**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)))

# 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**

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 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**

First Order Optimality Condition:

2.

Plugging equation 1 into equation 2 we get:

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

Second Order Optimality Condition:

For

For

**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**

**A graph of a surface plot

Description automatically generated**

**Problem 3 – Optimality Conditions**

First Order Optimality Condition:

or

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

For :

For :

Therefore, our critical points () are:

Second Order Optimality Condition:

For :

For :

For

For :

**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**

A graph of a surface plot

Description automatically generated