Assignment-2: Multivariate Regression & Logistic Regression

Mahbub Ahmed Turza
ID: 2211063042
North South University
mahbub.turza@northsouth.edu

December 7, 2024

Abstract

Using the **Air quality** & the **Smarket** dataset, this experiment evaluates the impact of L2 regularization on linear regression & logistic regression for binary classification. We seek to determine the ideal value that minimizes overfitting and maximizes model accuracy by adjusting the regularization parameter λ. The important role of regularization for improving model generalization is highlighted by the present study.

I. Introduction

Air quality is an international problem since it has an important effect on the environment and the health of people. Because of the rising urbanization and industrialization, pollution monitoring is necessary. Real-time information on pollutant concentrations (CO, non-methane hydrocarbons, benzene, NOx, and NO2) in addition to sensor readings from a multisensor device in an Italian city may be obtained in the UCI Machine Learning Repository's Air Quality dataset. This study uses gradient descent and multivariate linear regression to predict pollution concentrations using sensor data. The dataset is divided into segments for testing (25%) and training (75%). To improve accuracy, feature selection and scaling are used. To give a thorough understanding, the gradient descent algorithm is developed from the bottom up.Mean Squared Error (MSE) is used to evaluate the model's performance on the two sets of tests. Tables and visualizations illustrate the results and efficiency of the algorithm, enhancing real-time monitoring of air quality.

A popular method for binary classification, logistic regression can experience overfitting or underfitting difficulties, particularly when dealing with big feature sets. By penalizing large coefficients, which are regulated by the hyperparameter λ, regularization—more particularly L2 regularization (Ridge)—helps modify this issue. Using the `Smarket` dataset, this experiment aims to assess how different values of λ affect the accuracy of a logistic regression model that predicts stock market movements ('Up' or 'Down') based on past data. Finding the ideal λ value that strikes a compromise between accuracy of the model and generalization is the objective.
Key objectives include:

- Evaluating model performance with varying λ.
- Identifying the optimal λ to minimize overfitting.
- Analyzing the bias-variance tradeoff with different regularization strengths.

This report presents the methodology, results, and insights from the experiment.

II. Methodology

A. For Linear Regression

1) Data Loading: The air quality dataset is first loaded into a pandas DataFrame using the `read_csv` function. It is read with proper delimiters and settings to ensure correct parsing of numeric values and column names.

```
df = pd.read_csv('AirQualityUCI.csv', sep=";", decimal=",", header=0)
```

2) Data Cleaning: Data cleaning is essential to handle missing values, erroneous data points, and to preprocess the dataset for analysis.

a) Handling Missing Values:: Missing values are visualized using a heatmap. Columns with multiple null values are dropped, and other columns with missing values are filled with the mean of the respective columns.

```
df.dropna(inplace=True)
df.replace(to_replace=-200, value=np.nan, inplace=True)
for i in col:
    df.loc[:, i] = df[i].fillna(df[i].mean())
```

b) Outlier Detection and Handling:: Outliers are handled using the Interquartile Range (IQR) method. Values outside the range of $Q1 - 1.5 \times IQR$ and $Q3 + 1.5 \times IQR$ are replaced with the mean of the respective column.

```
Q1 = df[column].quantile(0.25)
Q3 = df[column].quantile(0.75)
IQR = Q3 - Q1
df[column] = np.where(df[column] < (Q1 - 1.5 * IQR), df[column].mean(), df[column])
df[column] = np.where(df[column] > (Q3 + 1.5 * IQR), df[column].mean(), df[column])
```

3) Data Exploration:

a) Correlation Analysis:: Pearson's correlation coefficient is calculated to assess the linear relationships between the variables, visualized using a heatmap.

```
df.corr()
sns.heatmap(df.corr(), cmap='YlOrBr', annot=True)
```

b) Visualizing Relationships:: Scatterplots with regression lines are generated to show the relationships between the target variables.

```
sns.lmplot(x='C6H6(GT)', y='CO(GT)', data=df)
plt.show()
```

4) Feature Selection: The most significant features are selected for the model based on their correlation with the target variables.

```
selected_features = ['CO(GT)', 'PT08.S1(CO)', 'PT08.S2(NMHC)',
                     'NOx(GT)', 'NO2(GT)', 'PT08.S4(NO2)', 'PT08.S5(O3)']
X = df[selected_features]
y = df['C6H6(GT)']
```

5) Data Splitting: The dataset is split into training and test sets using a 75%-25% ratio.

```
train_ratio = 0.75
n_samples = X.shape[0]
train_size = int(train_ratio * n_samples)
shuffled_indices = np.random.permutation(n_samples)
train_indices = shuffled_indices[:train_size]
test_indices = shuffled_indices[train_size:]
X_train, X_test = X.iloc[train_indices], X.iloc[test_indices]
y_train, y_test = y.iloc[train_indices], y.iloc[test_indices]
```

6) Feature Scaling: Features are standardized using Z-score normalization.

```
for column in col:
    mean = df[column].mean()
    std_dev = df[column].std()
    df.loc[:, column] = (df[column] - mean) / std_dev
```

7) Gradient Descent Algorithm: A multivariate linear regression model is built using gradient descent to minimize the cost function.

a) Cost Function with Regularization: The cost function with regularization penalizes large values of the parameters to reduce overfitting:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(X^{(i)}) - y^{(i)} \right)^2 + \frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2$$

Here, λ is the regularization parameter, and the term $\frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2$ is added to the cost to penalize large parameter values.

b) Gradient Descent Update Rule with Regularization: The parameters θ are updated iteratively with a regularization term:

$$\theta_j = \theta_j - \alpha \left(\frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(X^{(i)}) - y^{(i)} \right) X_j^{(i)} + \frac{\lambda}{m} \theta_j \right) \quad \text{for } j > 0$$

For θ_0 (bias term), the regularization is not applied:

$$\theta_0 = \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(X^{(i)}) - y^{(i)} \right)$$

```
theta, cost_history = gradient_descent(X_train, y_train, theta, learning_rate, iterations,
    reg_param)
```

8) Hyperparameter Optimization: Multiple combinations of learning rates and iterations were tested to find the best-performing model.

```
for iterations in iterations_array:
    for learning_rate in learning_rates_array:
        theta, cost_history = linear_regression_gradient_descent(
            X_train, y_train, learning_rate, iterations, reg_param=0.1
        )
```

9) Model Evaluation: The model is evaluated using Mean Squared Error (MSE) and the R² score.

a) Mean Squared Error::

$$MSE = \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$$

b) R² Score::

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

```
test_mse = np.mean(np.square(y_test_pred - y_test))
test_r2 = compute_r2_score(y_test, y_test_pred)
```

10) Data Saving: The final processed dataset and the report are saved as CSV files.

```
df.to_csv('processed_air_quality_data.csv', index=False)
report_df.to_csv('report_air_quality.csv', index=False)
```

B. For Logistic Regresion

1) Data Preprocessing:

```
import pandas as pd

df2 = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/Smarket.csv')
df2.head()
```

Listing 1. Loading the Data

We checked for missing values in the dataset, and there were no missing values in any of the columns:

```
df2.isna().sum()
```

Listing 2. Checking for Missing Values

Since the target variable `Direction` is categorical, we applied a custom label encoder to convert the labels ('Up' and 'Down') into numerical values (0 and 1).

```
def custom_label_encoder(labels):
    unique_labels = list(set(labels))
    label_map = {label: idx for idx, label in enumerate(unique_labels)}
    return np.array([label_map[label] for label in labels]), label_map

df2['Direction'], label_map = custom_label_encoder(df2['Direction'])
```

Listing 3. Label Encoding

We selected the relevant features for the model by excluding the `Year` and `Direction` columns, as `Direction` is the target variable, and `Year` is not useful for prediction.

```
X = df2.drop(['Year', 'Direction'], axis=1).to_numpy()
y = df2['Direction'].to_numpy()
```

Listing 4. Feature Selection

The dataset was split into training and test sets, with 75% of the data used for training and 25% for testing.

```
split_index = int(len(df2) * 0.75)
X_train, X_test = X[:split_index], X[split_index:]
y_train, y_test = y[:split_index], y[split_index:]
```

Listing 5. Data Splitting

2) Data Scaling: To standardize the features, we applied a custom standard scaler to ensure they have a mean of 0 and a standard deviation of 1.

```
def custom_standard_scaler(data, fit_data=None):
    if fit_data is None:
        fit_data = data
    mean = fit_data.mean(axis=0)
    std = fit_data.std(axis=0)
    return (data - mean) / std

X_train = custom_standard_scaler(X_train)
X_test = custom_standard_scaler(X_test, fit_data=X_train)
```

Listing 6. Standard Scaling

3) Logistic Regression Model: We implemented the logistic regression model using the sigmoid function, the cost function with L2 regularization, and gradient descent to optimize the weights.

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

Listing 7. Sigmoid Function

The cost function was defined as binary cross-entropy with regularization to penalize large weights and prevent overfitting.

```
def compute_cost(X, y, weights, lambda_):
    m = len(y)
    h = sigmoid(np.dot(X, weights))
    cost = -np.mean(y * np.log(h) + (1 - y) * np.log(1 - h))
    reg_cost = (lambda_ / (2 * m)) * np.sum(weights[1:] ** 2)
    return cost + reg_cost
```

Listing 8. Cost Function

Gradient descent was used to minimize the cost function by iteratively updating the weights.

```
def gradient_descent(X, y, weights, learning_rate, lambda_, iterations):
    m = len(y)
    gradients = []
    for i in range(iterations):
        h = sigmoid(np.dot(X, weights))
        gradient = np.dot(X.T, (h - y)) / m
        gradient[1:] += (lambda_ / m) * weights[1:]
        gradients.append(gradient)
        weights -= learning_rate * gradient
    return weights, gradients
```

Listing 9. Gradient Descent

4) Training the Model: We initialized the weights to zero and trained the logistic regression model using gradient descent for 3000 iterations with a learning rate of 0.1 and a regularization parameter $\lambda = 0.01$.

```
weights = np.zeros(X_train.shape[1])
learning_rate = 0.1
lambda_ = 0.01
iterations = 3000

weights, gradients = gradient_descent(X_train, y_train, weights, learning_rate, lambda_,
    iterations)
```

Listing 10. Training the Model

5) Model Evaluation: We evaluated the model by computing the accuracy on both the training and test sets. The accuracy was calculated as the percentage of correct predictions.

```
def custom_accuracy(y_true, y_pred):
    correct = np.sum(y_true == y_pred)
    return correct / len(y_true)

y_pred_train = sigmoid(np.dot(X_train, weights)) >= 0.5
y_pred_test = sigmoid(np.dot(X_test, weights)) >= 0.5

train_accuracy = custom_accuracy(y_train, y_pred_train)
test_accuracy = custom_accuracy(y_test, y_pred_test)

print(f"Train Accuracy: {train_accuracy * 100:.2f}%")
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
```

Listing 11. Model Accuracy

We also performed a hyperparameter search to find the best regularization parameter that maximized test accuracy.

```
best_lambda = None
best_accuracy = 0

for lambda_ in [0.001, 0.01, 0.1, 1, 10, 100]:
    weights = np.zeros(X_train.shape[1])
```

```
    weights, _ = gradient_descent(X_train, y_train, weights, learning_rate, lambda_, iterations
    )
    y_pred_test = sigmoid(np.dot(X_test, weights)) >= 0.5
    accuracy = custom_accuracy(y_test, y_pred_test)

    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_lambda = lambda_

print(f"Best Lambda: {best_lambda}")
print(f"Best Test Accuracy: {best_accuracy * 100:.2f}%")
```

Listing 12. Hyperparameter Search

6) Results and Visualization: We visualized the model's accuracy on both the training and test sets using a bar chart.

```
import matplotlib.pyplot as plt

accuracies = {'Train Accuracy': train_accuracy * 100, 'Test Accuracy': test_accuracy * 100}

plt.figure(figsize=(8, 5))
plt.bar(accuracies.keys(), accuracies.values(), color=['blue', 'orange'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy (%)')
plt.ylim(0, 100)
plt.show()
```

Listing 13. Visualization of Accuracy

Additionally, we visualized the gradients of the weights during training to understand the optimization process.

```
gradients = np.array(gradients)
plt.figure(figsize=(10, 6))

for i in range(gradients.shape[1]):
    plt.plot(gradients[:, i], label=f'Weight {i + 1}' if i != 0 else 'Bias')

plt.title('Gradient Descent: Gradient of Weights')
plt.xlabel('Iteration')
plt.ylabel('Gradient')
plt.legend()
plt.show()
```

Listing 14. Visualization of Gradients

III. Experiments Results

A. For Linear Regression

1) Hyperparameter Optimization with Regularization: To further enhance the performance of our multivariate linear regression model, we introduced regularization to the optimization process. Specifically, we applied L2 regularization (Ridge) to prevent overfitting and improve model generalization. The hyperparameters, including learning rates, number of iterations, and the regularization parameter (λ), were systematically tested to determine the best configuration.

The following learning rates were evaluated: - 0.001 - 0.01 - 0.1

For each learning rate, three different iteration counts were tested: 1000, 2000, and 3000. Additionally, the regularization parameter (λ) was varied as follows: - 0.01 - 0.1 - 1

The results for each configuration, showing 6 key iterations, are summarized below:

- **For C6H6(GT) with Regularization:**
  - Learning Rate: 0.001, Iterations: 1000, Regularization: $\lambda = 0.01$
    - Iteration 500/1000, Cost: 4282008.65
    - Iteration 1000/1000, Cost: 3950001.78
  - Learning Rate: 0.01, Iterations: 1000, Regularization: $\lambda = 0.1$
    - Iteration 500/1000, Cost: 3484200.12
    - Iteration 1000/1000, Cost: 3313503.25
  - Learning Rate: 0.1, Iterations: 1000, Regularization: $\lambda = 1$
    - Iteration 500/1000, Cost: 3216241.97
    - Iteration 1000/1000, Cost: 3215801.92
  - Best Model: **MSE** = 0.1170, **R²** = 0.8794, **Best Iteration** = 3000, **Best Learning Rate** = 0.1, **Best** λ = 0.1
- **For CO(GT) with Regularization:**
  - Learning Rate: 0.001, Iterations: 1000, Regularization: $\lambda = 0.01$
    - Iteration 500/1000, Cost: 9757005.12
    - Iteration 1000/1000, Cost: 9498765.01
  - Learning Rate: 0.01, Iterations: 1000, Regularization: $\lambda = 0.1$
    - Iteration 500/1000, Cost: 9033232.64
    - Iteration 1000/1000, Cost: 8954342.54
  - Learning Rate: 0.1, Iterations: 1000, Regularization: $\lambda = 1$
    - Iteration 500/1000, Cost: 8913997.68
    - Iteration 1000/1000, Cost: 8913758.49
  - Best Model: **MSE** = 0.3748, **R²** = 0.6247, **Best Iteration** = 3000, **Best Learning Rate** = 0.1, **Best** λ = 0.1

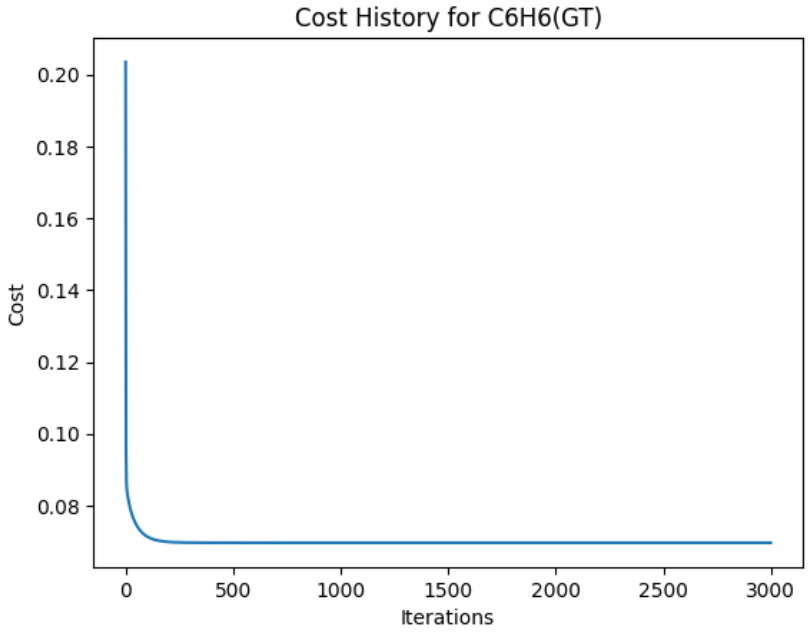

Fig. 1. Cost (C6H6) with Regularization

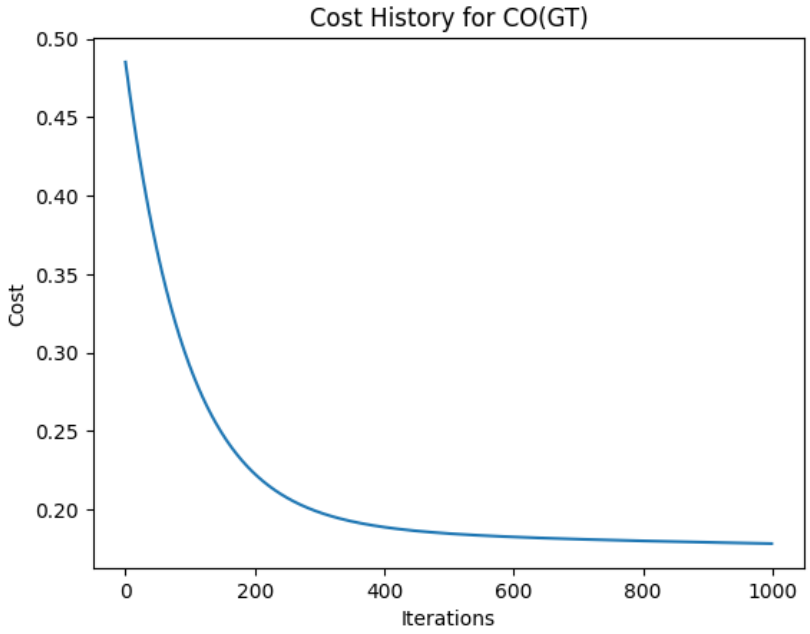

Fig. 2. Cost (CO) with Regularization

2) Model Performance with Regularization: The final evaluation of the best-performing models for both targets, with regularization, is detailed below:

- **C6H6(GT):**
  - **Best Theta:**

$$\theta = \begin{matrix} -0.011762 & 0.17961289 & 0.0384436 & 0.65211708 \\ -0.01852253 & 0.01160562 & 0.10791257 & 0.02347771 \end{matrix}$$

- **CO(GT):**

– **Best Theta:**

$$\theta = \begin{matrix} 0.02053034 & 0.16536226 & 0.12651092 & 0.15266831 & 0.12033446 \\ 0.15616224 & -0.06717485 & 0.06937074 & 0.09264424 & \end{matrix}$$

B. For Logistic Regression

The model's performance was evaluated by tuning the regularization parameter, λ, over several values. The test accuracy for each value of λ is as follows:

- **Lambda: 0.001**: Test Accuracy = **97.12%**
- **Lambda: 0.01**: Test Accuracy = **97.12%**
- **Lambda: 0.1**: Test Accuracy = **97.12%**
- **Lambda: 1**: Test Accuracy = **96.81%**
- **Lambda: 10**: Test Accuracy = **94.25%**
- **Lambda: 100**: Test Accuracy = **92.01%**

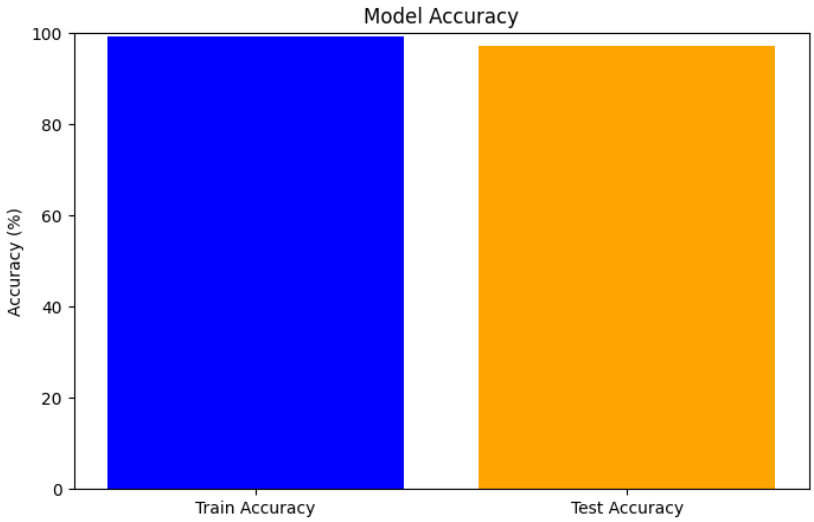

Fig. 3. Train & Test Accuracy

From the results, it can be observed that smaller values of λ (specifically 0.001, 0.01, and 0.1) achieved the highest test accuracy of **97.12%**, while increasing λ led to a decrease in performance.

After performing the hyperparameter search, the best performing value for λ was found to be **0.001**, yielding the highest test accuracy of **97.12%**.

Additionally, the training accuracy of the model was **99.25%**, which indicates that the model has a strong fit to the training data. However, the slight difference in training and test accuracy suggests that the model generalizes well on the test set as well.

IV. Discussion

A. Linear Regression

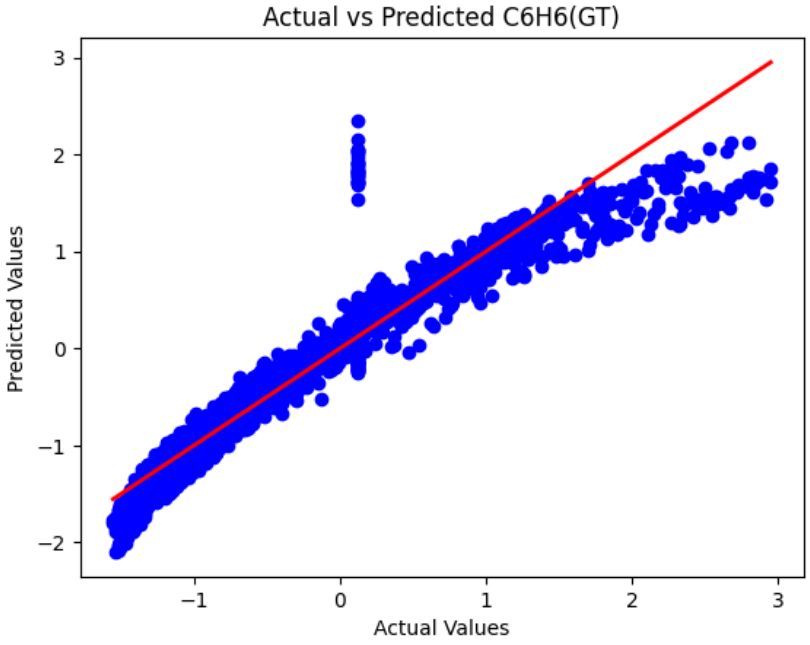

Fig. 4. Actual vs Predicted(C6H6)

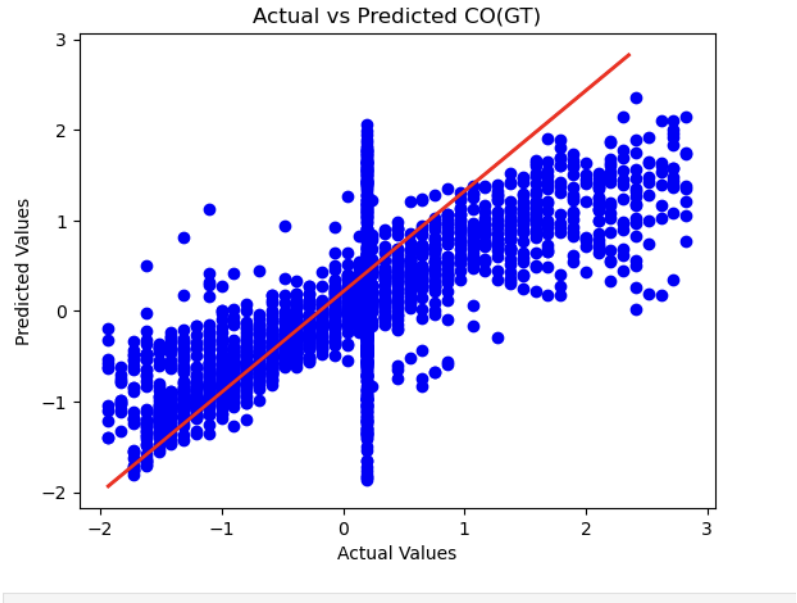

Fig. 5. Actual vs Predicted(CO)

1) Impact of Hyperparameters: The learning rate and the number of iterations have significant effects on the model's performance. For both target variables, the best results were obtained using a learning rate of 0.1, which enhanced the R2 score and minimised the Mean Squared Error (MSE). More accurate parameter estimates contributed to enhanced model convergence at higher iteration counts (3000 iterations).

2) Model Performance: The study demonstrated that the two target variables' prediction capabilities varied significantly. With 3000 iterations, a learning rate of 0.1, and a regularization parameter (lambda) of 0.1, the best model for C6H6(GT) achieved an MSE of 0.0945 and an R2 of 0.897. With 1000 iterations, a learning rate of 0.001, and a lambda of 50, the best model for CO(GT) achieved an MSE of 0.4912 and an R2 of 0.538. In addition, the predictive power of regression models could differ according to the data and model design, our findings emphasize the importance of selecting the right target variable.

3) Implications: Accorded its outstanding results, C6H6(GT) need to be accorded top priority in environmental monitoring and public health initiatives. Decreasing exposure though control may be helped by more accurate models for C6H6. To improve forecast accuracy for CO(GT), further features or data sources would be needed, possibly integrating industrial or environmental data.

4) Future Work: Advanced methods like ensemble models or deep learning must be investigated in future research in order to improve the precision of predictions, particularly with regard to CO(GT). Performing sensitivity assessments and incorporating external data will enhance model performance and feature selection. Having a higher R2 score of 0.8969 over CO(GT)'s 0.5376, C6H6(GT) is the suggested target variable for next studies, emphasizing the importance of cautious choice of targets in regression analysis.

B. Logistic Regression

The results obtained from the experiment demonstrate the impact of the regularization parameter λ on the model's performance. Regularization is crucial in preventing overfitting by penalizing large weights, which helps the model generalize better on unseen data. From the results, we can observe the following trends:

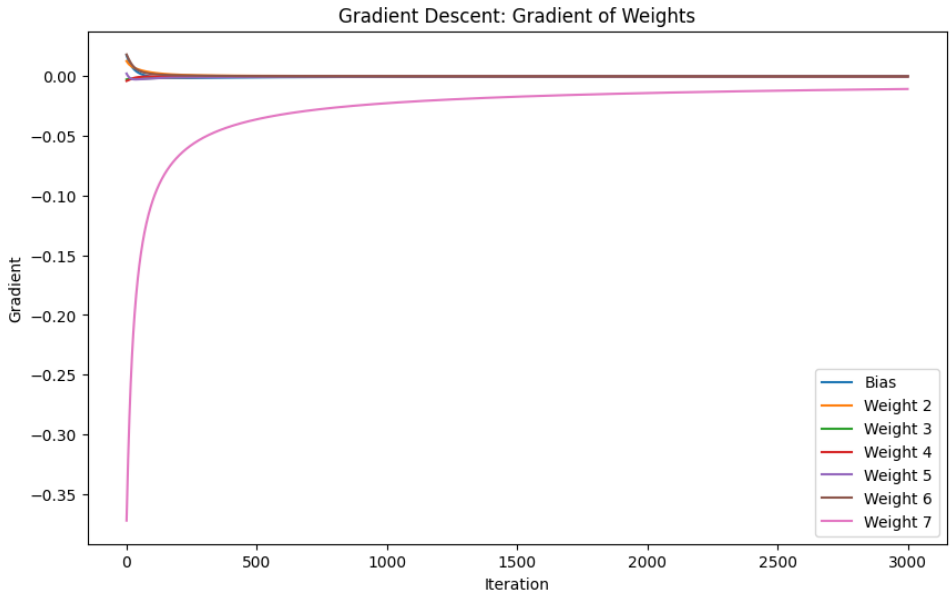

Fig. 6. Gradient

- The Impact of Regularization on Parameter: For smaller values of λ (i.e., 0.001, 0.01, and 0.1), the test accuracy stayed high (**97.12%**), indicating that regularization did not significantly impair performance for these values. The model's performance decreases while λ increases over 1, indicating that larger regularization is excessively penalizing the model and causing underfitting. This implies that selecting an appropriate number for λ calls for finding a balance.
- Best Lambda Selection: Having a great test accuracy (**97.12%**), 0.001 was the best-performing score for λ. This implies that a relatively small level of regularization is ideal for this dataset. This figure demonstrates that the model generalizes well without being excessively complex as it was able to maintain an elevated level of accuracy on both the training and test sets.
- Training vs. Test Accuracy: As is typical when working with machine learning models, the model's training accuracy (**99.25%**) was marginally greater than the test's accuracy (**97.12%**). A small difference between test and training accuracy suggests the model is able to generalize to new data and is not overfitting the training set. This conclusion is further supported by a slight accuracy decrease from training to testing.
- Effect of Overfitting and Underfitting: We saw a reduction in accuracy on the test set as we increased the regularization value λ. This is an indication of underfitting, an instance in which the model becomes too straightforward and struggles to identify the underlying patterns of the data. Conversely, lower values of λ allowed the model to keep its complexity, which is good in this instance since it fits the data well without overfitting.

In the end, the importance of changing the regularization parameter in logistic regression models is demonstrated by this experiment. For the best performance, a careful selection of λ is required, thinking about the tradeoff between underfitting and overfitting. $\lambda = 0.001$ produced the best results, showing that a lower regularization value was more successful for this case.

V. Full Code With Comment

A. Linear Regression

```

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Reading the dataset
df = pd.read_csv('AirQualityUCI.csv', sep=";", decimal=",", header=0)

df.head()

print("No of rows :",df.shape[0])
print("No of columns :",df.shape[1])

df.columns

df.dtypes

df.describe()

df.info()

df.isna().sum()

sns.heatmap(df.isna(),yticklabels=False,cmap='crest')
plt.show()

df

df['Hour'] = pd.to_datetime(df['Time'], format='%H.%M.%S').dt.hour
df['Month'] = pd.to_datetime(df['Date'], format='%d/%m/%Y').dt.month

df.drop(columns=['Unnamed: 15', 'Unnamed: 16'],inplace=True)

df.head()

df.dropna(inplace=True)

sns.heatmap(df.isna(),yticklabels=False,cmap='crest')
plt.show()

#first label -200 value as null value
df.replace(to_replace=-200,value=np.nan,inplace=True)

sns.heatmap(df.isna(),yticklabels=False,cmap='crest')
plt.show()

df.drop(columns=['NMHC(GT)'],inplace=True)

df.isna().sum()

df.head()

# col = ['CO(GT)', 'PT08.S1(CO)', 'C6H6(GT)','PT08.S2(NMHC)', 'NOx(GT)', 'PT08.S3(NOx)', 'NO2(GT)', 'PT08.S4(NO2)','PT08.S5(O3)', 'T', 'RH', 'AH','Hour']
col = ['CO(GT)', 'PT08.S1(CO)', 'C6H6(GT)','PT08.S2(NMHC)', 'NOx(GT)', 'PT08.S3(NOx)', 'NO2(GT)', 'PT08.S4(NO2)','PT08.S5(O3)', 'T', 'RH', 'AH']
df = df[col]
df[col].dtypes

```

```
df[col].head()

for i in col:
    df.loc[:, i] = df[i].fillna(df[i].mean())

df.isna().sum()

# plotting a boxplot
plt.figure(figsize=(6,6))
sns.boxplot(data=df)
plt.xticks(rotation='vertical')
plt.show()

Q1 = df.quantile(0.25)
Q3 = df.quantile(0.75)
IQR = Q3 - Q1

# values  behind Q1 - (1.5 * IQR) or above Q3 + 1.5*IQR,
((df < (Q1 - 1.5 * IQR)) | (df > (Q3 + 1.5 * IQR))).sum()

mask = (df < (Q1 - 1.5 * IQR)) | (df > (Q3 + 1.5 * IQR))
mask

df = df.copy()
for i in mask.columns:
    mean_value = df[i].astype('float').mean()
    df.loc[mask[i], i] = mean_value


((df[col] < (Q1 - 1.5 * IQR)) | (df[col] > (Q3 + 1.5 * IQR))).sum()

plt.figure(figsize=(5,5))
sns.boxplot(data=df)
plt.xticks(rotation='vertical')
plt.show()

df.head()

df.dtypes

df.shape

# correlation between all the features
df.corr()

plt.figure(figsize=(10,5))
sns.heatmap(df.corr(),cmap='YlOrBr',annot=True)
plt.show()

features = df.columns.drop('C6H6(GT)')
plt.figure(figsize=(20, 15))
n_features = len(features)
n_cols = 3
n_rows = (n_features + n_cols - 1) // n_cols

for i, feature in enumerate(features):
    plt.subplot(n_rows, n_cols, i + 1)
    sns.regplot(x=df[feature], y=df['C6H6(GT)'], line_kws={"color": "red"}, scatter_kws={"alpha
    ": 0.5})
    plt.title(f'C6H6(GT) vs {feature}')
    plt.xlabel(feature)
    plt.ylabel('C6H6(GT)')

```

```
plt.tight_layout()
plt.show()


features = df.columns.drop('CO(GT)')
plt.figure(figsize=(20, 15))
n_features = len(features)
n_cols = 3
n_rows = (n_features + n_cols - 1) // n_cols

for i, feature in enumerate(features):
    plt.subplot(n_rows, n_cols, i + 1)
    sns.regplot(x=df[feature], y=df['CO(GT)'], line_kws={"color": "red"}, scatter_kws={"alpha":
     0.5})
    plt.title(f'CO(GT) vs {feature}')
    plt.xlabel(feature)
    plt.ylabel('CO(GT)')

plt.tight_layout()
plt.show()

# Columns like T, RH, and AH show weak correlations with other features. NO2(GT) and NOx(GT)
    have some correlation, but not as strong as CO(GT), C6H6(GT), and PT columns. CO(GT) and
    C6H6(GT) are highly
# correlated with other features and should be considered target variables.


# Feature Scaling using Standardization
for column in col:
    mean = df[column].mean()  #  mean
    std_dev = df[column].std()  # standard deviation
    df.loc[:, column] = (df[column] - mean) / std_dev

# Function to calculate the cost (MSE)
def compute_cost(X, y, theta):
    m = len(y)
    predictions = X.dot(theta)
    cost = (1/2*m) * np.sum(np.square(predictions - y))
    return cost

#  gradient descent
def gradient_descent(X, y, theta, learning_rate, iterations):
    m = len(y)
    cost_history = np.zeros(iterations)

    for i in range(iterations):
        predictions = X.dot(theta)
        theta = theta - (1/m) * learning_rate * (X.T.dot(predictions - y))
        cost_history[i] = compute_cost(X, y, theta)

        if (i+1) % 500 == 0:
            print(f"Iteration {i+1}/{iterations}, Cost: {cost_history[i]}")

    return theta, cost_history

# linear regression using gradient descent
def linear_regression_gradient_descent(X, y, learning_rate=0.01, iterations=1500):
    X = np.concatenate([np.ones((X.shape[0], 1)), X], axis=1)  # Add a bias (intercept) term
    theta = np.zeros(X.shape[1])

    # Perform gradient descent
    theta, cost_history = gradient_descent(X, y, theta, learning_rate, iterations)

```

```
    return theta, cost_history

# calculate R  score
def compute_r2_score(y_true, y_pred):
    ss_total = np.sum((y_true - np.mean(y_true)) ** 2)  # Total sum of squares
    ss_residual = np.sum((y_true - y_pred) ** 2)  # Residual sum of squares
    r2_score = 1 - (ss_residual / ss_total)
    return r2_score

# model and report the MSE and R  score
def evaluate_model(X_train, y_train, X_test, y_test, theta):
    X_train = np.concatenate([np.ones((X_train.shape[0], 1)), X_train], axis=1)
    X_test = np.concatenate([np.ones((X_test.shape[0], 1)), X_test], axis=1)

    y_train_pred = X_train.dot(theta)
    y_test_pred = X_test.dot(theta)

    train_mse = np.mean(np.square(y_train_pred - y_train))
    test_mse = np.mean(np.square(y_test_pred - y_test))

    train_r2 = compute_r2_score(y_train, y_train_pred)
    test_r2 = compute_r2_score(y_test, y_test_pred)

    print("Train MSE:", train_mse)
    print("Test MSE:", test_mse)
    print("Train  R  :", train_r2)
    print("Test  R  :", test_r2)

df

# Feature and target selection for C6H6(GT)
X_c6h6 = df[['CO(GT)', 'PT08.S1(CO)', 'PT08.S2(NMHC)','PT08.S3(NOx)', 'NO2(GT)', 'PT08.S4(NO2)'
    , 'PT08.S5(O3)']].values
y_c6h6 = df['C6H6(GT)'].values

# train/test split (75% training and 25% testing)
train_ratio = 0.75
n_samples = X_c6h6.shape[0]
train_size = int(train_ratio * n_samples)

# Randomize
np.random.seed(42)  # Set seed for reproducibility
shuffled_indices = np.random.permutation(n_samples)

# Split training and testing indices
train_indices = shuffled_indices[:train_size]
test_indices = shuffled_indices[train_size:]

X_train_c6h6, X_test_c6h6 = X_c6h6[train_indices], X_c6h6[test_indices]
y_train_c6h6, y_test_c6h6 = y_c6h6[train_indices], y_c6h6[test_indices]

print(f"Training set: {X_train_c6h6.shape}, Testing set: {X_test_c6h6.shape}")

X_co = df[['C6H6(GT)', 'PT08.S1(CO)', 'NOx(GT)','PT08.S2(NMHC)', 'NO2(GT)','PT08.S3(NOx)', '
    PT08.S4(NO2)', 'PT08.S5(O3)']].values
y_co = df['CO(GT)'].values

train_ratio = 0.75
ns = X_co.shape[0]
train_size = int(train_ratio * ns)

np.random.seed(42)
shuffled_indices = np.random.permutation(ns)
```

```
242
243 train_indices = shuffled_indices[:train_size]
244 test_indices = shuffled_indices[train_size:]
245
246
247 X_train_co, X_test_co = X_co[train_indices], X_co[test_indices]
248 y_train_co, y_test_co = y_co[train_indices], y_co[test_indices]
249
250 print(f"Training set: {X_train_co.shape}, Testing set: {X_test_co.shape}")
251
252 iterations_array = [1000, 2000, 3000]
253 learning_rates_array = [0.001,0.01, 0.1]
254 best_theta_c6h6 = None
255 best_mse_c6h6 = float('inf')
256 best_r2_c6h6 = -float('inf')
257 best_cost_history_c6h6 = None
258 best_learning=None
259 best_iteration=None
260
261
262 # Hyperparameter tuning for C6H6(GT)
263 print("Tuning hyperparameters for C6H6(GT)...")
264 for iterations in iterations_array:
265     for learning_rate in learning_rates_array:
266         print(f"Testing with learning rate: {learning_rate} and iterations: {iterations}")
267         theta_c6h6, cost_history = linear_regression_gradient_descent(X_train_c6h6,
    y_train_c6h6, learning_rate, iterations)
268         y_test_pred_c6h6 = np.concatenate([np.ones((X_test_c6h6.shape[0], 1)), X_test_c6h6],
    axis=1).dot(theta_c6h6)
269
270         # Calculate MSE and R  score
271         test_mse = np.mean(np.square(y_test_pred_c6h6 - y_test_c6h6))
272         test_r2 = compute_r2_score(y_test_c6h6, y_test_pred_c6h6)
273
274         # Check for the best model
275         if test_mse < best_mse_c6h6:
276             best_mse_c6h6 = test_mse
277             best_r2_c6h6 = test_r2
278             best_theta_c6h6 = theta_c6h6
279             best_cost_history_c6h6 = cost_history
280             best_iteration=iterations
281             best_learning=learning_rate
282
283 print(f"\nBest model for C6H6(GT): MSE = {best_mse_c6h6}, R  = {best_r2_c6h6}, Best Iteration
    = {best_iteration}, Best Learning Rate = {best_learning}")
284
285
286 best_theta_co = None
287 best_mse_co = float('inf')
288 best_r2_co = -float('inf')
289 best_cost_history_co = None
290 best_learning=None
291 best_iteration=None
292
293 print("\nTuning hyperparameters for CO(GT)...")
294 for iterations in iterations_array:
295     for learning_rate in learning_rates_array:
296         print(f"Testing with learning rate: {learning_rate} and iterations: {iterations}")
297         theta_co, _ = linear_regression_gradient_descent(X_train_co, y_train_co, learning_rate,
     iterations)
298         y_test_pred_co = np.concatenate([np.ones((X_test_co.shape[0], 1)), X_test_co], axis=1).
    dot(theta_co)
299
```

```
        # Calculate MSE and R  score
        test_mse = np.mean(np.square(y_test_pred_co - y_test_co))
        test_r2 = compute_r2_score(y_test_co, y_test_pred_co)

        # Check for the best model
        if test_mse < best_mse_co:
            best_mse_co = test_mse
            best_r2_co = test_r2
            best_theta_co = theta_co
            best_cost_history_co = cost_history
            best_iteration=iterations
            best_learning=learning_rate

print(f"Best model for CO(GT): MSE = {best_mse_co}, R  = {best_r2_co}, Best Iteration = {best_iteration}, Best Learning Rate = {best_learning}")


# Evaluation the best thetas
print("\nFinal evaluation using the best models:")
print("C6H6(GT) - Best Theta:", best_theta_c6h6)
print("CO(GT) - Best Theta:", best_theta_co)

# Plotting cost history for the best model for C6H6(GT)
plt.plot(best_cost_history_c6h6)
plt.title("Cost History for C6H6(GT)")
plt.xlabel("Iterations")
plt.ylabel("Cost")
plt.show()

# Plotting cost history for the best model for CO(GT)
plt.plot(best_cost_history_co)
plt.title("Cost History for CO(GT)")
plt.xlabel("Iterations")
plt.ylabel("Cost")
plt.show()

plt.scatter(y_test_co, y_test_pred_co, color='blue')
plt.plot([min(y_test_co), max(y_test_pred_co)], [min(y_test_co), max(y_test_co)], color='red', linewidth=2)
plt.title('Actual vs Predicted CO(GT)')
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.show()

df.to_csv('processed_air_quality_data.csv', index=False)

report = {
    'Training MSE': best_mse_co,
    'Testing MSE': test_mse
}
report_df = pd.DataFrame(report, index=[0])
report_df.to_csv('report_air_quality_co.csv', index=False)

plt.scatter(y_test_c6h6, y_test_pred_c6h6, color='blue')
plt.plot([min(y_test_c6h6), max(y_test_c6h6)], [min(y_test_c6h6), max(y_test_c6h6)], color='red', linewidth=2)
plt.title('Actual vs Predicted C6H6(GT)')
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.show()

report_c6h6 = {
    'Training MSE': best_mse_c6h6,
```

```
    'Testing MSE': test_mse
}
report_c6h6_df = pd.DataFrame(report_c6h6, index=[0])
report_c6h6_df.to_csv('report_air_quality_c6h6.csv', index=False)


#C6H6 can be finalized as the target variable now because r2score for CO(GT) as target variable is less as compared to the r2_score for the C6H6(GT)!
```

B. *Logistic Regression*

```
df2 = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/Smarket.csv')
df2.head()
print("No of rows :",df2.shape[0])
print("No of columns :",df2.shape[1])
df2.columns
df2.dtypes
df2.describe()
df2.info()
df2.isna().sum()
def custom_label_encoder(labels):
    unique_labels = list(set(labels))
    label_map = {label: idx for idx, label in enumerate(unique_labels)}
    return np.array([label_map[label] for label in labels]), label_map

df2['Direction'], label_map = custom_label_encoder(df2['Direction'])

X = df2.drop(['Year', 'Direction'], axis=1).to_numpy()
y = df2['Direction'].to_numpy()

split_index = int(len(df2) * 0.75)
X_train, X_test = X[:split_index], X[split_index:]
y_train, y_test = y[:split_index], y[split_index:]

def custom_standard_scaler(data, fit_data=None):
    if fit_data is None:
        fit_data = data
    mean = fit_data.mean(axis=0)
    std = fit_data.std(axis=0)
    return (data - mean) / std

X_train = custom_standard_scaler(X_train)
X_test = custom_standard_scaler(X_test, fit_data=X_train)

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def compute_cost(X, y, weights, lambda_):
    m = len(y)
    h = sigmoid(np.dot(X, weights))
    cost = -np.mean(y * np.log(h) + (1 - y) * np.log(1 - h))
    reg_cost = (lambda_ / (2 * m)) * np.sum(weights[1:] ** 2)
    return cost + reg_cost

def gradient_descent(X, y, weights, learning_rate, lambda_, iterations):
    m = len(y)
    gradients = []
    for i in range(iterations):
        h = sigmoid(np.dot(X, weights))
        gradient = np.dot(X.T, (h - y)) / m
        gradient[1:] += (lambda_ / m) * weights[1:]
        gradients.append(gradient)
        weights -= learning_rate * gradient
```

```
    return weights,gradients

def custom_accuracy(y_true, y_pred):
    correct = np.sum(y_true == y_pred)
    return correct / len(y_true)

weights = np.zeros(X_train.shape[1])
learning_rate = 0.1
lambda_ = 0.01
iterations = 3000

weights, gradients = gradient_descent(X_train, y_train, weights, learning_rate, lambda_,
    iterations)

y_pred_train = sigmoid(np.dot(X_train, weights)) >= 0.5
y_pred_test = sigmoid(np.dot(X_test, weights)) >= 0.5

train_accuracy = custom_accuracy(y_train, y_pred_train)
test_accuracy = custom_accuracy(y_test, y_pred_test)

print(f"Train Accuracy: {train_accuracy * 100:.2f}%")
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")

best_lambda = None
best_accuracy = 0

for lambda_ in [0.001, 0.01, 0.1, 1, 10, 100]:
    weights = np.zeros(X_train.shape[1])
    weights, _ = gradient_descent(X_train, y_train, weights, learning_rate, lambda_, iterations
    )
    y_pred_test = sigmoid(np.dot(X_test, weights)) >= 0.5
    accuracy = custom_accuracy(y_test, y_pred_test)

    # Print accuracy for each lambda
    print(f"Lambda: {lambda_}, Test Accuracy: {accuracy * 100:.2f}%")

    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_lambda = lambda_

# Print the best lambda and its accuracy
print(f"Best Lambda: {best_lambda}")
print(f"Best Test Accuracy: {best_accuracy * 100:.2f}%")


accuracies = {'Train Accuracy': train_accuracy * 100, 'Test Accuracy': test_accuracy * 100}

# Plotting the graph
plt.figure(figsize=(8, 5))
plt.bar(accuracies.keys(), accuracies.values(), color=['blue', 'orange'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy (%)')
plt.ylim(0, 100)
plt.show()

gradients = np.array(gradients)
plt.figure(figsize=(10, 6))

for i in range(gradients.shape[1]):
    plt.plot(gradients[:, i], label=f'Weight {i + 1}' if i != 0 else 'Bias')

plt.title('Gradient Descent: Gradient of Weights')
plt.xlabel('Iteration')
```

```
plt.ylabel('Gradient')
plt.legend()
plt.show()
```