Mohan Srinivas Kanarapu

811434915

mk84800@uga.edu

02/10/2025

Machine Learning Homework - 1

1. Comparison of Closed-Form Solution and Gradient Descent

Closed-Form Solution:

- The closed-form solution directly calculates the optimal parameters 'w' using the formula: $w = (X^T X)^{-1} X^T y$
- It's efficient for small datasets because it provides an exact solution in one step. As seen in the plot, the predicted prices closely align with the actual prices, indicating a good fit.

Gradient Descent Solution:

- Gradient descent iteratively updates Θ using the gradient of the cost function: $\Theta = \Theta - \alpha \nabla J(\Theta)$
- where α is the learning rate. It requires tuning hyperparameters (learning rate, number of iterations) and may take multiple iterations to converge. The plot shows similar alignment between actual and predicted prices, though the method's effectiveness depends on appropriate tuning.

2. Differences in Parameters

Parameter Values:

- The closed-form solution produces a unique set of parameters because it solves the equation exactly.
- Gradient descent approximates the parameters over iterations and may yield slightly different values depending on the number of iterations and the learning rate.

Interpretation:

- If the gradient descent method converges properly, the parameter values should be close to those obtained via the closed-form solution.
- Minor discrepancies could arise due to numerical precision or insufficient convergence.

3. Computational Efficiency and Preference Scenarios

Time Complexity:

- **Closed-Form Solution**:
  The time complexity is $O(n^3)$ due to the matrix inversion step $(X^T X)^{-1}$. It is efficient for small to moderately sized datasets but becomes impractical for large datasets.
- **Gradient Descent**:
  The time complexity is $O(n \cdot m \cdot k)$, where n is the number of features, m is the number of data points, and k is the number of iterations. It is scalable for large datasets, but the convergence rate depends on hyperparameters.

Situational Preference:

- Use the **closed-form solution** for small datasets where computational resources allow for matrix inversion, opt for **gradient descent** for large datasets, datasets with many features, or when distributed computation is required.

Screenshots of Output plots:

Gradient Descent Solution:

Code + Text

```
# import libraries
import numpy as np
import pandas as pd
from sklearn import datasets
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score
```

```
# Load the Boston Housing dataset

# boston = datasets.load_boston()
boston = fetch_openml(name="boston", version=1, as_frame=True)
X = boston.data
y = boston.target

# Convert to DataFrame
df = pd.DataFrame(X, columns=boston.feature_names)
df['MEDV'] = y  # 'MEDV' is the median value of owner-occupied homes in $1000s

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(df.drop('MEDV', axis=1), df['MEDV'], test_size=0.2, random_state=42)

# Display the first few rows of the dataset
df.head()
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	18.7	396.90	5.33	36.2

```
# Normalize the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Convert to numpy arrays
X_train_scaled = np.c_[np.ones(X_train_scaled.shape[0]), X_train_scaled]  # Add bias term
X_test_scaled = np.c_[np.ones(X_test_scaled.shape[0]), X_test_scaled]
y_train = y_train.values
```

Code + Text

```python
y_train = y_train.values
y_test = y_test.values
```

```python
# Initialize parameters
def initialize_parameters(n_features):
    return np.zeros(n_features)
```

```python
# Cost function (Mean Squared Error)
def compute_cost(X, y, theta):
    m = len(y)
    predictions = X.dot(theta)
    cost = (1 / (2 * m)) * np.sum((predictions - y) ** 2)
    return cost
```

```python
# Gradient descent
def gradient_descent(X, y, theta, alpha, num_iters):
    m = len(y)
    cost_history = []

    for i in range(num_iters):
        # Calculate gradient
        gradients = (1 / m) * X.T.dot(X.dot(theta) - y)
        # Update theta
        theta -= alpha * gradients
        # Compute and save the cost
        cost = compute_cost(X, y, theta)
        cost_history.append(cost)

        # Print cost every 100 iterations
        if i % 100 == 0:
            print(f"Iteration {i}, Cost: {cost:.4f}")

    return theta, cost_history
```

```python
# Initialize
theta = initialize_parameters(X_train_scaled.shape[1])

# Hyperparameters
alpha = 0.01  # Learning rate
num_iters = 1000  # Number of iterations

# Train the model
theta, cost_history = gradient_descent(X_train_scaled, y_train, theta, alpha, num_iters)
```

```
Iteration 0, Cost: 295.9003
Iteration 100, Cost: 47.3601
Iteration 200, Cost: 16.2071
```


Code + Text

```python
y_train = y_train.values
y_test = y_test.values
```

```python
# Initialize parameters
def initialize_parameters(n_features):
    return np.zeros(n_features)
```

```python
# Cost function (Mean Squared Error)
def compute_cost(X, y, theta):
    m = len(y)
    predictions = X.dot(theta)
    cost = (1 / (2 * m)) * np.sum((predictions - y) ** 2)
    return cost
```

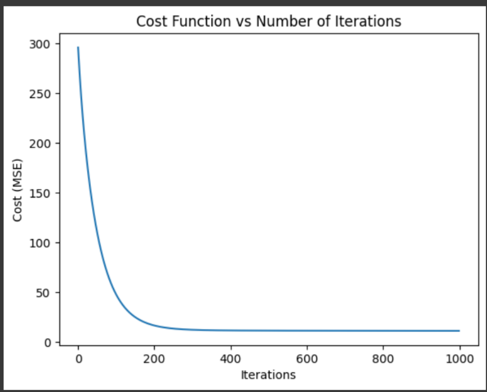
```python
# Gradient descent
def gradient_descent(X, y, theta, alpha, num_iters):
    m = len(y)
    cost_history = []

    for i in range(num_iters):
        # Calculate gradient
        gradients = (1 / m) * X.T.dot(X.dot(theta) - y)
        # Update theta
        theta -= alpha * gradients
        # Compute and save the cost
        cost = compute_cost(X, y, theta)
        cost_history.append(cost)

        # Print cost every 100 iterations
        if i % 100 == 0:
            print(f"Iteration {i}, Cost: {cost:.4f}")

    return theta, cost_history
```

```python
# Initialize
theta = initialize_parameters(X_train_scaled.shape[1])

# Hyperparameters
alpha = 0.01  # Learning rate
num_iters = 1000  # Number of iterations

# Train the model
theta, cost_history = gradient_descent(X_train_scaled, y_train, theta, alpha, num_iters)
```

```
Iteration 0, Cost: 295.9003
Iteration 100, Cost: 47.3601
Iteration 200, Cost: 16.2071
```

Code + Text

```python
# Plot the cost function vs number of iterations
plt.plot(range(num_iters), cost_history)
plt.xlabel("Iterations")
plt.ylabel("Cost (MSE)")
plt.title("Cost Function vs Number of Iterations")
plt.show()
```




```python
# Evaluate the model
y_pred = X_test_scaled.dot(theta)
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
```

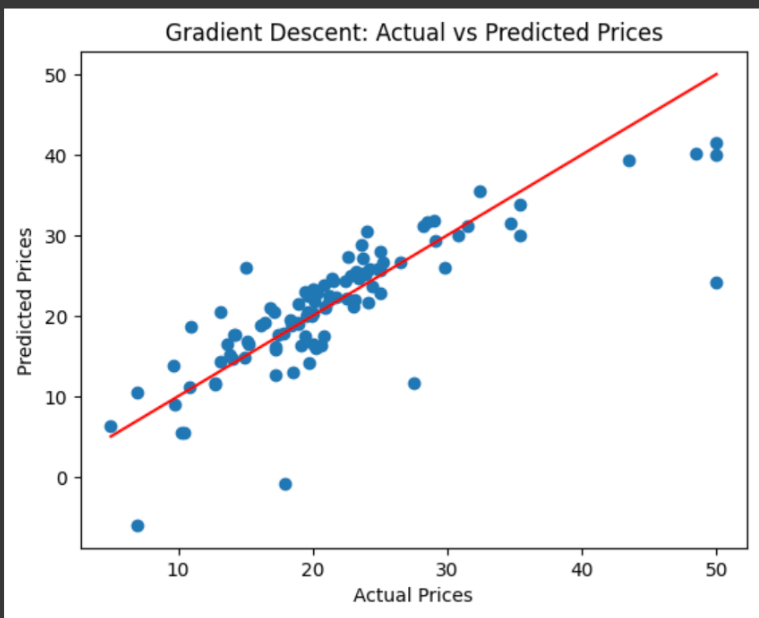
```python
print(f"Mean Squared Error (MSE): {mse:.4f}")
print(f"R-squared Score (R²): {r2:.4f}")
```

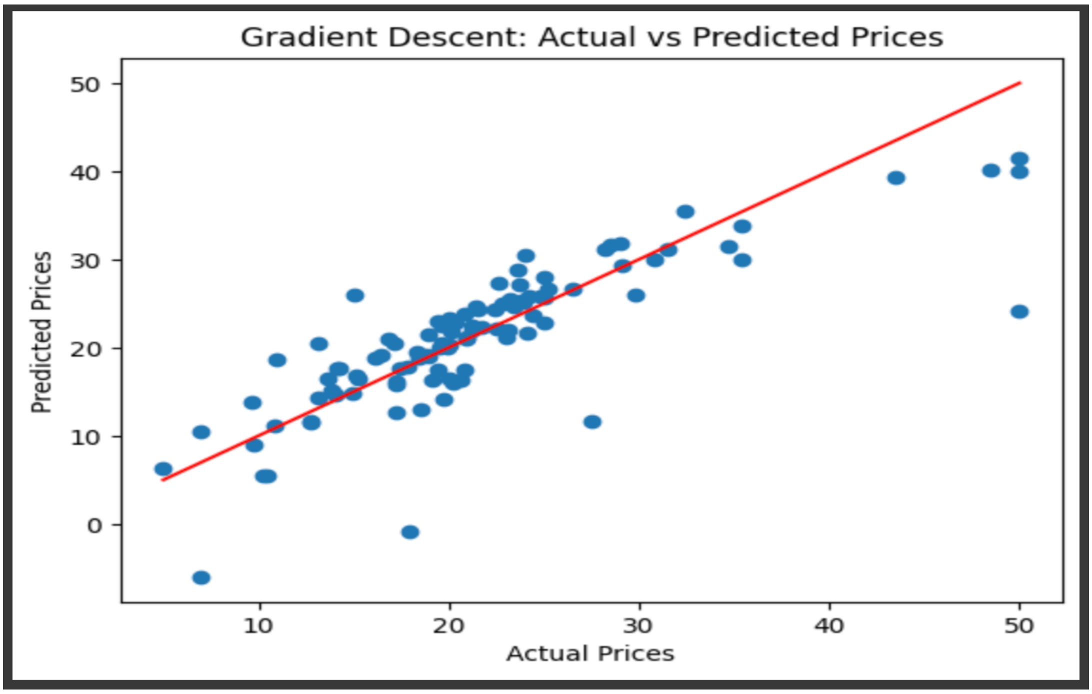
```
Mean Squared Error (MSE): 25.3497
R-squared Score (R²): 0.6543
```

```python
# Plot Actual vs Predicted prices
plt.scatter(y_test, y_pred)
plt.xlabel("Actual Prices")
plt.ylabel("Predicted Prices")
```

Code + Text

```python
# Plot Actual vs Predicted prices
plt.scatter(y_test, y_pred)
plt.xlabel("Actual Prices")
plt.ylabel("Predicted Prices")
plt.title("Gradient Descent: Actual vs Predicted Prices")
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], color='red')
plt.show()
```





Closed Form Solution:

Code + Text

+ Code + Text

```python
# Import libraries
import numpy as np
import pandas as pd
from sklearn import datasets
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
```

```python
# Load the Boston Housing dataset

# boston = datasets.load_boston()
boston = fetch_openml(name="boston", version=1, as_frame=True)
X = boston.data
y = boston.target

# Convert to DataFrame
df = pd.DataFrame(X, columns=boston.feature_names)
df['MEDV'] = y  # 'MEDV' is the median value of owner-occupied homes in $1000s

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(df.drop('MEDV', axis=1), df['MEDV'], test_size=0.2, random_state=42)

# Display the first few rows of the dataset
df.head()
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	18.7	396.90	5.33	36.2

```python
# Add a bias term (column of 1s) for closed-form solution
X_train_b = np.c_[np.ones((X_train.shape[0], 1)), X_train]
X_test_b = np.c_[np.ones((X_test.shape[0], 1)), X_test]

# Convert X_train and y_train to NumPy arrays with dtype np.float64
X_train_np = np.array(X_train_b, dtype=np.float64)
y_train_np = np.array(y_train, dtype=np.float64).reshape(-1, 1)
```

```python
# Convert X_test to float for prediction
X_test_np = np.array(X_test_b, dtype=np.float64)
```

```python
def closed_form_solution(X, y):
    w = np.linalg.inv(X.T @ X) @ X.T @ y
    return w;
```

```python
def predict(X, w):
    return X @ w
```

```python
# Closed-form solution: w = (X^T X)^-1 X^T y
w = closed_form_solution(X_train_np, y_train_np)

y_pred = X_test_np @ w
```

```python
# Calculate MSE
mse = mean_squared_error(y_test, y_pred)
print(f"Closed-Form Solution MSE: {mse:.4f}")
```

```
Closed-Form Solution MSE: 24.2911
```

```python
# Plotting
plt.scatter(y_test, y_pred)
plt.xlabel("Actual Prices")
plt.ylabel("Predicted Prices")
plt.title("Closed-Form Solution: Actual vs Predicted Prices")
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], color='red')
plt.show()
```