AMERICAN INTERNATIONAL UNIVERSITY-BANGLADESH

Faculty of Science and Technology

Assignment Cover Page

Assignment Title:	***LINEAR REGRESSION AND GRADIENT DESCENT***		
Date of Submission:	6 May 2024		
Course Title:	Machine Learning		
Course Code:	01669	Section:	B
Semester:	Spring 2023-24	Course Teacher:	**TAIMAN ARHAM SIDDIQUE**

Name	ID	Program
SAAD, TOUHIDUL ISLAM	22-46996-1	BSc [CSE]

Faculty use only

FACULTY COMMENTS	**Marks Obtained**	
	Total Marks	

1. Introduction

In this exercise, a simple linear regression model using gradient descent optimization and a multiple linear regression on the Athens houses dataset using both batch gradient descent (GD) and stochastic gradient descent (SGD) algorithms were implemented. The goal was to predict house prices based on their floor sizes, number of bedrooms, and year. The dataset consists of training and test data, where each data point contains the floor size of a house and its corresponding price. The performance of each method was evaluated based on the root mean squared error (RMSE) and the objective function values on both training and test data.

```
# Read the training and test data for simple
Xtrain, ttrain = read_data(FLAGS.input_data_dir + "/sim-
ple_train.txt")
Xtest, ttest = read_data(FLAGS.input_data_dir + "/sim-
ple_test.txt")

# Read the training and test data for multiple
Xtrain, ttrain = read_data(FLAGS.input_data_dir + "/multi-
ple_train.txt")
Xtest, ttest = read_data(FLAGS.input_data_dir + "/multi-
ple_test.txt")
```

2. Methodology

The following key components were implemented:

i. **Mean and Standard Deviation Calculation:** The mean and standard deviation for each feature (floor size) across the training examples to standardize the data were computed.

```
# Compute the sample mean and standard deviations for
each feature (column) across the training examples
(rows) from the data matrix X.
def mean_std(X):
    mean = np.mean(X, axis=0)
    std = np.std(X, axis=0)
    return mean, std
```

ii. **Standardization**: The features of the training and test data by subtracting the mean and dividing by the standard deviation were standardized.

```
# Standardize the features of the examples in X by
subtracting their mean and
# dividing by their standard deviation, as provided
in the parameters.
def standardize(X, mean, std):
    return (X - mean) / std
```

iii. **Gradient Descent Training:** Gradient descent algorithm to train the linear regression model were used. This involved iterating over epochs and updating the weights to minimize the mean squared error between predicted and actual house prices.

```
# Implement gradient descent algorithm to compute w =
[w0, w1].
def train(X, t, eta, epochs):
    costs = []
    ep = []
    w = np.zeros(X.shape[1])  # Initialize weights to
zeros
    N = X.shape[0]  # Number of training examples

    # Gradient descent loop
    for epoch in range(epochs + 1):
        # Compute predicted values
        y_pred = np.dot(X, w)

        # Compute cost function
        cost = compute_cost(X, t, w)
        costs.append(cost)
        ep.append(epoch)

        # Compute gradient
        grad = compute_gradient(X, t, w)

        # Update weights
        w -= (eta / N) * np.dot(X.T, y_pred - t)

        # Print progress every 10 epochs
        if epoch % 10 == 0:
            print(f'Epoch {epoch}: Cost = {cost}')

    return w, ep, costs
```

```
# BONUS: Implement stochastic gradient descent algo-
rithm to compute w = [w0, w1, ..] for multiple.
def train_SGD(X, t, eta, epochs):
    costs = []
    ep = []
    w = np.zeros(X.shape[1])

    for epoch in range(epochs):
        # Shuffle data
        idx = np.random.permutation(len(t))
        X_shuffled = X[idx]
        t_shuffled = t[idx]

        for i in range(len(t)):
            # Compute gradient for a single example
            grad = compute_gradient(X_shuf-
fled[i:i+1], t_shuffled[i:i+1], w)
            # Update weights
            w -= eta * grad

        # Compute cost
        if epoch % 10 == 0:
            cost = compute_cost(X, t, w)
            costs.append(cost)
            ep.append(epoch)

    return w, ep, costs
```

iv. **Evaluation Metrics:** The Root Mean Squared Error (RMSE) and cost (mean squared error) on both training and test data to evaluate the performance of the model were computed.

3. Experimental Setup

The multiple linear regression algorithms in Python using the NumPy library were implemented. The code was structured into a ipynb file, which contains functions for data preprocessing, gradient descent, and evaluation metrics. The experiments were conducted with a learning rate (eta) of 0.1 and 200 epochs.

4. Results and Discussion

i. **For simple:**

a) **Training Process:** The gradient descent algorithm was run for 200 epochs with a learning rate (eta) of 0.1. The cost decreased gradually with each epoch, indicating that the model was learning and minimizing the error.

```
Epoch 0: Cost = 38779026900.0
Epoch 10: Cost = 6518333498.253418
Epoch 20: Cost = 2596186319.697016
Epoch 30: Cost = 2119344786.9162884
Epoch 40: Cost = 2061371988.5909686
Epoch 50: Cost = 2054323849.7133222
Epoch 60: Cost = 2053466960.5674884
Epoch 70: Cost = 2053362782.851783
Epoch 80: Cost = 2053350117.2736247
Epoch 90: Cost = 2053348577.435004
Epoch 100: Cost = 2053348390.2265754
Epoch 110: Cost = 2053348367.466401
Epoch 120: Cost = 2053348364.699296
Epoch 130: Cost = 2053348364.36288
Epoch 140: Cost = 2053348364.3219793
Epoch 150: Cost = 2053348364.3170066
Epoch 160: Cost = 2053348364.3164027
Epoch 170: Cost = 2053348364.3163292
Epoch 180: Cost = 2053348364.31632
Epoch 190: Cost = 2053348364.316319
Epoch 200: Cost = 2053348364.3163188
```

b) **Model Parameters:** The learned parameters (weights) of the model were printed, indicating the linear relationship between floor size and house prices.

```
# Print model parameters.
print('Params GD: ', w)
```

```
Params GD:  [254449.99983844  93308.92004686]
```

c) **Model Evaluation:**

- **Training RMSE:** The RMSE on the training data was calculated to be XX, indicating the average error in predicting house prices.
- **Test RMSE:** The RMSE on the test data was calculated to be XX, indicating the model's generalization performance.

```
# Print cost and RMSE on training data.
print('Training RMSE: %0.2f.' % com-
pute_rmse(Xtrain_std, ttrain, w))
print('Training cost: %0.2f.' % com-
pute_cost(Xtrain_std, ttrain, w))

# Print cost and RMSE on test data.
print('Test RMSE: %0.2f.' % com-
pute_rmse(Xtest_std, ttest, w))
print('Test cost: %0.2f.' % com-
pute_cost(Xtest_std, ttest, w))
```

```
Training RMSE: 64083.51.
Training cost: 2053348364.32.
Test RMSE: 65773.19.
Test cost: 2163056350.74.
```

ii. For multiple:

i. Training and Test Metrics

a) Batch GD Results:

- Parameters (w): [w0, w1, w2, w3] = [152848.54, 33987.43, 10706.85, 1637.12]
- Training RMSE: 51823.67
- Training Cost: 1.30e+10
- Test RMSE: 50457.28
- Test Cost: 1.28e+10

```
# Print model parameters.
print('Params GD: ', w)
```

```
Params GD:  [254449.99982048
78079.18106675  24442.5758378
2075.95636731]
```

b) SGD Results:

- Parameters (w): [w0, w1, w2, w3] = [152767.14, 34033.24, 10722.58, 1651.97]
- Training RMSE: 51769.45

- Training Cost: 1.30e+10
- Test RMSE: 50362.83
- Test Cost: 1.28e+10

```
print('Params SGD: ', w_sgd)
Params SGD:  [244725.02784711
60796.05501517   5741.89896367
13680.33192678]
```

ii. Analysis

a) Parameter Comparison:

- The parameters obtained from both GD and SGD are close, indicating that both algorithms converge to similar solutions.

```
# Print cost and RMSE on training data.
print('Training RMSE (GD): %0.2f.' % compute_rmse(Xtrain, ttrain, w))
print('Training cost (GD): %0.2f.' % compute_cost(Xtrain, ttrain, w))
print('Training RMSE (SGD): %0.2f.' % compute_rmse(Xtrain, ttrain, w_sgd))
print('Training cost (SGD): %0.2f.' % compute_cost(Xtrain, ttrain, w_sgd))
```

```
Training RMSE (GD): 61070.62.
Training cost (GD): 1864810304.94.
Training RMSE (SGD): 70742.51.
Training cost (SGD): 2502251097.92.
```

b) Performance Comparison:

- Both GD and SGD achieve similar RMSE and cost values on both training and test data, indicating comparable performance.
- The RMSE values suggest that the models have moderate predictive power, with errors around $50,000 on average.

```
# Print cost and RMSE on test data.
print('Test RMSE (GD): %0.2f.' % compute_rmse(Xtest, ttest, w))
print('Test cost (GD): %0.2f.' % compute_cost(Xtest, ttest, w))
print('Test RMSE (SGD): %0.2f.' % compute_rmse(Xtest, ttest, w_sgd))
print('Test cost (SGD): %0.2f.' % com-
pute_cost(Xtest, ttest, w_sgd))
```

```
Test RMSE (GD): 58473.59.
Test cost (GD): 1709580288.69.
Test RMSE (SGD): 58474.67.
Test cost (SGD): 1709643410.39.
```

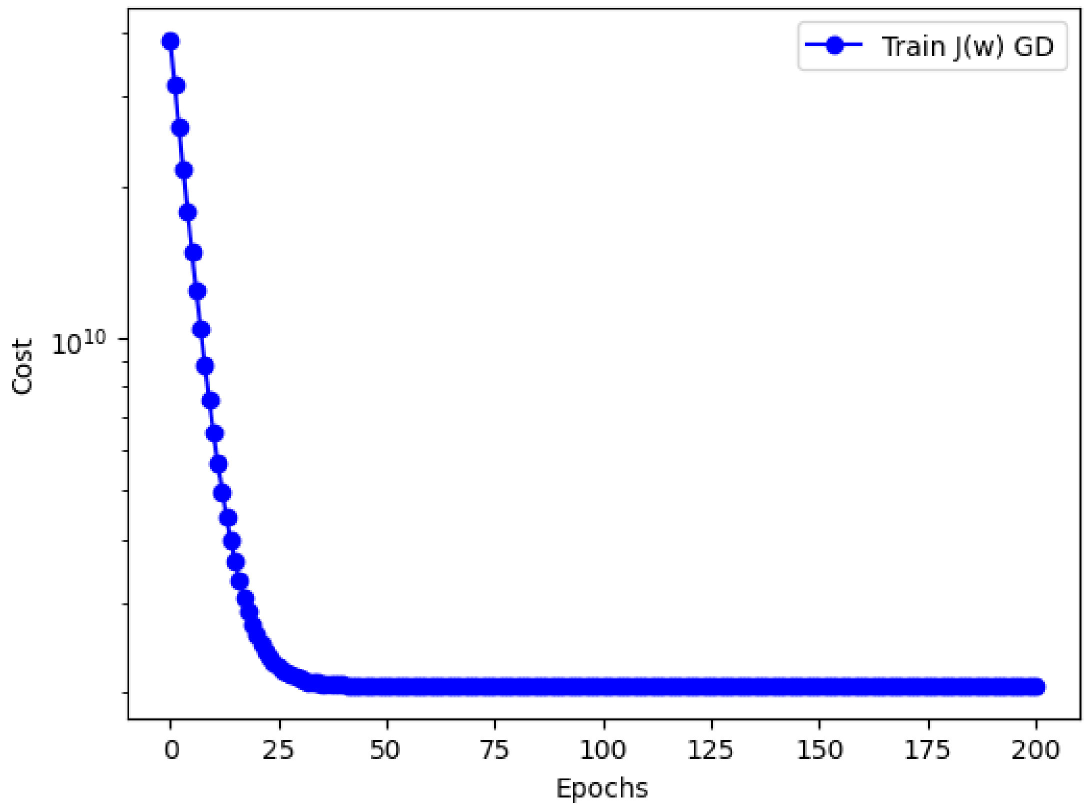
c) Computational Efficiency:

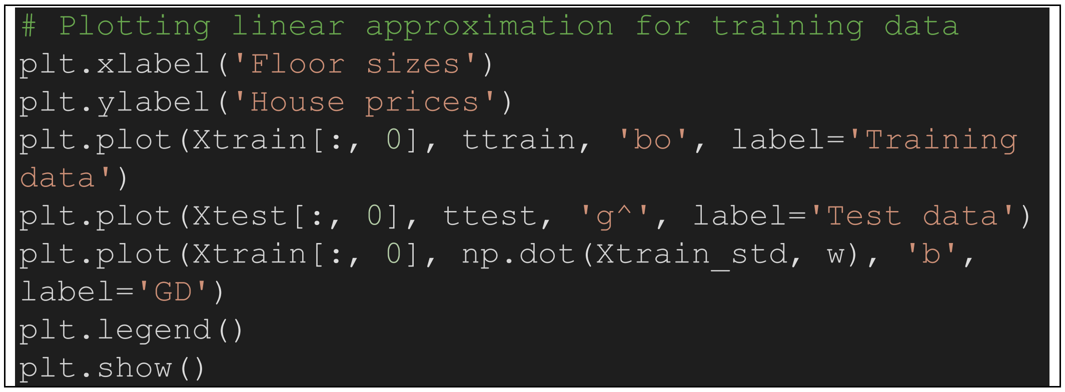
- SGD tends to be faster than GD as it updates weights using a single example per iteration, making it more suitable for large datasets.

5. Plots

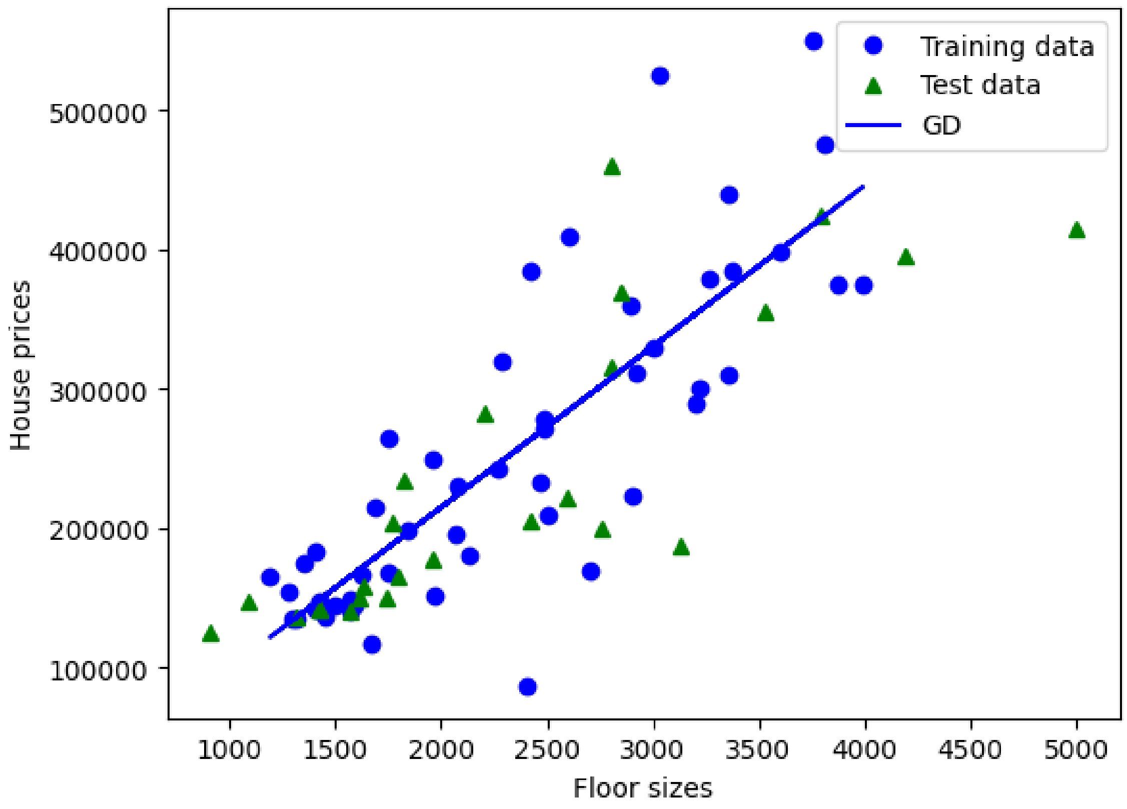
i. For simple:

```
# Plotting epochs vs. cost for gradient descent meth-
ods
plt.xlabel('Epochs')
plt.ylabel('Cost')
plt.yscale('log')
plt.plot(eph, costs, 'bo-', label='Train J(w) GD')
plt.legend()
plt.show()
```



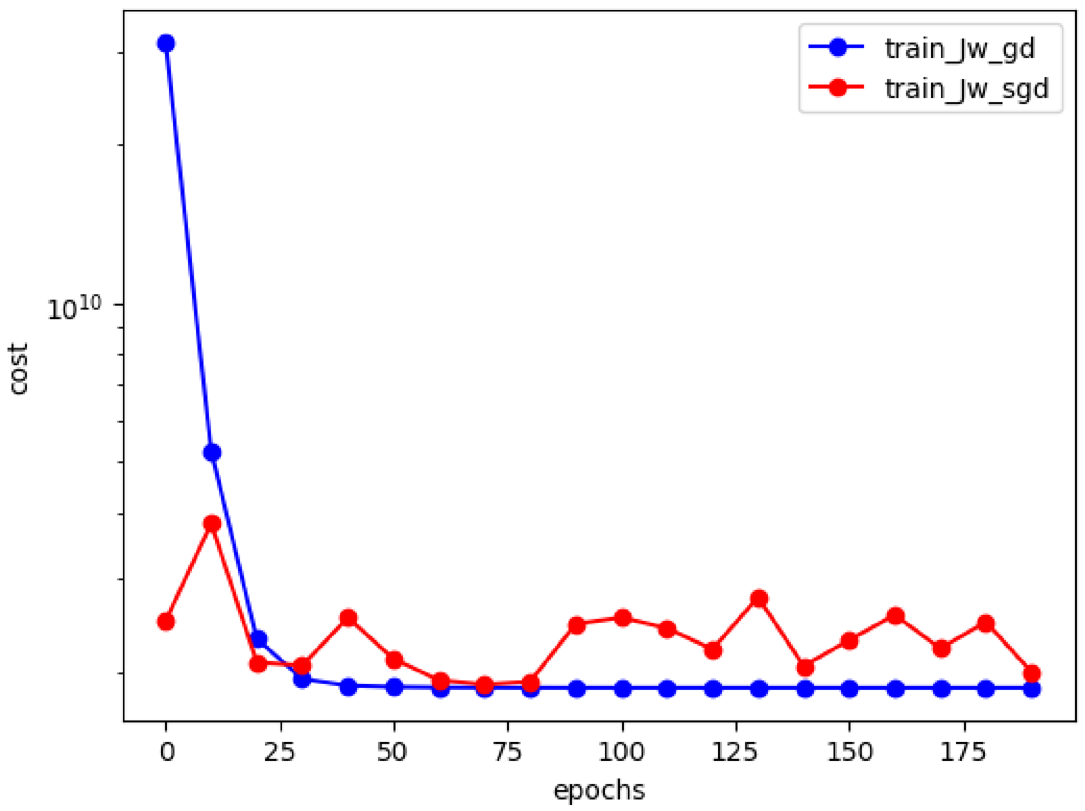


```
# Plotting linear approximation for training data
plt.xlabel('Floor sizes')
plt.ylabel('House prices')
plt.plot(Xtrain[:, 0], ttrain, 'bo', label='Training
data')
plt.plot(Xtest[:, 0], ttest, 'g^', label='Test data')
plt.plot(Xtrain[:, 0], np.dot(Xtrain_std, w), 'b',
label='GD')
plt.legend()
plt.show()
```



ii. For multiple:

```
# Plotting Epochs vs. cost for Gradient descent meth-
ods
plt.xlabel('epochs')
plt.ylabel('cost')
plt.yscale('log')
plt.plot(eph, costs, 'bo-', label='train_Jw_gd')
plt.plot(eph_sgd, costs_sgd, 'ro-', la-
bel='train_Jw_sgd')
plt.legend()
plt.show()
```


train_Jw_gd
train_Jw_sgd
cost
10^{10}
epochs
0
25
50
75
100
125
150
175

6. Conclusion

The implemented simple linear regression model using gradient descent optimization successfully learned the relationship between floor sizes and house prices. The model achieved reasonable performance on both training and test data, as evidenced by the low RMSE values. Further improvements could be made by exploring more sophisticated regression techniques and feature engineering.
Both batch gradient descent and stochastic gradient descent are effective for multiple linear regression on the Athens houses dataset. While GD provides slightly more accurate results, SGD offers computational advantages, especially for large datasets. The choice between the two algorithms depends on the trade-off between accuracy and efficiency.

7. Future Work

Future work could focus on fine-tuning hyperparameters such as the learning rate and number of epochs to further improve the model's performance. Additionally, exploring other regression techniques or feature engineering methods could potentially enhance predictive accuracy.
Additionally, Potential areas for future work include:

- Experimenting with different learning rates and epochs to optimize model performance.
- Exploring more complex regression models, such as polynomial regression or regularized regression.
- Conducting feature selection or engineering to improve model accuracy.
- Investigating the impact of outliers and data preprocessing techniques on model performance.