Henrique da Ponte

946930-5

ECE 133

Prof. Alizadeh

Homework 2

Problem 1 – Housing Prices

Code

```
import numpy as np
import pandas as pd
import cvxpy as cp

# ====================== Utility Functions ======================

def meanSquaredError(predicted, actual):
    '''
    Calculate the mean squared error (MSE) between predicted and actual values.

    Parameters:
    - predicted (numpy array): The array of predicted values generated by the model.
    - actual (numpy array): The array of actual or true values.

    Returns:
    - int: The mean squared error, rounded to the nearest whole number, between the
predicted and actual values.

    Notes:
    The mean squared error is computed as the average of the squared differences
between predicted and actual values.
    '''

    r = actual - predicted # Residuals
    mse = np.sum(r**2) / len(predicted)

    return round(mse)

def preprocessData(filename, trainDataPct):
    '''
    Preprocesses the data by loading it and dividing it into training and test sets.
```

```
    Parameters:
    - filename (str): Path to the input data file.
    - trainDataPct (float): Percentage of data to be used for training (e.g., 0.7 for
70%).

    Returns:
    - tuple:
        - X_train (numpy array): Training data attributes.
        - Y_train (numpy array): Training data target values.
        - X_test (numpy array): Test data attributes.
        - Y_test (numpy array): Test data target values.
    '''

    data = pd.read_csv(filename, delimiter='\t')

    # Using trainDataPct% of the data for training
    trainDataSize = int(trainDataPct * data.shape[0])
    trainData = data.iloc[:trainDataSize, :]
    X_train = trainData.iloc[:, :-1].values
    Y_train = (trainData.iloc[:, -1].values).reshape(-1, 1)

    # Using testDataPct% of the data for testing
    testData = data.iloc[trainDataSize:, :]
    X_test = testData.iloc[:, :-1].values
    Y_test = (testData.iloc[:, -1].values).reshape(-1, 1)

    return X_train, Y_train, X_test, Y_test

def trainLinearModel(X_train, Y_train):
    ''''
    Trains a linear regression model using CVX optimization.

    Parameters:
    - X_train (numpy array): Training data attributes.
    - Y_train (numpy array): Training data target values.

    Returns:
    - tuple:
        - Ytrain_pred (numpy array): Predicted values for the training data.
        - beta (numpy array): Learned coefficients for the attributes.
        - alpha (float): Learned intercept term.
    '''

    # Initilizing decision variables
    alpha = cp.Variable()
    beta = cp.Variable((X_train.shape[1], 1))
```

```
    # Defining function for our predictions
    Ytrain_pred = alpha + X_train @ beta

    # Defining objective function
    objective = cp.Minimize(cp.sum_squares(Y_train - Ytrain_pred))

    # Formulating problem
    problem = cp.Problem(objective)

    # solving the problem
    problem.solve()

    return Ytrain_pred.value, beta.value, alpha.value

def trainL1Model(X_train, Y_train):
    '''
    Trains a linear regression model using L1 (Lasso) regularization with CVX
optimization.

    Parameters:
    - X_train (numpy array): Training data attributes/features.
    - Y_train (numpy array): Training data target/output values.

    Returns:
    - tuple:
        - Ytrain_pred (numpy array): Predicted values for the training data using the
trained L1 model.
        - beta (numpy array): Learned coefficients for the attributes in the L1 model.
        - 0 (int): Placeholder for intercept term (always returns 0 since no intercept
is used in this model).

    Notes:
    - The L1 regularization tends to induce sparsity in the model, which means it
might produce a model where many feature weights are exactly zero.
    - The function uses CVX optimization to solve the regression problem with L1
penalty.
    - This version of L1 regressor does not incorporate an intercept term, hence the
return value of 0 for the intercept.
    '''

    # Initilizing decision variables
    beta = cp.Variable((X_train.shape[1], 1))

    # Defining function for our predictions
    Ytrain_pred = X_train @ beta

    # Defining objective function
    objective = cp.Minimize(cp.sum(cp.abs(Y_train - Ytrain_pred)))
```

```python
    # Formulating problem
    problem = cp.Problem(objective)

    # solving the problem
    problem.solve()

    return Ytrain_pred.value, beta.value, 0

def trainPolynomialModel(X_train, Y_train):
    '''
    Trains a polynomial regression model using CVX optimization.

    Parameters:
    - X_train (numpy array): Training data attributes.
    - Y_train (numpy array): Training data target values.

    Returns:
    - tuple:
        - Ytrain_pred (numpy array): Predicted values for the training data.
        - betas (numpy array): Learned coefficients for the attributes.
        - alpha (float): Learned intercept term.
    '''

    num_features = X_train.shape[1]

    # Generating polynomial terms for the input features
    X_poly = np.hstack([X_train, X_train**2, X_train**3])

    # Initializing decision variables
    alpha = cp.Variable()
    betas = cp.Variable((3 * num_features, 1))

    # Defining function for our predictions
    Ytrain_pred = alpha + X_poly @ betas

    # Defining objective function
    objective = cp.Minimize(cp.sum_squares(Y_train - Ytrain_pred))

    # Formulating problem
    problem = cp.Problem(objective)

    # Solving the problem
    problem.solve(solver=cp.SCS)

    return Ytrain_pred.value, betas.value, alpha.value

def deployModel(filename, split, trainMethod, modelName, poly=False):
```

```
    '''
    Trains a regression model on a given dataset and computes the mean squared error
for both training and test data.

    Parameters:
    - filename (str): Path to the input data file.
    - split (float): Percentage (expressed as a decimal, e.g., 0.3 for 30%) of data to
be used for training.
    - trainMethod (function): A function that trains the model. This function should
return predicted values for the training set, coefficients for the attributes, and an
intercept term.
    - modelName (str): A descriptive name for the model which will be used in print
statements.
    - poly (bool, optional): If True, the function assumes the model is polynomial and
will generate polynomial terms for the test data. Default is False.

    Returns:
    None

    Outputs:
    The function prints the mean squared error for the training data and test data.

    Notes:
    - This function assumes the input data file is tab-delimited and the target values
are in the last column.
    - The preprocessData function is used to split the data and should be defined
elsewhere in the code.
    - If 'poly' is set to True, the function will generate polynomial terms up to the
third degree for the test data.

    Example:
    deployModel('data.txt', 0.3, trainLinearModel, "Linear Regression")
    deployModel('data.txt', 0.3, trainPolynomialModel, "Polynomial Regression",
poly=True)
    '''

    X_train, Y_train, X_test, Y_test = preprocessData(filename, split) # Splitting
with split% of data for training

    Ytrain_pred, beta, alpha = trainMethod(X_train, Y_train) # Training with split%
training data on 'model' model

    if poly:
        X_test = np.hstack([X_test, X_test**2, X_test**3])
        Y_pred =  alpha + X_test @ beta # Testing 30% data of the data for training on
polynomial model

    else:
```

```
        Y_pred =  alpha + X_test @ beta # Testing split% of the data for training on 'model' model

    print(f'Mean squared error for training data on {modelName} model ({split * 100}% of data for training): ', meanSquaredError(Ytrain_pred, Y_train))
    print(f'Mean squared error for testing data on {modelName} model ({split* 100}% of data for training): ', meanSquaredError(Y_pred, Y_test))
    print('\n')

# ====================== Main ======================

deployModel('housing.txt', 0.3, trainLinearModel, 'Linear', poly=False)
deployModel('housing.txt', 0.6, trainLinearModel, 'Linear', poly=False)
deployModel('housing.txt', 0.3, trainPolynomialModel, 'Polynomial', poly=True)
deployModel('housing.txt', 0.6, trainPolynomialModel, 'Polynomial', poly=True)
deployModel('housing.txt', 0.3, trainL1Model, 'L1 Regressor', poly=False)
deployModel('housing.txt', 0.6, trainL1Model, 'L1 Regressor', poly=False)
```

Results

```
Mean squared error for training data on Linear model (30.0% of data for training):  5
Mean squared error for testing data on Linear model (30.0% of data for training):  413


Mean squared error for training data on Linear model (60.0% of data for training):  10
Mean squared error for testing data on Linear model (60.0% of data for training):  171
```

```
Mean squared error for training data on Polynomial model (30.0% of data for training):  2
Mean squared error for testing data on Polynomial model (30.0% of data for training):  740686561


Mean squared error for training data on Polynomial model (60.0% of data for training):  5
Mean squared error for testing data on Polynomial model (60.0% of data for training):  200020194


Mean squared error for training data on L1 Regressor model (30.0% of data for training):  6
Mean squared error for testing data on L1 Regressor model (30.0% of data for training):  286


Mean squared error for training data on L1 Regressor model (60.0% of data for training):  11
Mean squared error for testing data on L1 Regressor model (60.0% of data for training):  103
```

	Linear Regressor		Polynomial Regressor		L1 Regressor	
Train. Data %	30	60	30	60	30	60
Training MSE	5	10	2	5	6	11
Testing MSE	413	171	740686561	200020194	286	103

Problem 2 – Optimality Conditions

$$f(x,y) = \frac{1}{2}x^2 + xy - \frac{3}{2}y^2 + 2x + 5y + \frac{1}{3}y^3$$

First Order Optimality Condition:

$$\nabla f(x,y) = \begin{bmatrix} \frac{\partial f(x,y)}{\partial x} \\ \frac{\partial f(x,y)}{\partial y} \end{bmatrix} = \begin{bmatrix} x + y + 2 \\ x - 3y + 5 + y^2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$x + y + 2 = 0$$

$$1.\ \ x = -y - 2$$

$$2.\ x - 3y + 5 + y^2 = \mathbf{0}$$

Plugging equation 1 into equation 2 we get:

$$(-y - 2) - 3y + 5 + y^2 = \mathbf{0}$$
$$y^2 - \mathbf{4y} + \mathbf{3} = \mathbf{0}$$

$$\Delta = b^2 - 4ac$$
$$\Delta = (-4)^2 - 4 \cdot 1 \cdot 3 = 16 - 12$$
$$\Delta = 4$$

$$y_1 = \frac{-b + \sqrt{\Delta}}{2a} = \frac{-(-4) + \sqrt{4}}{2 \cdot 1} = \frac{4 + 2}{2} = \frac{6}{2}$$

$$y_1 = 3$$

$$y_2 = \frac{-b - \sqrt{\Delta}}{2a} = \frac{-(-4) - \sqrt{4}}{2 \cdot 1} = \frac{4 - 2}{2} = \frac{2}{2}$$

$$y_2 = 1$$

Now we plug the values of y we found into equation 1 to find the critical points:

$$x_1 = -y_1 - 2 = -3 - 2$$
$$x_1 = -5$$

$$x_2 = -y_2 - 2 = -1 - 2$$
$$x_1 = -3$$
$$\therefore$$
$$Critical\ Points\ (x, y):$$
$$c_1 = (-5,3), \qquad c_2 = (-3,1)$$

Second Order Optimality Condition:

$$\mathrm{H} = \nabla^2 f(x,y) = \begin{bmatrix} \frac{\partial^2 f(x,y)}{\partial x^2} & \frac{\partial^2 f(x,y)}{\partial x \partial y} \\ \frac{\partial^2 f(x,y)}{\partial y \partial x} & \frac{\partial^2 f(x,y)}{\partial y^2} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 2y-3 \end{bmatrix}$$

$$det(H) = 2y - 3 - 1$$

For $c_1 = (-5,3)$:

$$det(H) = 2 \cdot 3 - 3 - 1 = 2 = P.D. = Local\ minima$$

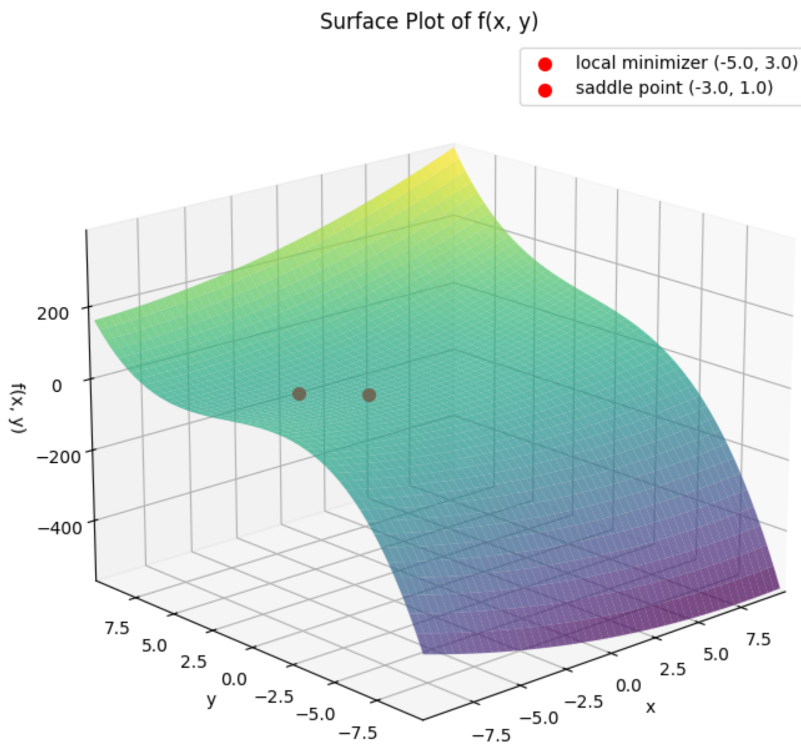
For $c_2 = (-3,1)$:

$$det(H) = 2 \cdot 1 - 3 - 1 = -2 = N.D. = Saddle\ Point$$

Code

```
# ====================== Problem 2 ======================

import matplotlib.pyplot as plt

def f_func_problem_2(x, y):
    ''''
    Function provided for Problem 2
    '''
    return 0.5*x**2 + x*y - 1.5*y**2 + 2*x + 5*y + (1/3)*y**3

critical_points_problem_2 = [(-5.0, 3.0), (-3.0, 1.0)]
classification_problem_2 = {(-5.0, 3.0): 'local minimizer', (-3.0, 1.0): 'saddle 
point'}


# Generate x and y values
x_values = np.linspace(-10, 10, 400)
y_values = np.linspace(-10, 10, 400)
x_mesh, y_mesh = np.meshgrid(x_values, y_values)

# Evaluate the function over the grid
f_values_2 = f_func_problem_2(x_mesh, y_mesh)

# Plot the surface
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(x_mesh, y_mesh, f_values_2, cmap='viridis', alpha=0.7)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('f(x, y)')
ax.set_title('Surface Plot of f(x, y)')
ax.view_init(elev=25, azim=-60)

# Mark the critical points on the plot
for point, label in classification_problem_2.items():
    ax.scatter(*point, f_func_problem_2(*point), color='r', s=50, label=f"{label}
{point}")

# Show the plot
plt.legend()
plt.show()
```

Results



Surface Plot of f(x, y)
local minimizer (-5.0, 3.0)
saddle point (-3.0, 1.0)
200
0
−200
−400
f(x, y)
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
y
x

Problem 3 – Optimality Conditions

$$f(x_1, x_2) = {x_1}^2 x_2 - 2x_1{x_2}^2 + 4x_1x_2 - 8$$

First Order Optimality Condition:

$$\nabla f(x_1, x_2) = \begin{bmatrix} \dfrac{\partial f(x_1, x_2)}{\partial x_1} \\ \dfrac{\partial f(x_1, x_2)}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2x_1x_2 - 2{x_2}^2 + 4x_2 \\ {x_1}^2 - 4x_1x_2 + 4x_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$2x_1x_2 - 2{x_2}^2 + 4x_2$$
$$x_2(\,2x_1 - 2x_2 + 4\,)$$
$$\therefore$$
$$x_2 = 0 \text{ or}$$
$$x_2 = x_1 + 2$$

Now if we substitute our x_2 values into the first equation we get:

For $x_2 = 0$:

$$\begin{gathered} {x_1}^2 - 4x_1 \cdot 0 + 4x_1 = 0 \\ {x_1}^2 + 4x_1 = 0 \\ x_1(x_1 + 4) = 0 \end{gathered}$$

$$\begin{gathered} x_1 = 0 \\ x_1 = -4 \end{gathered}$$

For $x_2 = x_1 + 2$:

$$\begin{gathered} {x_1}^2 - 4x_1 \cdot (x_1 + 2) + 4x_1 = 0 \\ -3{x_1}^2 - 4x_1 = 0 \\ x_1(-3x_1 - 4) = 0 \end{gathered}$$

$$\begin{gathered} x_1 = 0 \rightarrow x_2 = 2 \\ x_1 = -\frac{4}{3} \rightarrow x_2 = \frac{2}{3} \end{gathered}$$

Therefore, our critical points ($c_i = (x_1, x_2)$) are:

$$c_1 = (0, 0)$$
$$c_2 = (-4, 0)$$
$$c_3 = (0, 2)$$
$$c_4 = (-\frac{4}{3}, \frac{2}{3})$$

Second Order Optimality Condition:

$$\mathrm{H} = \nabla^2 f(x,y) = \begin{bmatrix} \frac{\partial^2 f(x,y)}{\partial x^2} & \frac{\partial^2 f(x,y)}{\partial x \partial y} \\ \frac{\partial^2 f(x,y)}{\partial y \partial x} & \frac{\partial^2 f(x,y)}{\partial y^2} \end{bmatrix} = \begin{bmatrix} 2x_2 & 2x_1 - 4x_2 + 4 \\ 2x_1 - 4x_2 + 4 & -4x_1 \end{bmatrix}$$

$$det(H) = -(2x_1 - 4x_2 + 4)^2 - 8x_1x_2$$

For $c_1 = (0, 0)$:

$$det(H) = -(2x_1 - 4x_2 + 4)^2 - 8x_1x_2 = det(H) = -(2 \cdot 0 - 4 \cdot 0 + 4)^2 - 8 \cdot 0 \cdot 0 = -16$$
$$Saddle\ Point$$

For $c_2 = (-4, 0)$:

$$det(H) = -(2x_1 - 4x_2 + 4)^2 - 8x_1x_2 = -(2 \cdot -4 - 4 \cdot 0 + 4)^2 - 8 \cdot -4 \cdot 0 = -16$$
$$Saddle\ Point$$

For $c_3 = (0, 2)$:

$$det(H) = -(2x_1 - 4x_2 + 4)^2 - 8x_1x_2 = -(2 \cdot 0 - 4 \cdot 2 + 4)^2 - 8 \cdot 0 \cdot 2 = -16$$
$$Saddle\ Point$$

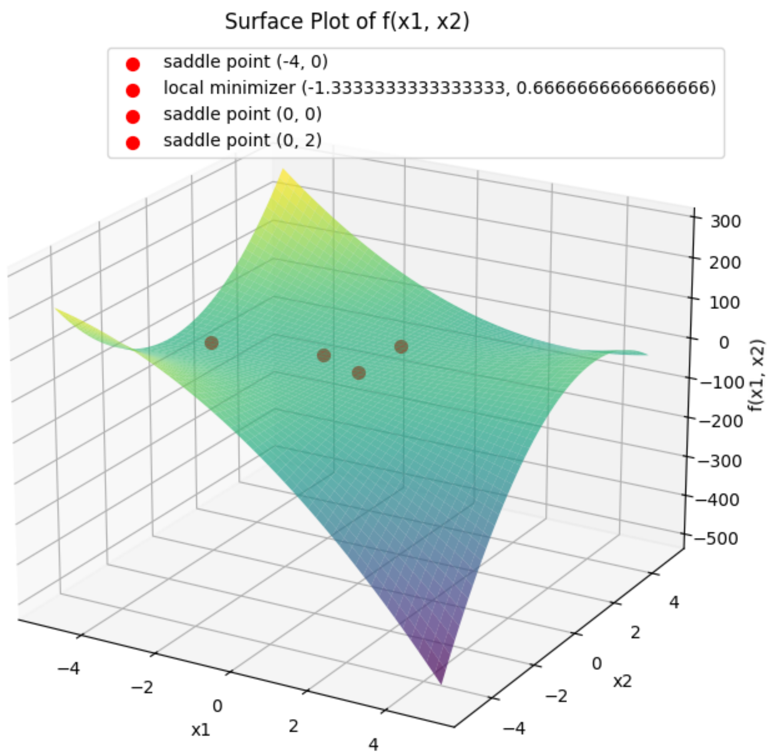
For $c_4 = (-\frac{4}{3}, \frac{2}{3})$:

$$det(H) = -(2x_1 - 4x_2 + 4)^2 - 8x_1x_2 = -\left(2 \cdot -\frac{4}{3} - 4 \cdot \frac{2}{3} + 4\right)^2 - 8 \cdot -\frac{4}{3} \cdot \frac{2}{3}$$
$$= \frac{432}{81} > 0$$
$$Local\ Minima$$

Code

```
# ===================== Problem 3 =====================

critical_points_problem_3 = [(-4, 0), (-4/3, 2/3), (0, 0), (0, 2)]
classification_problem_3 = {(-4, 0): 'saddle point', (-4/3, 2/3): 'local minimizer',
(0, 0): 'saddle point', (0, 2): 'saddle point'}

# Generate x1 and x2 values
x1_values = np.linspace(-5, 5, 400)
x2_values = np.linspace(-5, 5, 400)
x1_mesh, x2_mesh = np.meshgrid(x1_values, x2_values)

# Evaluate the function over the grid
f_values = f_func_problem_3(x1_mesh, x2_mesh)

# Plot the surface
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(x1_mesh, x2_mesh, f_values, cmap='viridis', alpha=0.7)
ax.set_xlabel('x1')
ax.set_ylabel('x2')
ax.set_zlabel('f(x1, x2)')
ax.set_title('Surface Plot of f(x1, x2)')
ax.view_init(elev=25, azim=-60)

# Mark the critical points on the plot
for point, label in classification_problem_3.items():
    ax.scatter(*point, f_func_problem_3(*point), color='r', s=50, label=f"{label}
{point}")

# Show the plot
plt.legend()
plt.show()
```

Results



Surface Plot of f(x1, x2)
saddle point (-4, 0)
local minimizer (-1.3333333333333333, 0.6666666666666666)
saddle point (0, 0)
saddle point (0, 2)
300
200
100
0
−100
−200
−300
−400
−500
f(x1, x2)
−4
−2
0
2
4
x1
−4
−2
0
2
4
x2