University of Isfahan

Multiple Linear Regression Project

Authors:

Melika Shirian

Kianoosh Vadaei

Fundamentals and Applications of Artificial Intelligence

Dr. Hossein Karshenas

2023-11-08

Introduction

Air travel plays a crucial role in our lives, and understanding how flight prices are determined is essential. This project focuses on employing Multiple Linear Regression (MLR) to explore the relationship between different flight details and their prices.

Our dataset includes factors such as departure locations, flight duration, day of the week, and airline information. Through MLR, we aim to identify patterns that can help us create a model for predicting flight prices. MLR is a useful tool as it allows us to consider how multiple factors work together to influence prices.

This project aims to provide insights into the factors affecting flight prices, benefiting both travelers seeking affordability and airlines looking to optimize pricing strategies.

By utilizing Multiple Linear Regression, we embark on a journey to uncover the connections between various flight aspects and their costs, contributing to a better understanding of the economics of air travel.

Background

Understanding the complex dynamics of flight pricing involves delving into the intricacies of Multiple Linear Regression (MLR), a statistical method that helps unravel the relationships between multiple factors and a target variable, in this case, flight prices.

Multiple Linear Regression is a statistical technique used to analyze the relationship between two or more independent variables and a single dependent variable. In the context of our project, independent variables might include departure locations, flight duration, day of the week, and airline details. The aim is to determine how these variables collectively influence the dependent variable—flight prices.

MLR goes beyond simple linear regression by accommodating multiple factors simultaneously. It assumes that the relationship between the independent variables and the dependent variable is linear and aims to find the best-fit line that represents this relationship. This method allows us to assess the individual impact of each variable while considering the presence of others.

In summary, Multiple Linear Regression is a valuable tool for our project as it enables us to examine and quantify the influence of various factors on flight prices, providing a nuanced understanding of the interconnected elements shaping the economics of air travel.

Methodology

Our dataset consists of 270,139 records with columns such as departure_time, stops, arrival_time, class, duration, days_left, and the variable of interest, price, which we aim to predict.

For the implementation of Multiple Linear Regression (MLR), we conducted exploratory data analysis and then preprocessed the data, handling missing values and ensuring uniform formatting.

To enhance the MLR model's performance, we labeled our categorical variables, such as class, using the one-hot encoding method. This approach converts categorical variables into numerical values, improving the model's ability to interpret and learn from these features.

It is essential to note that we made no alterations to the flight data itself. All relevant variables were extracted directly from the input CSV file using the pandas library in Python.

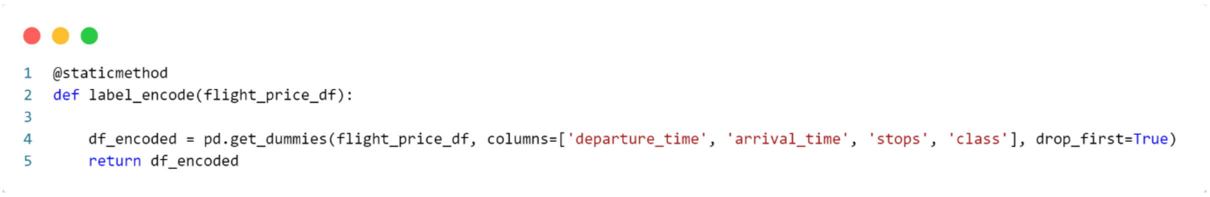
By adopting these steps in our methodology, we aimed to develop a robust MLR model capable of predicting flight prices based on the given features, providing valuable insights into the factors influencing the cost of air travel.

All significant implementations of this problem were done by us.

Code Explanation

In this project, all model implementations are carried out in the **MachineLearning** module. Now, let's delve into the details.

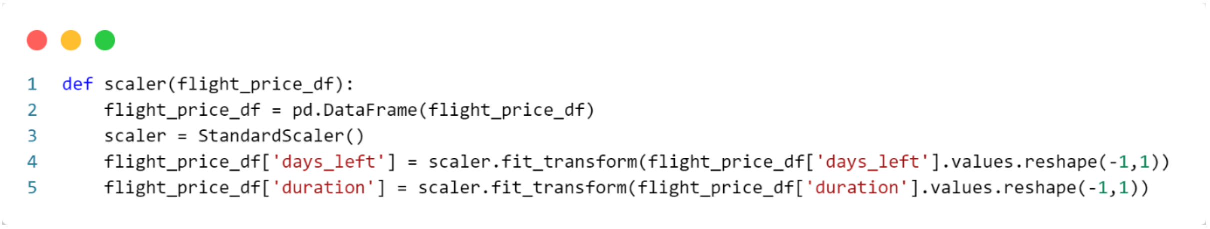
MachineLearning Module:



```python
@staticmethod
def label_encode(flight_price_df):

    df_encoded = pd.get_dummies(flight_price_df, columns=['departure_time', 'arrival_time', 'stops', 'class'], drop_first=True)
    return df_encoded
```

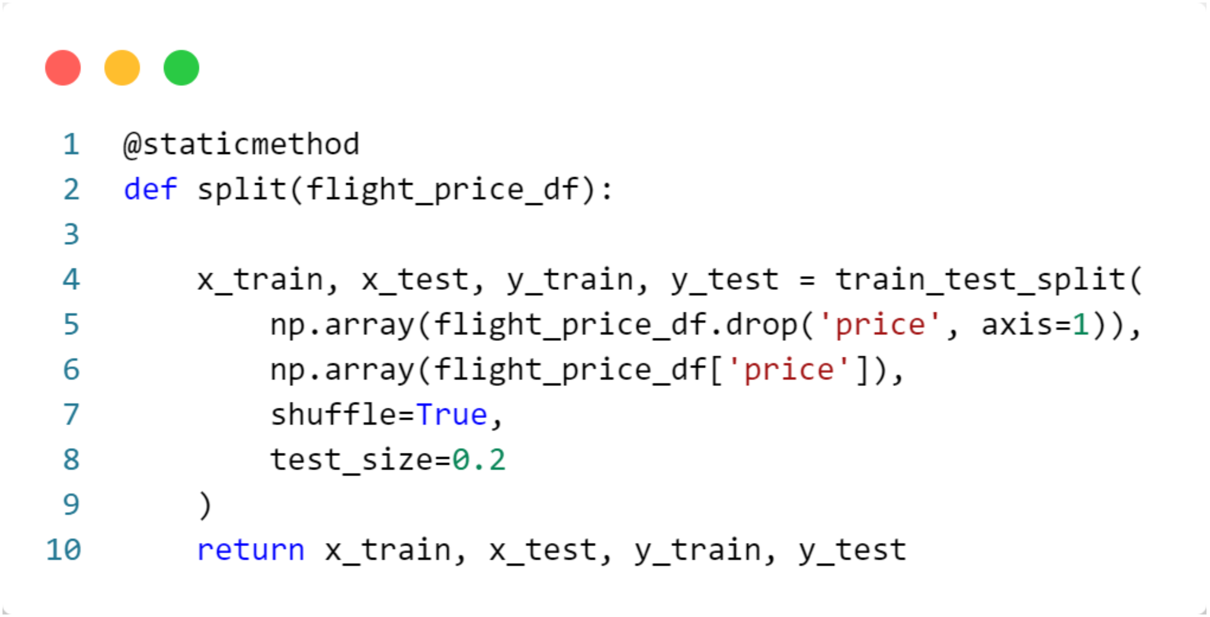
In the label_encode function, using the pandas library, the features 'departure_time,' 'arrival_time,' 'stops,' and 'class'—which have nominal and ordinal values—are encoded with the one-hot labeling method. After labeling, these features themselves are removed from the dataset for further processing.

Note that **flight_price_df** is our dataset dataframe.



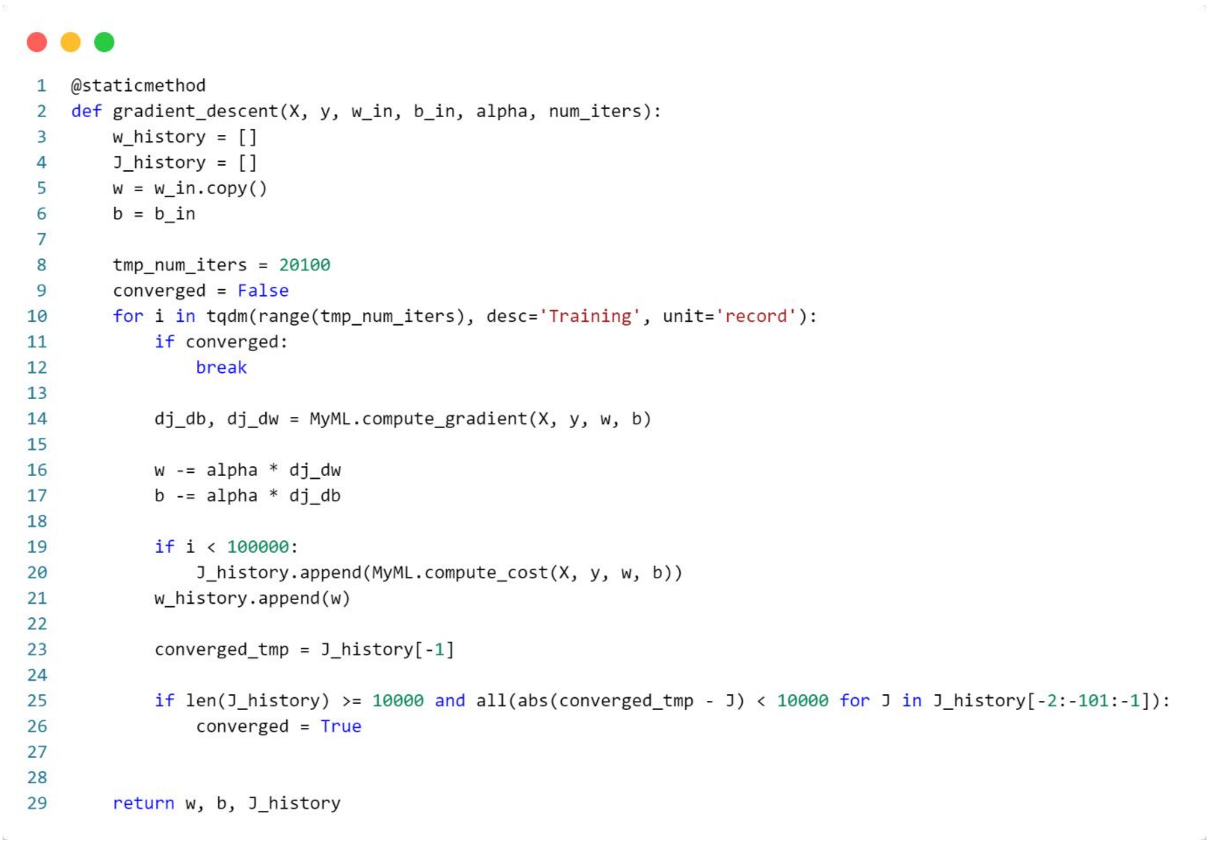
```python
def scaler(flight_price_df):
    flight_price_df = pd.DataFrame(flight_price_df)
    scaler = StandardScaler()
    flight_price_df['days_left'] = scaler.fit_transform(flight_price_df['days_left'].values.reshape(-1,1))
    flight_price_df['duration'] = scaler.fit_transform(flight_price_df['duration'].values.reshape(-1,1))
```

In the **scaler** function, using the sklearn library and the StandardScaler class, the features 'days_left' and 'duration'—whose values vary widely—are scaled.



```python
@staticmethod
def split(flight_price_df):

    x_train, x_test, y_train, y_test = train_test_split(
        np.array(flight_price_df.drop('price', axis=1)),
        np.array(flight_price_df['price']),
        shuffle=True,
        test_size=0.2
    )
    return x_train, x_test, y_train, y_test
```

In the **split** function, all features (all columns except 'price') are placed in **x**, and the predictive value (the 'price' column) is placed in **y**. Subsequently, using the **train_test_split** function, the data is shuffled and divided into an 80-20 split for training and testing, respectively. The model is then trained on 80% of the data and tested on the remaining 20%.



```python
@staticmethod
def gradient_descent(X, y, w_in, b_in, alpha, num_iters):
    w_history = []
    J_history = []
    w = w_in.copy()
    b = b_in

    tmp_num_iters = 20100
    converged = False
    for i in tqdm(range(tmp_num_iters), desc='Training', unit='record'):
        if converged:
            break

        dj_db, dj_dw = MyML.compute_gradient(X, y, w, b)

        w -= alpha * dj_dw
        b -= alpha * dj_db

        if i < 100000:
            J_history.append(MyML.compute_cost(X, y, w, b))
        w_history.append(w)

        converged_tmp = J_history[-1]

        if len(J_history) >= 10000 and all(abs(converged_tmp - J) < 10000 for J in J_history[-2:-101:-1]):
            converged = True


    return w, b, J_history
```

The **gradient_descent** function is the same function invoked by our program. In this function, vectors **x** and **y**, representing the training data, are passed. Additionally, the initial weight vector **w**, initial scalar **b**, and the learning rate (which will be explained shortly) are passed, along with the total number of data points.

In this function, for a specified number of iterations, here taken as 20100, and with the convergence condition of the cost function not converging, the gradient descent algorithm is executed to iteratively update the weight vector **w** and scalar **b**. This process aims to find optimal values for **w** and **b** that minimize the model's error.

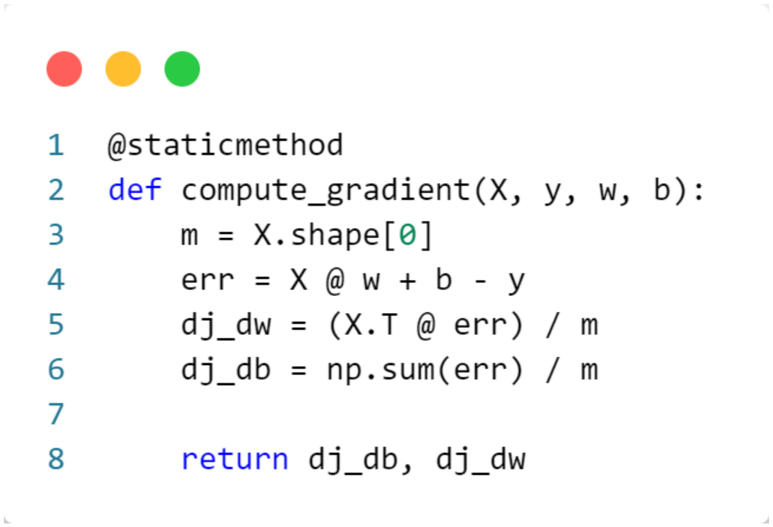
Note that in this function, the **tqdm** library is used to display a progress bar during the process.

The gradients of the weight vector **w** and scalar **b** are computed in the **compute_gradient** function, which will be explained later. These gradients are used to update the weight vector **w** and scalar **b**.

The cost of the model, calculated in the **compute_cost** function, is accumulated in the cost history during each iteration. This history allows for plotting the cost function later.

Based on the values of the cost function, if the difference between the last 100 cost in the history and the current cost in the iteration is less than 10000, it indicates that the cost function has approximately converged, and the iteration should be stopped.

Finally, the optimal weight vector **w**, scalar **b**, and the cost history are returned by the function.



```python
@staticmethod
def compute_gradient(X, y, w, b):
    m = X.shape[0]
    err = X @ w + b - y
    dj_dw = (X.T @ err) / m
    dj_db = np.sum(err) / m

    return dj_db, dj_dw
```

Based on the gradient formula, the **compute_cost** function initially calculates the model error on vectors **x** and **y** using the obtained weight vector **w** and scalar **b**. This error represents the difference between the predicted values and the actual values. The function then transposes the vector **x** and multiplies it by the error, dividing the result by the total number of features to calculate the gradient vector **w**.

Similarly, the function divides the sum of error vector elements by the number of features to calculate the gradient of the scalar **b**. These gradients are then returned by the function.

It's worth mentioning that all vectors and related functions are implemented using the numpy library.

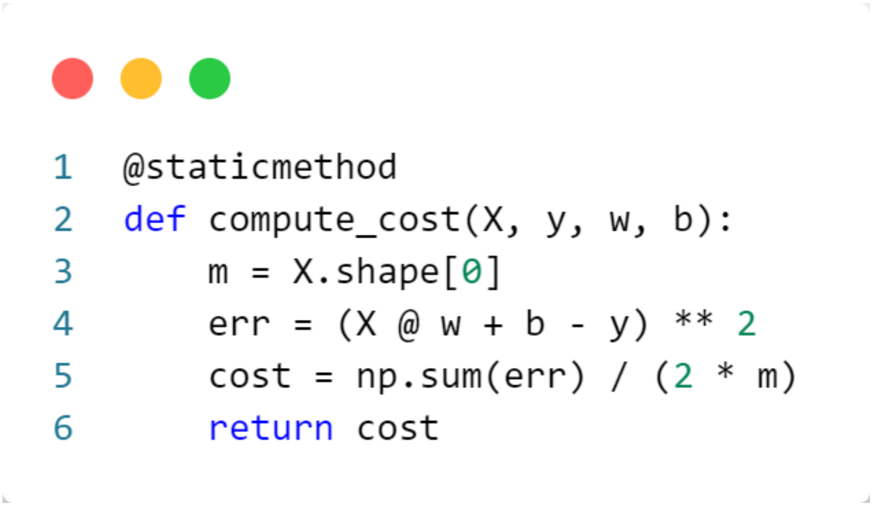
$$w_j = w_j - \alpha \frac{\partial J(w,b)}{\partial w_j}$$

$$b = b - \alpha \frac{\partial J(w,b)}{\partial b}$$

In a manner such that:

$$\frac{\partial J(w,b)}{\partial w_j} = \frac{1}{m} \sum_{i=0}^{m-1} \left(f_{w,b}\left(x^{(i)}\right) - y^i\right) x_j^{(i)}$$

$$\frac{\partial J(w,b)}{\partial b} = \frac{1}{m} \sum_{i=0}^{m-1} \left(f_{w,b}\left(x^{(i)}\right) - y^i\right)$$



```
@staticmethod
def compute_cost(X, y, w, b):
    m = X.shape[0]
    err = (X @ w + b - y) ** 2
    cost = np.sum(err) / (2 * m)
    return cost
```

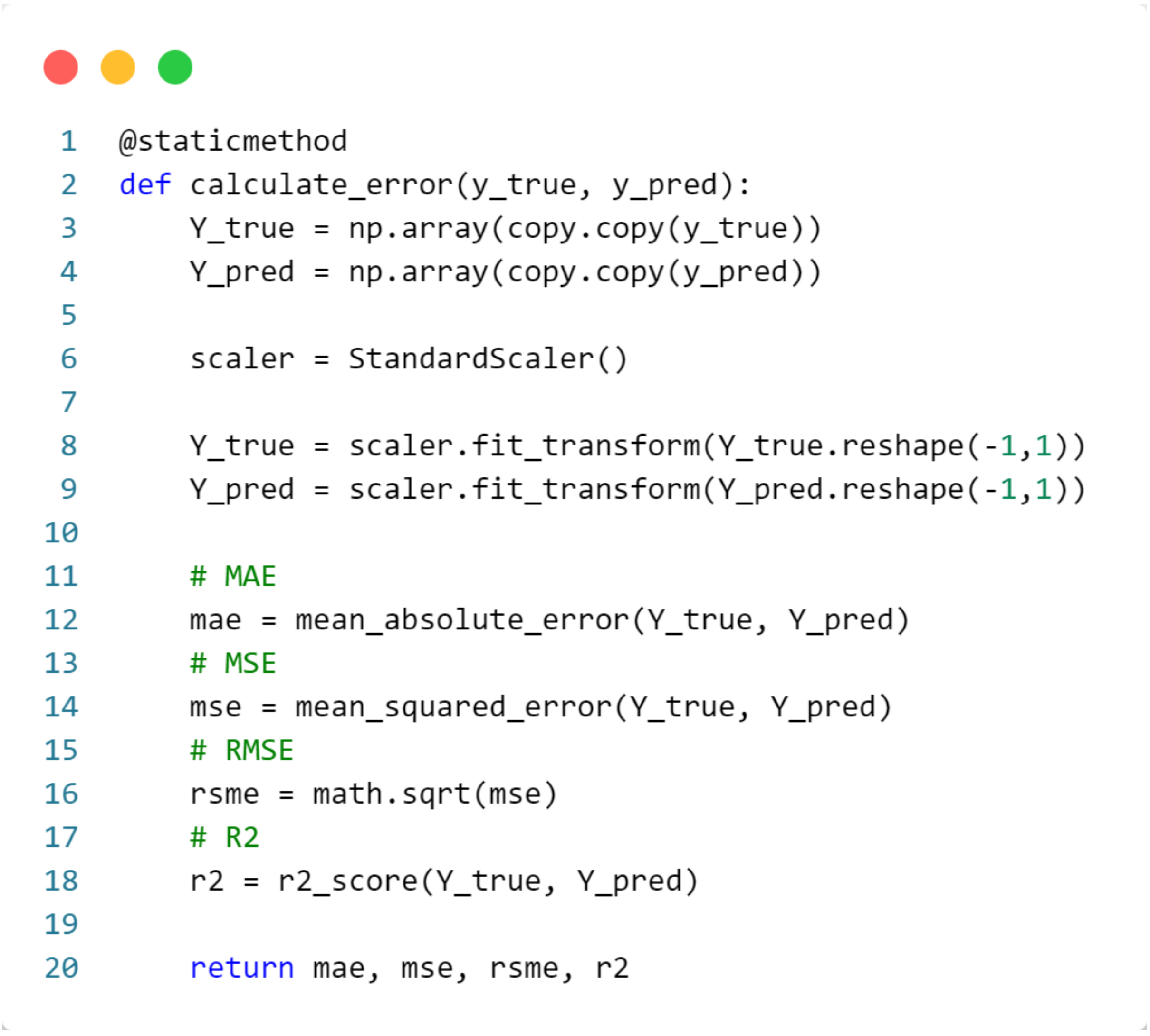
The **compute_cost** function, when given vectors **x**, **y**, and parameters **w** and **b**, calculates the model error based on Mean Squared Error (MSE). In essence, this function returns the average of the sum of squared errors.

In the above code, the **@** operator denotes the dot product.

$$J(w,b) = \frac{1}{2m} \sum_{i=0}^{m-1} \left(f_{w,b}\left(x^{(i)}\right) - y^i\right)^2$$

$$f_{w,b}\left(x^{(i)}\right) = w \,.\, x^{(i)} + b$$

It should be noted that we use 2m instead of m to calculate the average in the cost function, allowing the number 2 to cancel out during the gradient calculation.



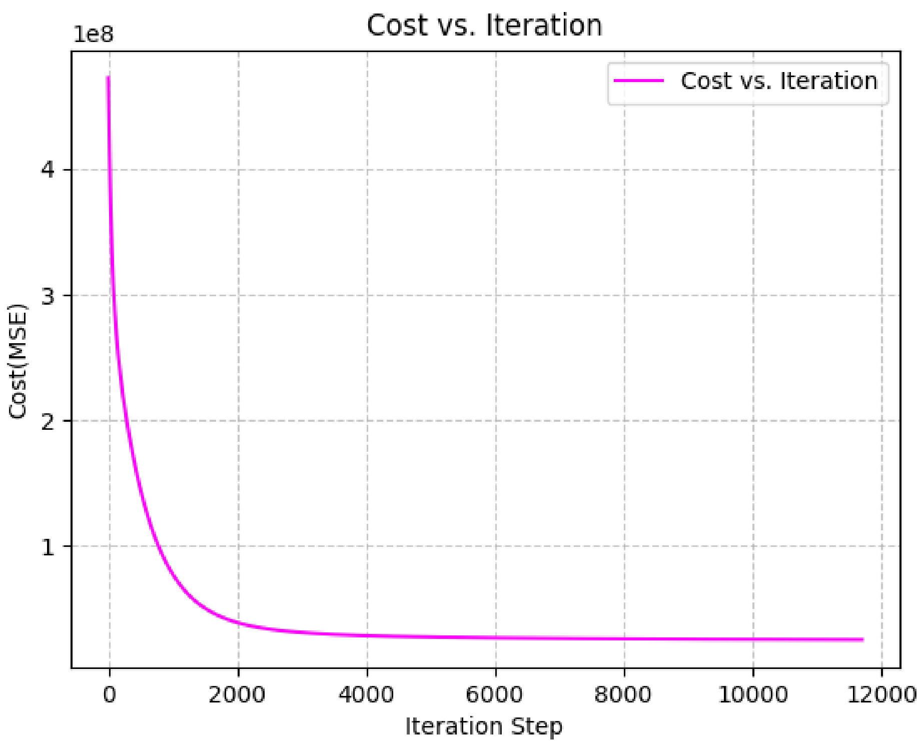
```python
@staticmethod
def calculate_error(y_true, y_pred):
    Y_true = np.array(copy.copy(y_true))
    Y_pred = np.array(copy.copy(y_pred))

    scaler = StandardScaler()

    Y_true = scaler.fit_transform(Y_true.reshape(-1,1))
    Y_pred = scaler.fit_transform(Y_pred.reshape(-1,1))

    # MAE
    mae = mean_absolute_error(Y_true, Y_pred)
    # MSE
    mse = mean_squared_error(Y_true, Y_pred)
    # RMSE
    rsme = math.sqrt(mse)
    # R2
    r2 = r2_score(Y_true, Y_pred)

    return mae, mse, rsme, r2
```

In the **calculate_error** function, for a better assessment of the model's performance, the vectors **y** and **ŷ**, representing the actual and predicted values, respectively, are first scaled on the test set. Subsequently, the Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-squared (R2) errors are calculated using the sklearn library and returned.

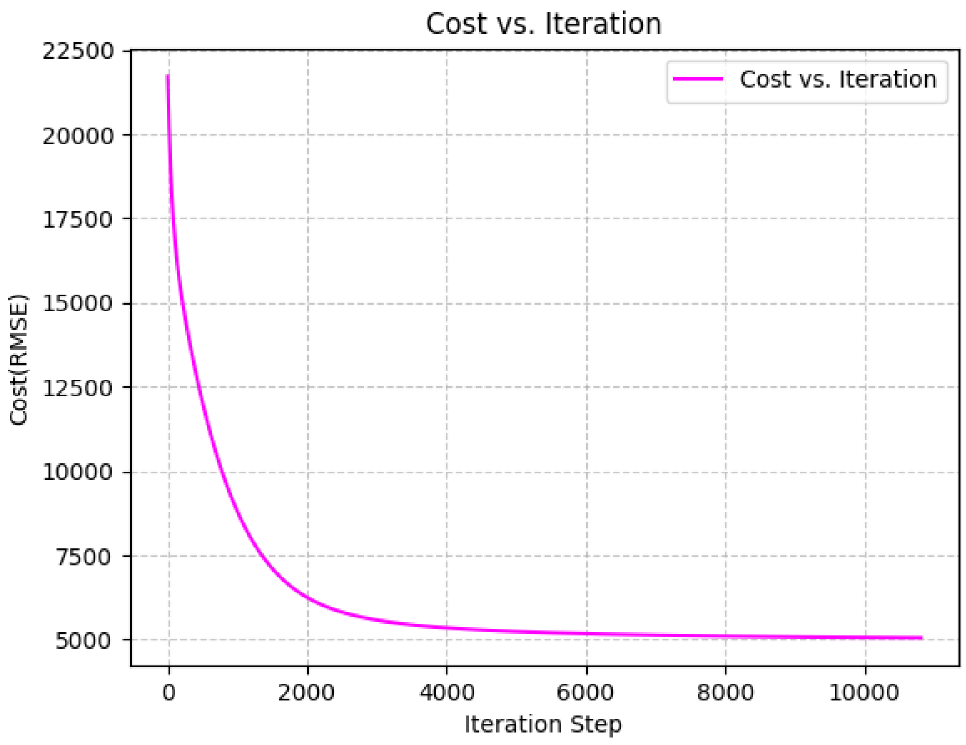
To initiate the model training, based on our investigations on the cost function, the learning rate is set to 0.006. Additionally, both the initial weight vector **w** and the initial scalar **b** are initialized to zero since the goal is to minimize the cost function.

Subsequently, a plot of the cost function's status based on MSE against the number of iterations is generated. The final model, the execution time of the training, and the cost function status based on various methods are saved in a different directory.

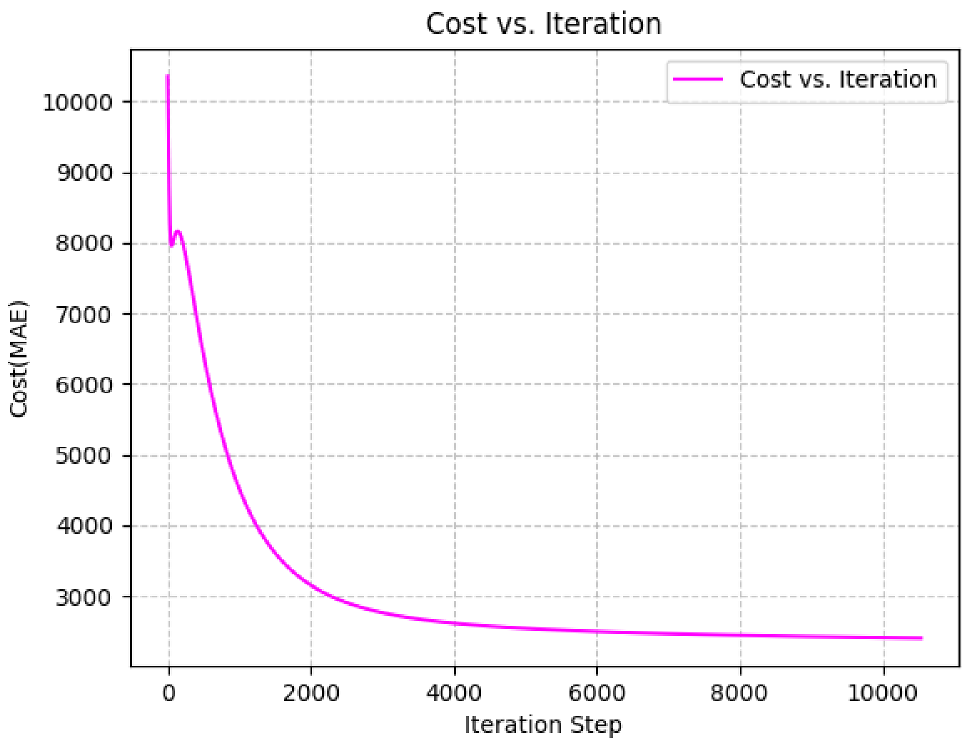
Cost Plots:



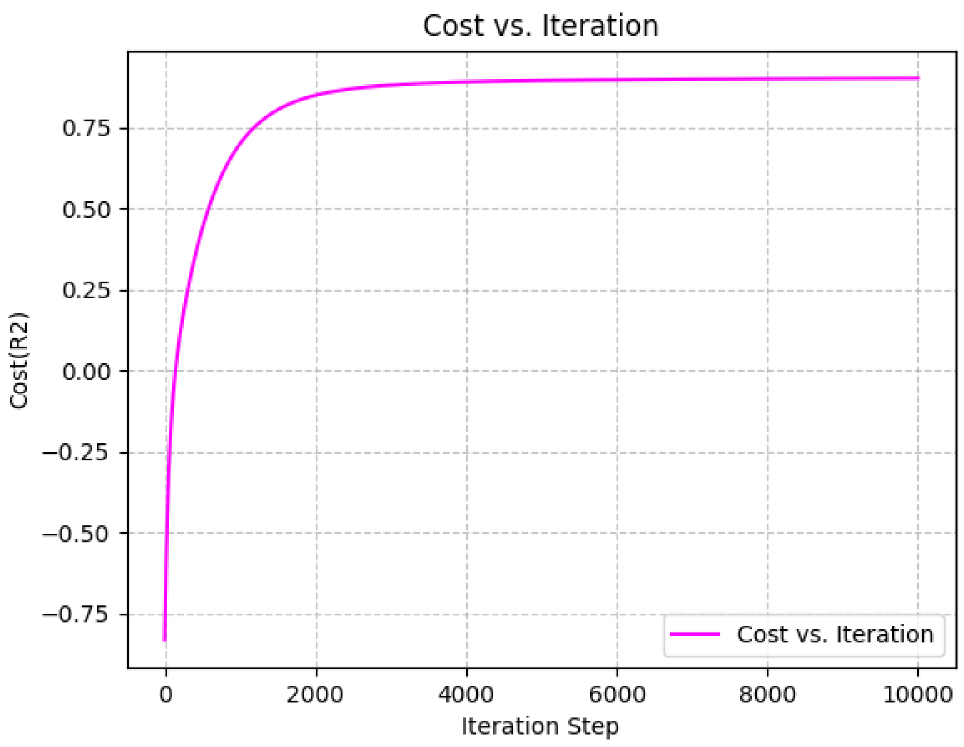
The plot of the cost function based on Mean Squared Error (MSE) with respect to the number of iterations.



The plot of the cost function based on Root Mean Squared Error (RMSE) with respect to the number of iterations.



The plot of the cost function based on Mean Absolute Error (MAE) with respect to the number of iterations.



The plot of the cost function based on R-squared (R2) with respect to the number of iterations.

Usage

Script Usage:

1. Clone the Repository:

```
git clone https://github.com/UIAI-4021/search-pandas.git
cd scripts
```

2. Install Dependencies:

```
pip install -r requirements.txt
```

3. Run the Script:

```
Python run_multiple_regression.py
```

Example Usage:

```
python run_multiple_regression.py
```

References

- Supervised Machine Learning: Regression and Classification Course by DeepLearning.AI & Stanford University
- OpenAI. "ChatGPT." https://www.openai.com/
- GeeksForGeeks. https://www.geeksforgeeks.org/
- Stack Overflow. https://stackoverflow.com/
- W3Schools. https://www.w3schools.com/
- Russell, Stuart, and Norvig, Peter. "Artificial Intelligence: A Modern Approach." (Book)