Indian Institute of Information Technology
Senapati, Manipur

Machine Learning (CS306) Assignment

Air Quality Prediction Using Linear Regression

Annepu Sai Charan
Computer Science and Engineering
20010132

Overview

This is an “Air Quality Prediction” model implemented using “Multivariate Linear Regression”. The “AirQualityUCI” dataset has been used. The model is implemented using scratch and compared with the builtin model of sklearn library.

Theory:

The linear regression algorithm is used when there is a linear relationship between input and output. The data with only one independent variable (one input variable) we call *“Univariate linear regression”* and with multiple variables it is called *“Multivariate linear regression”*. Linear regression is a **supervised learning** algorithm.

General representation of multivariate linear regression:

Y = b + a1x1 + a2x2 + a3x3 + … + amxm.
Y -> label (dependent variable)
x1–xm -> parameters (independent variable)
b -> bias

The below shown formula is the cost function of linear regression, our goal is to minimize the cost value, which eventually calculates the weight values.

Cost Function

$$J\left(\Theta_0, \Theta_1\right) = \frac{1}{2m}\sum_{i=1}^{m}[\underset{\text{Predicted Value}}{\underset{\uparrow}{h_\Theta(x_i)}} - \underset{\text{True Value}}{\underset{\uparrow}{y_i}}]^2$$

The below formula is the gradient descent formula which calculates the weight by minimizing the cost function.

Gradient Descent

$$\Theta_j = \Theta_j - \underset{\substack{\uparrow \\ \text{Learning Rate}}}{\alpha} \frac{\partial}{\partial \Theta_j} J\left(\Theta_0, \Theta_1\right)$$

Libraries used:

These are the following libraries used in this project:

- Pandas (for data preprocessing)
- Numpy (for mathematical computations)
- Matplotlib (for visualizing learning curve)
- Sklearn (splitting data)

Implementation:

.

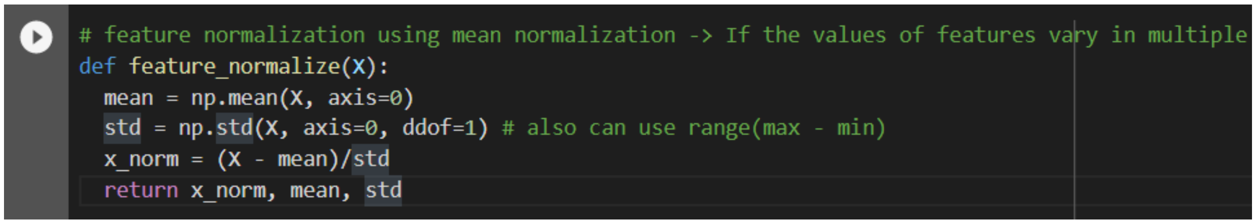
Following are the details about the implementation of the model.

Preprocessing

The dataset is first imported and following preprocessing is done:

- The null rows and columns are dropped from the dataset.
- The floating point numbers are formatted in standard form.
- The data and time field is dropped because those will not have any effect in predicting the output.
- Then the data splitted into a train and test set. The 60% of data goes for the train and the rest for the test set.
- The labels are separated from the actual data.

Few of the fields in the dataset vary in multiples which will have a negative impact on gradient descent function so, the **feature normalization** is performed. There are two types of feature normalization, feature scaling and mean normalization. The ***mean normalization*** is used in this implementation.
In mean normalization the average value is subtracted from the input and divided by the range(max-min) or standard deviation.

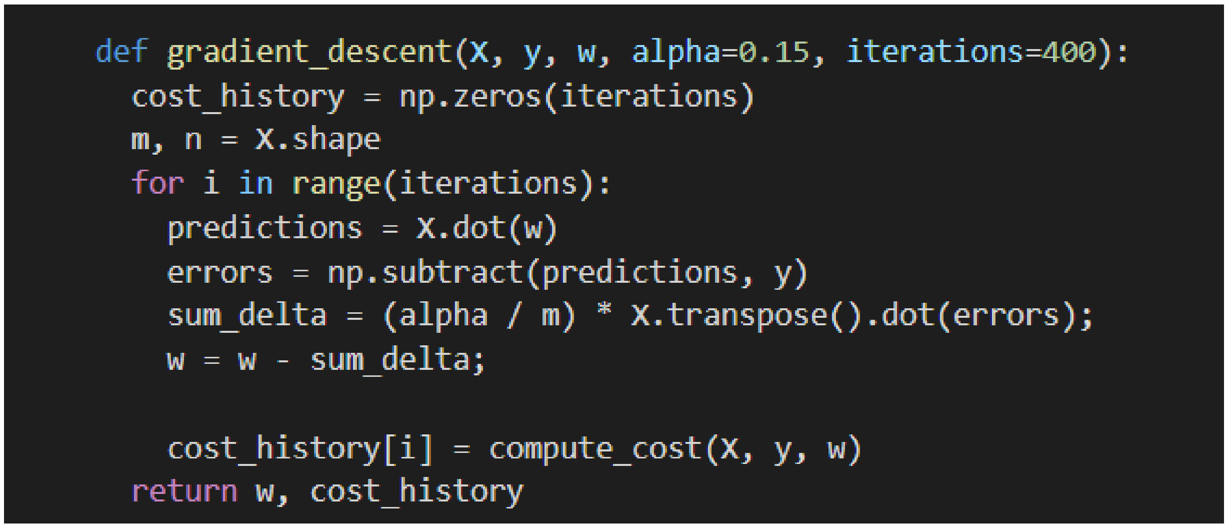


```
# feature normalization using mean normalization -> If the values of features vary in multiple
def feature_normalize(X):
  mean = np.mean(X, axis=0)
  std = np.std(X, axis=0, ddof=1) # also can use range(max - min)
  x_norm = (X - mean)/std
  return x_norm, mean, std
```

The above figure shows the feature normalization function.
The new column is added to the dataset and with this our new dataset is formed.

Algorithm implementation

This implementation has mainly two functions, one is for gradient descent and other for cost computation.

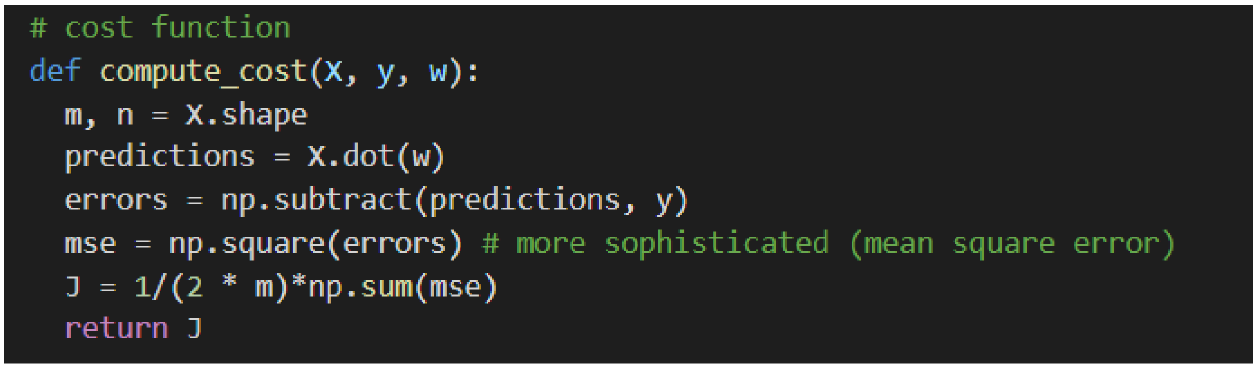


```
def gradient_descent(X, y, w, alpha=0.15, iterations=400):
  cost_history = np.zeros(iterations)
  m, n = X.shape
  for i in range(iterations):
    predictions = X.dot(w)
    errors = np.subtract(predictions, y)
    sum_delta = (alpha / m) * X.transpose().dot(errors);
    w = w - sum_delta;

    cost_history[i] = compute_cost(X, y, w)
  return w, cost_history
```

The above figure shows the gradient descent algorithm implementation.

The gradient descent algorithm will help us to find the final weights of the parameters.
The gradient descent algorithm has **step sizes of “0.15” and “400” iterations**. It takes the random weights for the first time and calculates the cost, then it updates the weight inorder to minimize the cost and eventually calculates the optimum weights.
The *“cost_history”* will be used for visualizing learning curves.



```python
# cost function
def compute_cost(X, y, w):
  m, n = X.shape
  predictions = X.dot(w)
  errors = np.subtract(predictions, y)
  mse = np.square(errors) # more sophisticated (mean square error)
  J = 1/(2 * m)*np.sum(mse)
  return J
```

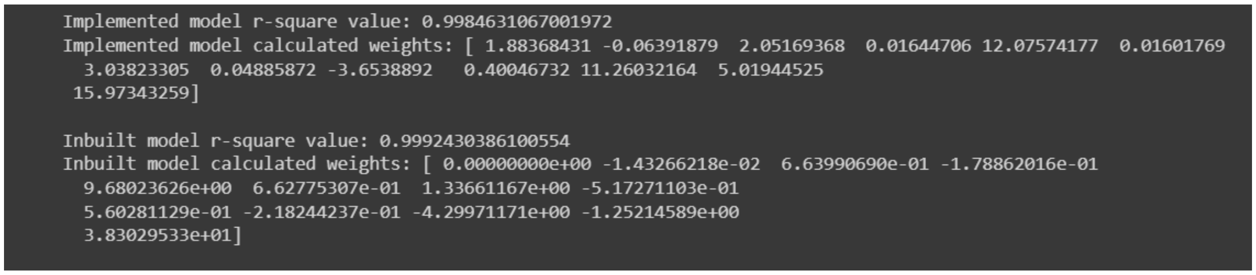
The above figure shows the implementation of the *“compute_cost”* function.

This function predicts the output and calculates the error by subtracting it from the actual values. The **“mean square error”** is used since it is more sophisticated. It calculates the cost and returns it which is then used by the gradient descent algorithm.

<u>Result:</u>

It is observed that the algorithm implemented from scratch performs very well. It is compared with the sklearn builtin linear regression library.

The following are the observations:
The r-square and weight values:

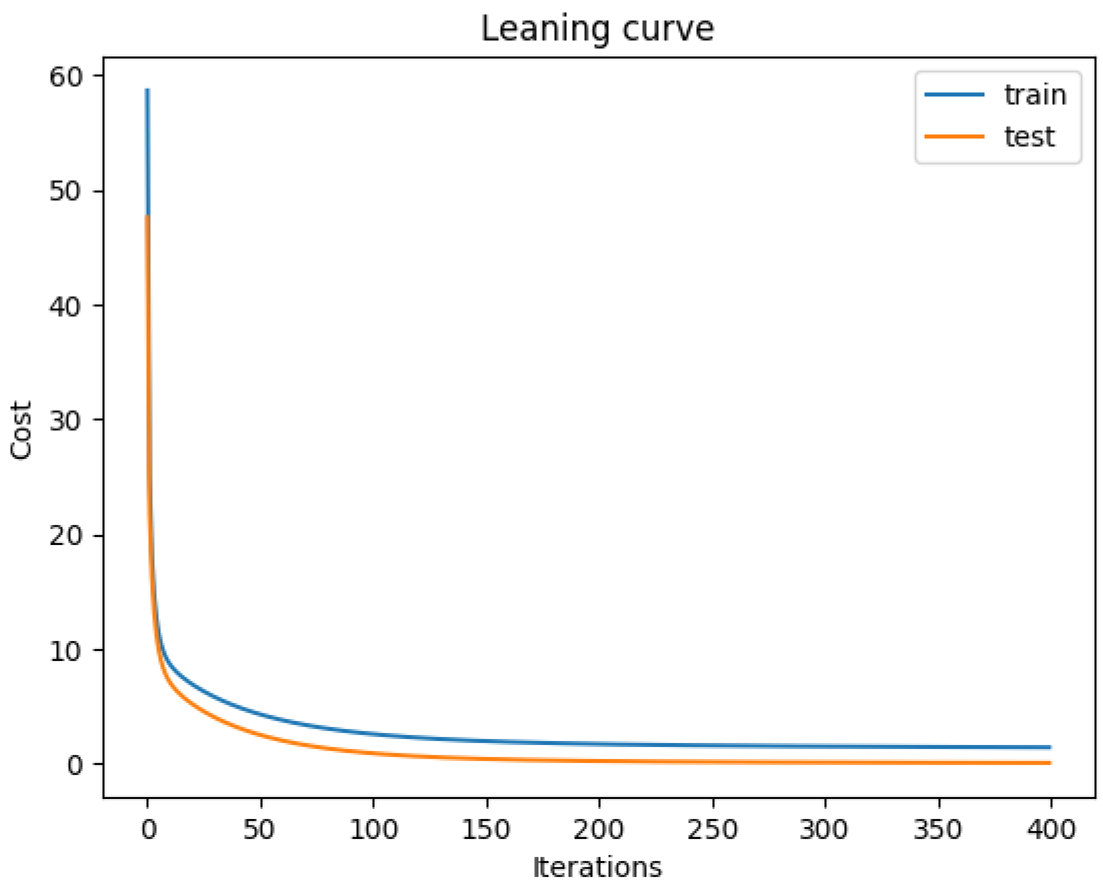


```
Implemented model r-square value: 0.9984631067001972
Implemented model calculated weights: [ 1.88368431 -0.06391879  2.05169368  0.01644706 12.07574177  0.01601769
  3.03823305  0.04885872 -3.6538892   0.40046732 11.26032164  5.01944525
 15.97343259]

Inbuilt model r-square value: 0.9992430386100554
Inbuilt model calculated weights: [ 0.00000000e+00 -1.43266218e-02  6.63990690e-01 -1.78862016e-01
  9.68023626e+00  6.62775307e-01  1.33661167e+00 -5.17271103e-01
  5.60281129e-01 -2.18244237e-01 -4.29971171e+00 -1.25214589e+00
  3.83029533e+01]
```

The implemented model has an r-square value of **“0.998”** and the inbuilt model has an r-square value of **“0.999”.**

The learning curve is shown below:


Leaning curve
train
test
Cost
Iterations