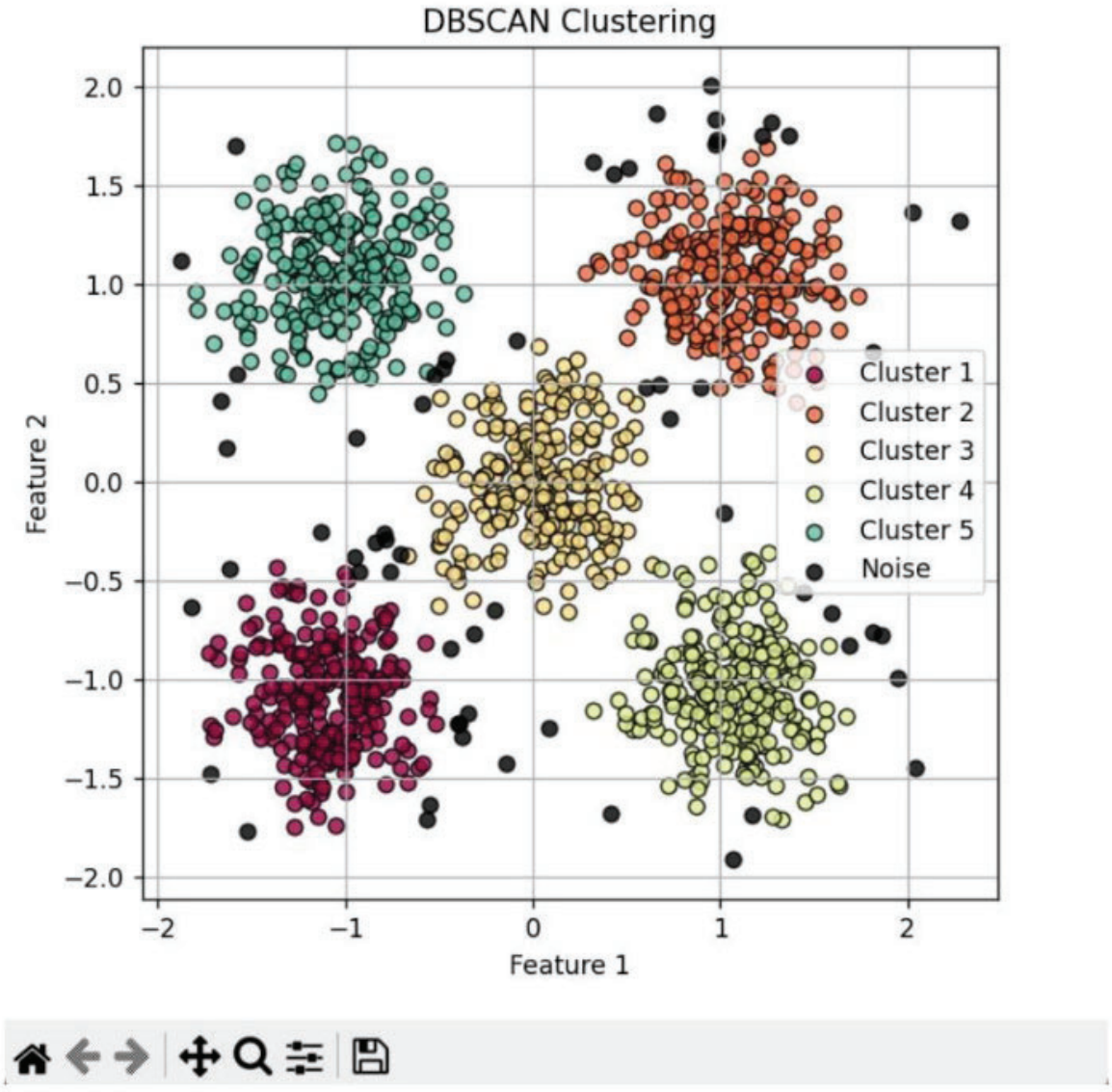


Step 4: Applied data from step1 with DB-SCAN

- Use the DB-SCAN algorithm to identify clusters in your data. Set the epsilon parameter to 0.19 and the minimum number of points required to form a dense region (min_samples) to 9.
- Make another graph to show your data with the groups found by DB-SCAN. This shows how the algorithm has organized the data.
- **Attach your post-clustering plotting code** with your report.

```
#DBSCAN

def euclidean_distance(p1, p2):
    return np.sqrt(np.sum((p1 - p2) ** 2))

# Region query to find neighbors within epsilon distance
def region_query(X, point_idx, eps):
    neighbors = []
    for idx, point in enumerate(X):
        if euclidean_distance(X[point_idx], point) <= eps:
            neighbors.append(idx)
    return neighbors

# Expanding a cluster from a core point
def expand_cluster(X, labels, point_idx, cluster_id, eps, min_samples):
    neighbors = region_query(X, point_idx, eps)
    if len(neighbors) < min_samples:
        labels[point_idx] = -1  # Mark as noise
        return False
    else:
        labels[point_idx] = cluster_id
        queue = neighbors
        while queue:
            current_idx = queue.pop(0)  # Process the first element in the queue
            if labels[current_idx] == -1:  # If it was marked as noise, change to
cluster_id
                labels[current_idx] = cluster_id
            elif labels[current_idx] == 0:  # If not visited
                labels[current_idx] = cluster_id
                current_neighbors = region_query(X, current_idx, eps)
                if len(current_neighbors) >= min_samples:
                    queue.extend(current_neighbors)  # Add neighbors to the queue
        return True

# DBSCAN function implemented from scratch
def scratch_dbscan(X, eps, min_samples):
```

```
    labels = np.zeros(len(X), dtype=int)  # Initialize all points as unvisited
(label 0)
    cluster_id = 0
    for point_idx in range(len(X)):
        if labels[point_idx] == 0:  # If not visited
            if expand_cluster(X, labels, point_idx, cluster_id + 1, eps,
min_samples):
                cluster_id += 1  # Increment the cluster ID for each new cluster
    return labels

eps = 0.19
min_samples = 9
labels = scratch_dbscan(X, eps, min_samples)

# Plotting the DBSCAN results
plt.figure(figsize=(6, 6))
unique_labels = set(labels)
colors = [plt.cm.Spectral(each) for each in np.linspace(0, 1,
len(unique_labels))]

for k, col in zip(unique_labels, colors):
    if k == -1:
        col = [0, 0, 0, 1]  # Black for noise
        label_name = "Noise"
    else:
        label_name = f"Cluster {k}"
    class_member_mask = (labels == k)
    xy = X[class_member_mask]
    plt.scatter(xy[:, 0], xy[:, 1], c=[col], edgecolors='k', alpha=0.8,
label=label_name)

plt.title('DBSCAN Clustering')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend(loc='best')
plt.grid(True)
plt.show()
```



Step 5: Interpret Your Results

- Explain the different the results from the K-means clustering show about your data and DBSCAN. What is pros and cons of each algorithm.
- **Attach your interpretation and any additional code** you used to analyze the results.

In both K MEANS and DBSCAN, there are 5 clusters, however, in K Means every point needs to be a part of a cluster, but in DBSCAN it does not and this is an advantage in DBSCAN as it allows the noise to not be a part of any of the cluster. What this means in AI, is that if there are rows or features or elements that are outliers, DBSCAN can mark these are noise while K MEANS will make it a part of a cluster making the predictions less accurate.

Q2. A* algrorithm

*Task: Implement the A Algorithm for the 8-Puzzle Problem**

STEP 1:

Task Setup:

- Define a Node class to encapsulate the **state of the puzzle**, its **parent state**, and the cost: g, h, and f.
    - **g (cost):** Represents the cost from the start node to the current node, where each move incurs a cost of one.
    - **h (heuristic):** The estimated cost from the current node to the goal, calculated using the Manhattan Distance between the current state and the goal state.
    - **f (total cost):** The sum of g and h, determining the path priority in the A* algorithm.

Implementation Details:

- Start with the initial puzzle state: ((1, 2, 3), (4, 0, 6), (7, 5, 8)).
- Set the goal state as: ((1, 2, 3), (4, 5, 6), (7, 8, 0)).
- Implement the A* search to explore paths, expand nodes based on the lowest f value, and efficiently find the shortest path to the goal state.
- Attach your python code and your result that your get.

Main Function for Running:

```
def main():
    """Main function to run A* algorithm on a given 8-puzzle problem."""
    start = (
        (1, 2, 3),
        (4, 0, 6),
```

```
        (7, 5, 8)
    )

    end = (
        (1, 2, 3),
        (4, 5, 6),
        (7, 8, 0)
    )

    path, total_cost = astar(start, end)

    if path:
        print(f"Solution found in {len(path) - 1} moves with total cost
{total_cost}:\n")
        for i, board in enumerate(path):
            print(f"Move {i}:")
            print_board(board)
    else:
        print("No solution found.")

if __name__ == '__main__':
    main()
```

Example code for Manhattan distance:

```
def manhattan_distance(start_state, goal_state):
    """Calculate the Manhattan Distance heuristic for the 8-puzzle."""
    distance = 0
    for num in range(1, 9):  # Loop through numbers 1 to 8 (tiles of the 8-
puzzle)
        # Find the position of the number in the start state
        x1, y1 = -1, -1
        for ix, row in enumerate(start_state):
            for iy, value in enumerate(row):
                if value == num:
                    x1, y1 = ix, iy
                    break
            if x1 != -1:
                break

        # Find the position of the number in the goal state
        x2, y2 = -1, -1
        for ix, row in enumerate(goal_state):
            for iy, value in enumerate(row):
                if value == num:
```

```
                    x2, y2 = ix, iy
                    break
            if x2 != -1:
                break

        # Calculate the Manhattan distance for this number
        distance += abs(x1 - x2) + abs(y1 - y2)
    return distance
```

Result Expectation:

Step 1: Task Setup

```python
class Node:
    def __init__(self, state, parent=None, g=0, h=0):
        self.state = state
        self.parent = parent
        self.g = g
        self.h = h
        self.f = g + h

    def __lt__(self, other):
        return self.f < other.f
```

```python
def manhattan_distance(start_state, goal_state):
    """Calculate the Manhattan Distance heuristic for the 8-puzzle."""
    distance = 0
    for num in range(1, 9):  # Loop through numbers 1 to 8 (tiles of the 8-puzzle)
        # Find the position of the number in the start state
        x1, y1 = -1, -1
        for ix, row in enumerate(start_state):
            for iy, value in enumerate(row):
                if value == num:
                    x1, y1 = ix, iy
                    break
            if x1 != -1:
                break

        # Find the position of the number in the goal state
        x2, y2 = -1, -1
        for ix, row in enumerate(goal_state):
            for iy, value in enumerate(row):
                if value == num:
                    x2, y2 = ix, iy
                    break
            if x2 != -1:
                break

        # Calculate the Manhattan distance for this number
        distance += abs(x1 - x2) + abs(y1 - y2)
    return distance
```

```python
def main():

    start = (
        (1, 2, 3),
        (4, 0, 6),
        (7, 5, 8)
    )

    end = (
        (1, 2, 3),
        (4, 5, 6),
        (7, 8, 0)
    )

    distance = manhattan_distance(start, end)
    print()
    print(f"The Manhattan Distance between the start and goal states is: {distance}")
    print()
if __name__ == '__main__':
    main()
```

```
The Manhattan Distance between the start and goal states is: 2
```

Application of A* Search Algorithm

```python
import heapq

def generate_neighbors(state):
    """Generate valid neighbors by sliding tiles."""
    neighbors = []
    rows, cols = len(state), len(state[0])
    zero_pos = [(i, j) for i, row in enumerate(state) for j, value in enumerate(row) if value == 0][0]
    x, y = zero_pos

    # (Up, Down, Left, Right)
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < rows and 0 <= ny < cols:
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state))

    return neighbors
```

```python
def astar(start_state, goal_state):
    """A* search algorithm for solving the 8-puzzle problem."""
    open_list = []
    closed_set = set()

    start_node = Node(state=start_state, g=0, h=manhattan_distance(start_state, goal_state))
    heapq.heappush(open_list, (start_node.f, start_node))

    while open_list:
        _, current_node = heapq.heappop(open_list)

        # Check if the goal state is reached
        if current_node.state == goal_state:
            path = []
            current = current_node
            while current:
                path.append(current.state)
                current = current.parent
            return path[::-1], current_node.g

        closed_set.add(current_node.state)

        # Generate neighbors
        for neighbor_state in generate_neighbors(current_node.state):
            if neighbor_state in closed_set:
                continue

            g = current_node.g + 1
            h = manhattan_distance(neighbor_state, goal_state)
            neighbor_node = Node(parent=current_node, state=neighbor_state, g=g, h=h)

            # Check if neighbor is already in open list with a lower cost
            if any(open_node.state == neighbor_node.state and open_node.g <= neighbor_node.g for _, open_node in open_list):
                continue

            heapq.heappush(open_list, (neighbor_node.f, neighbor_node))

    return None, 0  # No solution found
```

```
def print_board(state):
    """Print the 8-puzzle board."""
    for row in state:
        print(" ".join(str(x) if x != 0 else " " for x in row))
    print()
def main():

    start = (
        (1, 2, 3),
        (4, 0, 6),
        (7, 5, 8)
    )

    end = (
        (1, 2, 3),
        (4, 5, 6),
        (7, 8, 0)
    )

    distance = manhattan_distance(start, end)
    print()
    print(f"The Manhattan Distance between the start and goal states is: {distance}")
    print()
    print("Addition of A*Search:")
    print()
    path, total_cost = astar(start, end)

    if path:
        print(f"Solution found in {len(path) - 1} moves with total cost {total_cost}:\n")
        for i, board in enumerate(path):
            print(f"Step {i}:")
            print_board(board)
    else:
        print("No solution found.")
```

Output:

```
Solution found in 2 moves with total cost 2:

Step 0:
1 2 3
4   6
7 5 8

Step 1:
1 2 3
4 5 6
7   8

Step 2:
1 2 3
4 5 6
7 8
```

STEP 2:

Testing:

Test the A* algorithm with alternative start points, such as: ((2, 5, 4), (0, 8, 3), (1, 7, 6)).

```
def main():
    """Main function to run A* algorithm on a given 8-puzzle problem."""
    start = (
        (2, 5, 4),
        (0, 8, 3),
        (1, 7, 6)
    )
```

```
    end = (
        (1, 2, 3),
        (4, 5, 6),
        (7, 8, 0)
    )

    path, total_cost = astar(start, end)

    if path:
        print(f"Solution found in {len(path) - 1} moves with total cost
{total_cost}:\n")
        for i, board in enumerate(path):
            print(f"Move {i}:")
            print_board(board)
    else:
        print("No solution found.")

if __name__ == '__main__':
    main()
```

Result Expectation:

- Document how many moves were needed to reach the goal state for each test case.
- Include all code snippets, testing scenarios, and results in your report.

Step 2: Testing

```
start2 = (
    (2, 5, 4),
    (0, 8, 3),
    (1, 7, 6)
)

end2 = (
    (1, 2, 3),
    (4, 5, 6),
    (7, 8, 0)
)
```

```
distance2 = manhattan_distance(start2, end2)
print()
print(f"The Manhattan Distance between the start and goal states is: {distance2}")
print()
print("Addition of A*Search:")
print()
path2, total_cost2 = astar(start2, end2)

if path2:
    print(f"Solution found in {len(path2) - 1} moves with total cost {total_cost2}:\n")
    for i, board in enumerate(path2):
        print(f"Step {i}:")
        print_board(board)
else:
    print("No solution found.")
```

Solution found in 13 moves

```
The Manhattan Distance between the start and goal states is: 11

Addition of A*Search:

Solution found in 13 moves with total cost 13:
```

```
Step 0:
2 5 4
  8 3
1 7 6

Step 1:
2 5 4
1 8 3
  7 6

Step 2:
2 5 4
1 8 3
7   6

Step 3:
2 5 4
1   3
7 8 6

Step 4:
2   4
1 5 3
7 8 6

Step 5:
2 4
1 5 3
7 8 6

Step 6:
2 4 3
1 5
7 8 6

Step 7:
2 4 3
1   5
7 8 6

Step 8:
2   3
1 4 5
7 8 6

Step 9:
  2 3
1 4 5
7 8 6

Step 10:
1 2 3
  4 5
7 8 6

Step 11:
1 2 3
4   5
7 8 6

Step 12:
1 2 3
4 5
7 8 6

Step 13:
1 2 3
4 5 6
7 8
```

Assignment 6: Gradient Descent for Linear Regression

Objective:
Learn how to fit a linear regression model ug gradient descent.

Instructions:
This assignment engages you in the practical application of gradient descent for linear regression modeling. Follow the steps carefully, and submit your work as a gle PDF file that includes all required code and graphs. Attach supplementary code files (.py and .ipynb) as used in this assignment.

Step-by-Step Instructions

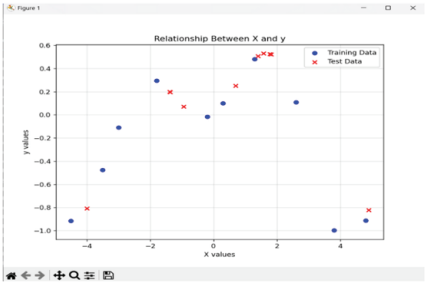
Question 1: Linear Regression Model ug Maximum Likelihood Estimation

1.1 Plotting Training and Testing Data
Base this task on Assignment 4, Question 1.

```
# ================================================

import numpy as np
import matplotlib.pyplot as plt

# Data Initialization
X = np.array([-4.5, -3.5, -3, -1.8, -0.2, 0.3, 1.3, 2.6, 3.8, 4.8]).reshape(-1,1)
y = np.array([
    [-0.91650116],
    [-0.47546053],
    [-0.10972425],
    [0.29504095],
    [-0.01596218],
    [0.10014949],
    [0.48104303],
    [0.10979023],
    [-0.99742128],
    [-0.91221826]
]).reshape(-1,1)

X_test = np.array([-3.99, -1.38, -1.37, -0.94,
  0.69,  1.4,   1.57,  1.78,  1.81,  4.89]).reshape(-1,1)
y_test = np.array([
    [-0.80737607],
    [0.19813376],
    [0.19537639],

    [0.07185977],
    [0.24954213],
    [0.50662504],
    [0.52943298],
    [0.52406997],
    [0.51999057],
    [-0.82318288]
]).reshape(-1,1)

#================================================
```

- **Task:** Plot the relationship between the training data (X, y) and the testing data (X_test, y_test) on the same graph ug distinct colors to differentiate between the two sets.
- **Requirements:**
  - Attach the Python code used to generate the graph in your PDF.
  - Display the graph, ensuring clarity by labeling axes and adding a legend.

```
X = np.array([-4.5, -3.5, -3, -1.8, -0.2, 0.3,1.3, 2.6, 3.8, 4.8]).reshape(-1,1)
y = np.array( [
[-0.91650116],
[-0.47546053],
[-0.10972425],
[0.29504095],
[-0.01596218],
[0.10014949],
[0.48104303],
[0.10979023],
[-0.99742128],
[-0.91221826]
]).reshape(-1,1)

X_test = np.array([-3.99, -1.38, -1.37, -0.94,
0.69, 1.4, 1.57, 1.78, 1.81, 4.89]).reshape(-1,1)
y_test = np.array([
[-0.80737607],[0.19813376],[0.19537639],
[0.07185977],[0.24954213],[0.50662504],
[0.52943298],
[0.52406997],
[0.51999057],
[-0.82318288]
]).reshape(-1,1)


plt.figure(figsize=(8, 6))
plt.scatter(X, y, color='blue',
            label='Training Data', marker='o')
plt.scatter(X_test, y_test, color='red',
            label='Test Data', marker='x')
plt.title('Relationship Between X and y')
plt.xlabel('X values')
plt.ylabel('y values')
plt.legend()
plt.grid(alpha=0.4)
plt.show()
```



1.2 Polynomial Feature Transformation

- **Task:** Implement the poly_features function to transform input data X into polynomial features of degree K.

(Polynomial Regression)

$$\phi(x) = \begin{bmatrix} \phi_0(x) \\ \phi_1(x) \\ \vdots \\ \phi_{K-1}(x) \end{bmatrix} = \begin{bmatrix} 1 \\ x \\ x^2 \\ x^3 \\ \vdots \\ x^{K-1} \end{bmatrix}$$

(Feature Matrix for Second-order Polynomials)

$$\Phi = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & x_N & x_N^2 \end{bmatrix}$$

- **Requirements:**
  - Attach the Python code used for the implementation.
  - Explain the purpose and effect of this transformation.

```python
def poly_features(X, degree):
    return np.hstack([
        X**i for i in range(degree + 1)])

degree = 5
X_poly = poly_features(X, degree)
X_poly_test = poly_features(X_test, degree)
print(f"Transformed X (degree {degree}):\n", X_poly)
```

The purpose of this is to add the bias term and expand feature to a certain degree to fit polynomial regression models which are non-linear. It expands the 'X' to a new feature matrix where each column is X^(column number). First column number is counted as 0, becoming the bias.

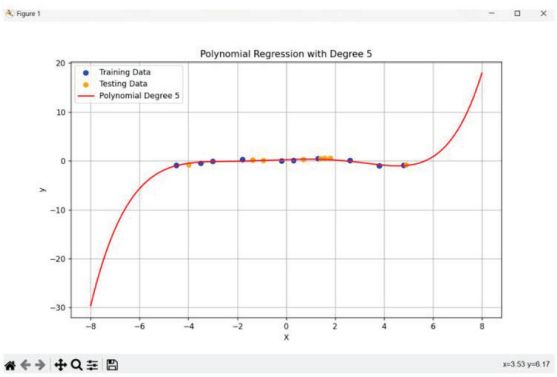
1.3 Fitting the Model Ug Maximum Likelihood

- **Task:** Describe the process of fitting a model ug the Maximum Likelihood Estimation (MLE) method. Transform both training and test datasets ug the poly_features function for a polynomial degree of 5.
- **Formula:**

$$\theta_{ML} = (\Phi^T\Phi)^{-1}\Phi^T y$$

- **Requirements:**
  - Attach Python code used, display calculated values of θ_{ML}, and include a graph that shows both data sets along with predictions for all x values from -8 to 8.

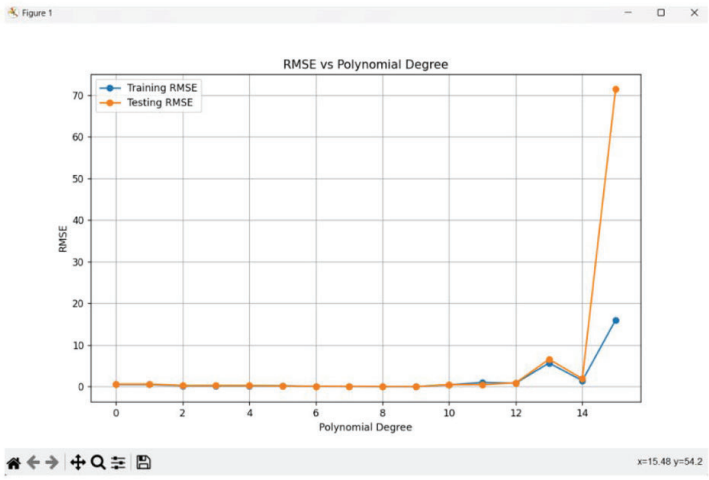
```python
theta_mle = np.linalg.inv(X_poly.T @ X_poly) @ X_poly.T @ y

# Predictions for plotting
x_range = np.linspace(-8, 8, 100).reshape(-1, 1)
x_range_poly = poly_features(x_range, degree)
y_pred = x_range_poly @ theta_mle

# Plotting
plt.figure(figsize=(10, 6))
plt.scatter(X, y, color='blue', label='Training Data')
plt.scatter(X_test, y_test, color='orange', label='Testing Data')
plt.plot(x_range, y_pred, color='red', label=f'Polynomial Degree {degree}')
plt.xlabel('X')
plt.ylabel('y')
plt.title(f'Polynomial Regression with Degree {degree}')
plt.legend()
plt.grid()
plt.show()
```



1.4 Model Evaluation

- **Task:** Use the Root Mean Square Error (RMSE) to evaluate the model accuracy.
  - Calculate and plot RMSE for polynomial degrees ranging from 0 to 15 for both training and test datasets.
  - Determine and discuss the optimal polynomial degree based on the test set RMSE.
- **Requirements:**
  - Attach Python code and generate the graph in your PDF.

```python
def calculate_rmse(y_true, y_pred):
    return np.sqrt(np.mean((y_true - y_pred)**2))

# Initialize variables
max_degree = 15
train_rmse = []
test_rmse = []

# Loop through polynomial degrees
for degree in range(max_degree + 1):
    # Polynomial feature transformation
    X_poly = poly_features(X, degree)
    X_poly_test = poly_features(X_test, degree)

    # Fit the model using MLE
    theta_mle = np.linalg.inv(X_poly.T @ X_poly) @ X_poly.T @ y

    # Predictions
    y_train_pred = X_poly @ theta_mle
    y_test_pred = X_poly_test @ theta_mle

    # Calculate RMSE
    train_rmse.append(calculate_rmse(y, y_train_pred))
    test_rmse.append(calculate_rmse(y_test, y_test_pred))

# Plotting RMSE vs Polynomial Degree
plt.figure(figsize=(10, 6))
plt.plot(range(max_degree + 1), train_rmse, label='Training RMSE', marker='o')
plt.plot(range(max_degree + 1), test_rmse, label='Testing RMSE', marker='o')
plt.xlabel('Polynomial Degree')
plt.ylabel('RMSE')
plt.title('RMSE vs Polynomial Degree')
plt.legend()
plt.grid()
plt.show()
```




```
Optimal Polynomial Degree: 9
```

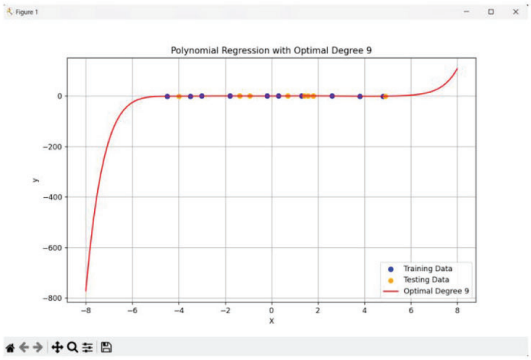
1.5 Selecting the Best Model

- **Task:** Identify and justify the polynomial degree that results in the lowest RMSE for the test dataset.
- **Requirements:**
  - Include a graph that shows both training and test sets, along with predictions for all x values from -8 to 8.
  - Attach Python code used for the implementation and generate the graph in your PDF.

```python
final_train_rmse = calculate_rmse(y, y_train_pred_optimal)
final_test_rmse = calculate_rmse(y_test, y_test_pred_optimal)

# Display Final RMSE
print(f"Final RMSE (Training Data, Degree {optimal_degree}): {final_train_rmse:.4f}")
print(f"Final RMSE (Testing Data, Degree {optimal_degree}): {final_test_rmse:.4f}")
```

```
Final RMSE (Training Data, Degree 9): 0.0000
Final RMSE (Testing Data, Degree 9): 0.0307
```

```python
# Print the optimal degree based on test RMSE
optimal_degree = np.argmin(test_rmse)
print(f"Optimal Polynomial Degree: {optimal_degree}")

optimal_degree = np.argmin(test_rmse)

# Transform data using optimal degree
X_poly_optimal = poly_features(X, optimal_degree)
X_poly_test_optimal = poly_features(X_test, optimal_degree)

# Fit the model using MLE for the optimal degree
theta_mle_optimal = np.linalg.inv(X_poly_optimal.T @ X_poly_optimal) @ X_poly_optimal.T @ y

# Predictions for plotting
x_range = np.linspace(-8, 8, 100).reshape(-1, 1)
x_range_poly = poly_features(x_range, optimal_degree)
y_pred_optimal = x_range_poly @ theta_mle_optimal

# Predictions for training and testing datasets
y_train_pred_optimal = X_poly_optimal @ theta_mle_optimal
y_test_pred_optimal = X_poly_test_optimal @ theta_mle_optimal

# Plotting
plt.figure(figsize=(10, 6))
plt.scatter(X, y, color='blue', label='Training Data')
plt.scatter(X_test, y_test, color='orange', label='Testing Data')
plt.plot(x_range, y_pred_optimal, color='red', label=f'Optimal Degree {optimal_degree}')
plt.xlabel('X')
plt.ylabel('y')
plt.title(f'Polynomial Regression with Optimal Degree {optimal_degree}')
plt.legend()
plt.grid()
plt.show()

# Display theta values
print(f"Optimal Theta Values (Degree {optimal_degree}):\n", theta_mle_optimal)
```



Question 2: Gradient Descent for Polynomial Regression

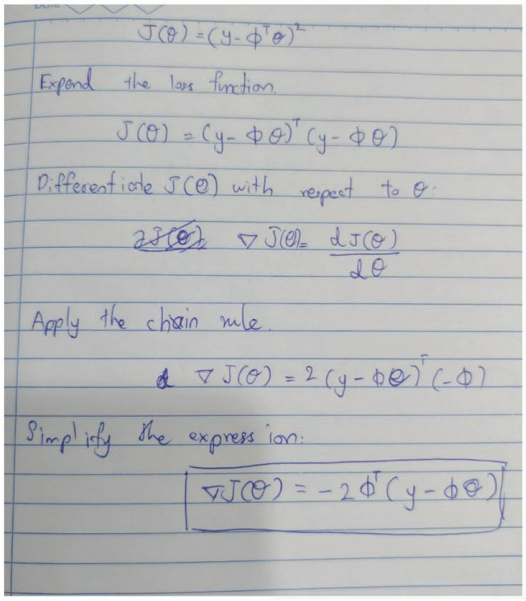
2.1 Mathematical Derivation

- **Task:** Derive the formula for the gradient of the Sum of square loss function used in polynomial regression. Show step-by-step differentiation of $J(\theta)$ with respect to θ.
- **Formula:**

$$J(\theta) = (y - \Phi^T\theta)^2$$

Step 1

Mathematical Derivation



2.2 Feature Transformation and Gradient Descent Implementation

- **Task:** Transform both training and test datasets into the e-modified polynomial feature space ug the optimal degree identified in Q1.
  - Code the Gradient Descent algorithm to minimize the MSE. Use a learning rate of 0.0002 and iterate up to 100,000 times. Implement feature normalization to aid in convergence.
  - Implement feature normalization example code:

```
#======================

def normalize_features(Phi):
    """ Normalize features to have zero mean and unit variance """
    mu = np.mean(Phi, axis=0)
    sigma = np.std(Phi, axis=0)
    Phi_normalized = (Phi - mu) / sigma
    return Phi_normalized, mu, sigma

Phi = poly_features(x_train, optimal_k)
Phi_norm, mu, sigma = normalize_features(Phi)

#======================
```

  - Discuss any dynamic adjustments to the learning rate during the descent to enhance convergence.

```python
# Step 2: Polynomial Feature Transformation
def poly_features(X, degree):
    return np.hstack([X**i for i in range(degree + 1)])

degree = 9  # Optimal degree from MLE analysis
Phi = poly_features(X, degree)  # Polynomial features for training data
Phi_test = poly_features(X_test, degree)  # Polynomial features for test data

# Normalize Features
def normalize_features(Phi):
    mu = np.mean(Phi, axis=0)
    sigma = np.std(Phi, axis=0)
    sigma[sigma == 0] = 1
    Phi_normalized = (Phi - mu) / sigma
    Phi_normalized[:, 0] = 1  # Ensure the bias term remains unchanged
    return Phi_normalized, mu, sigma

Phi, mu, sigma = normalize_features(Phi)# Normalize training features
Phi_test = (Phi_test - mu) / sigma  # Normalize test features
Phi_test[:, 0] = 1  # Keep the bias term unchanged
```

```python
np.random.seed(42)
theta = np.random.randn(Phi.shape[1], 1)  # Initialize theta
alpha = 0.0002  # Learning rate
iterations = 100000  # Number of iterations
decay_rate = 0.00001
```

Decay rate is used to reduce the learning rate over time and ensure stable convergence, I got the lowest RMSE with 0.0002 learning rate with the decay rate of 0.00001.

2.3 Model Evaluation and Comparison

- **Task:** Monitor and record the RMSE on the test set at each step of the Gradient Descent. Compare this RMSE with that obtained ug the MLE method from Q1.

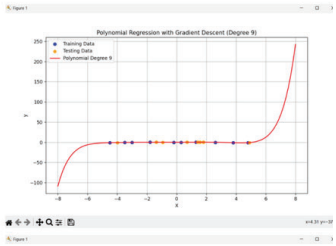
- **Requirements:**
  - Include complete Python code that carries out feature transformation, gradient descent optimization, RMSE calculation, and plotting.
  - Attach plots demonstrating cost reduction over iterations and the final model fit against the test data.
  - Discuss the implementation, convergence behavior, and effectiveness compared to MLE.

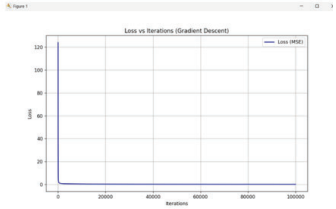
Bonus: Earn an additional 10 points, not included in the standard 100 points. You will receive these points if the RMSE of the test set, using the Theta parameters from Question 2, is lower than the RMSE obtained using the MLE method from Question 1.

Task 2.3 Model Evaluation and Comparison:

```python
# Step 3: Gradient Descent Implementation
def gradient_descent(X, y, theta, alpha, iteration, decay_rate):
    n = len(y)
    cost_history = [0] * iteration  # To store loss values at each iteration
    for i in range(iteration):
        prediction = X.dot(theta)
        error = y-prediction
        gradient = -2 * X.T.dot(error)
        alpha_t = alpha / (1 + decay_rate * i)
        theta -= alpha_t * gradient  # Update theta
        cost_history[i] = lossfunction(X, y, theta)

        # Early stopping condition for divergence or NaN
        if np.isnan(cost_history[i]) or (i > 0 and cost_history[i] > cost_history[i - 1]):
            print("Higher cost detected, breaking at iteration:", i)
            break

    return theta, cost_history
```

```python
# Run Gradient Descent
theta_gd, cost_history = gradient_descent(Phi, y, theta, alpha, iterations, decay_rate)

# Display Results
print("Optimized Theta (Gradient Descent):\n", theta_gd.flatten())

# Step 4: Predictions
y_train_pred = Phi @ theta_gd  # Training predictions
y_test_pred = PHi_test @ theta_gd  # Test predictions

# RMSE Calculation
def calculate_rmse(y_true, y_pred):
    return np.sqrt(np.mean((y_true - y_pred)**2))

final_train_rmse = calculate_rmse(y, y_train_pred)
final_test_rmse = calculate_rmse(y_test, y_test_pred)

# Display Final RMSE
print(f"Final RMSE (Training Data, Degree {degree}): {final_train_rmse:.4f}")
print(f"Final RMSE (Testing Data, Degree {degree}): {final_test_rmse:.4f}")

# Step 5: Plot Results
x_range = np.linspace(-8, 8, 100).reshape(-1, 1)
x_range_poly = poly_features(x_range, degree)
x_range_poly = (x_range_poly - mu) / sigma  # Normalize
x_range_poly[:, 0] = 1  # Keep bias term unchanged
y_pred_range = x_range_poly @ theta_gd
```






```
Optimized Theta (Gradient Descent):
 [-0.2441264   0.16307241  0.32179693  0.13057472 -2.06363488 -1.26602881
  1.05791751  0.33596328  0.23002671  0.6634029 ]
Final RMSE (Training Data, Degree 9): 0.1150
Final RMSE (Testing Data, Degree 9): 0.1605
```

Since the gradient descent iteration loop did not break, it means that the optimal convergence still has not occurred, hence the RMSE for test data here is larger than in MLE. The number of iterations must be increase or the learning rate must be increased to get to optimal convergence. Below is the snippet where I changed the initial parameters and got better results.