Linear Regression - 2

Gradient Descent Revision

Let's take a f^n :

$f(x) = x^2$

Steps of GD

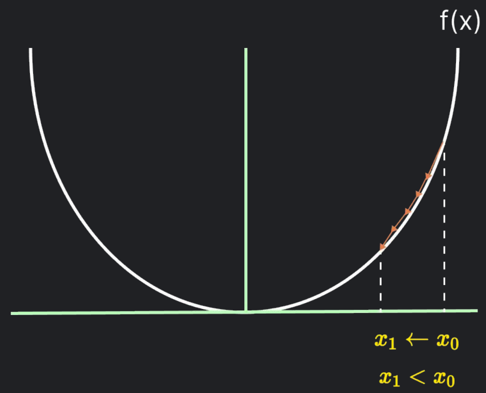
1. Pick x_o randomly

2. $\left.\frac{df}{dx}\right|_{x_o}$ = +ve [inc . f^n]

3. $x_1 = x_0 + \eta(\frac{-\partial f}{\partial x})_{x=x_0}$

Learning rate

Why negative?



- Derivative positive at x0 since function is increasing
- -ve because to move in opposite direction to reach minima
- $x_o \longrightarrow x_1$ Moves towards pt closer to minima

Quiz - 4

Why GD?

To optimize the loss fn and find its minima

Loss $f^n \longrightarrow$ MSE

Minimize $$\frac{1}{m}\sum_{i=0}^{m}(\hat{y}^{(i)} - y^{(i)})^2$$

Linear Regression Helper Functions

—>>> We define our linear regression class and set LR & no. of iterations

```
import numpy as np
class LinearRegression() :
  def __init__(self, learning_rate=0.01, iterations=5):
    self.learning_rate = learning_rate
    self.iterations = iterations
```

—>>> Next we define our predicted Fn

Remember : $\hat{y} = w_t\, x + w_0$ → We will call this as bias

```
def predict(self, X):
  return np.dot(X, self.W)+self.b

LinearRegression.predict=predict
```

Note:

We are using this notation of adding a function, as seen from the last line LinearRegression.predict=predict, since we aren't defining the class in one go

—->>> Finally we evaluate our evaluation metric $R^2 Score$

```
def r2_score(self, X, y):
  y_ = predict(self,X)
  ss_res = np.sum((y-y_)**2)
  ss_tot = np.sum((y- y.mean())**2)
  score = (1- ss_res/ss_tot)
  return score

LinearRegression.score=r2_score
```

How to learn Weight Optimization?

Given :

$$D = \{(x^{(i)}, y^{(i)})_{i=1}^{i=n}; x^{(i)} \in R^d, y^{(i)} \in R\}$$

Objective :

$$Find\ w\ s.t.,$$

$$\hat{y}^{(i)} = f(x^{(i)}) = w^T x^{(i)} + w_0, \forall i : 1- > m$$

$$w^T = [w_1, w_2, w_3, \ldots, w_d]$$

Calculate :

$$\underbrace{\hat{y}}_{\text{Predicted}} \approx \underbrace{y}_{\text{True}}$$

That is,

$$\frac{1}{m}\sum_{i=1}^{m} \underbrace{\hat{y}^{(i)} - y^{(i)}}_{\text{ERROR}}$$

Minimize

How to minimize error?

By minimizing the Loss f^n ⟶ MSE

Loss f^n ??

Remember MSE/ MAE?

So, **MSE** or **MAE**

WHY ??

Therefore, Loss f^n = MSE ⟶ Minimize

MAE vs MSE as cost function

MAE gives a +1 ,-1

when gradient

$$\frac{dMAE}{dy_{pred}} = \begin{cases} +1 & y_{pred} > y_{true} \\ -1 & y_{pred} < y_{true} \end{cases}$$

MAE can be a valid cost function as

if you are predicting too high ($ypred$ > $ytrue$) ,

then increasing $ypred$ yet more by one unit will increase the MAE by an equal amount of one unit,

so the gradient encourages you to reduce $ypred$. And vice versa if $ypred$ < $ytrue$

But MAE is not used for Linear Regression **MAE isn't differentiable at** $y_{true} = y_{pred}$	While MSE is differentiable as well as increasing function is preferred in LR We will see its calculations in some time

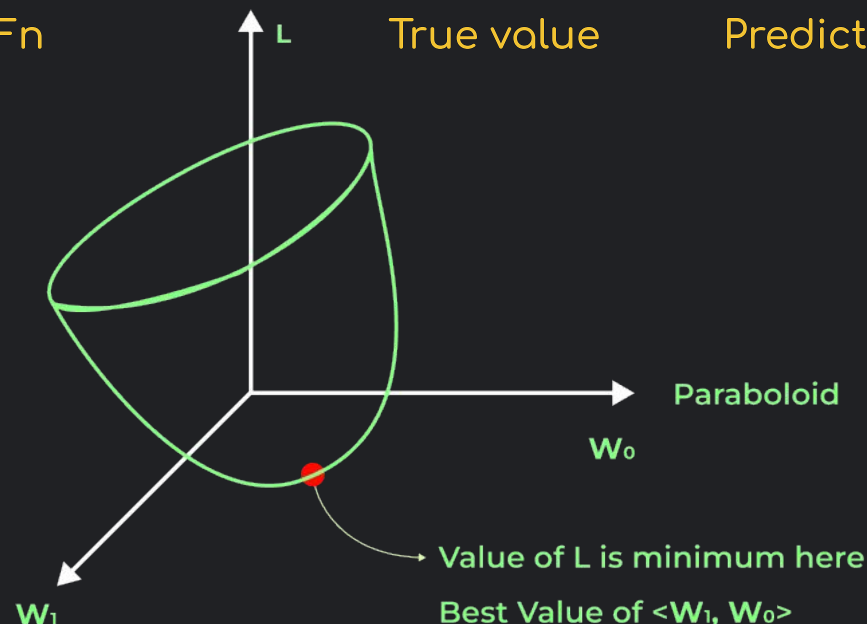
EXAMPLE of a single predictor

$$L = \min_{w_0, w_1} \frac{1}{m} \sum_{i=1}^{m} [y^{(i)} - (w_0 + w_1 x^{(i)})]^2$$

Loss Fn

True value

Predicted value



How will L change for multiple predictors ??

$$L = \min_{w_0, w_1, \ldots, w_d} \frac{1}{m} \sum_{i=1}^{m} [y^{(i)} - (w_0 + w_1 x_1^{(i)} + w_2 x_2^{(i)} + \ldots + w_d x_d^{(i)})]^2$$

Geometrically, A hyperplane in df 1 dimensions

Global minima : Best value of < w_1 , w_0 >

How to find global minima?

Using Gradient Descent

We revised gradient descent at the start of this lecture

To do Gradient Descent of L , we use partial derivatives

$$L \text{ is a } \underbrace{F^n \text{ of } (w_0, w_1, w_2 \ldots\ldots\ldots\ldots w_d)}_{\text{Quadratic Function}}$$

Optimization

Take a partial derivative of L w.r.t each variable

$$\frac{\partial L}{\partial w_0} = 0$$

$$\frac{\partial L}{\partial w_1} = 0$$

$$\cdots$$

$$\frac{\partial L}{\partial w_d} = 0$$

System of Linear Equations

L is Quad

Derivative will be linear

Let’s take $< w_1, w_2, w_0 >$ for simplicity

Calculate:

$$\frac{\partial L}{\partial w_0} = 0, \frac{\partial L}{\partial w_1} = 0 , \frac{\partial L}{\partial w_2} = 0$$

$$L(w_2, w_1, w_0) = (y - (w_d x_2 + w_1 x_1 + w_0)^2$$

constant

$\hat{y}$

$$\frac{\partial L}{\partial w_0} = \frac{\partial (y - (w_2 x_2 + w_1 x_1 + w_0))^2}{\partial w_0}$$

$$= 2(y - \hat{y}) . \frac{\partial(-\hat{y})}{\partial w_0}$$

= -1

$$= -2(y - \hat{y})$$

= error

-2 error

constant

$$\frac{\partial L}{\partial w_1} = \frac{\partial (y - (w_2 x_2 + w_1 x_1 + w_0))^2}{\partial w_1}$$

$$= 2(y - \hat{y}) . \frac{\partial(-\hat{y})}{\partial w_1}$$

$$= 2(y - \hat{y}) . -x_1$$

$$= -2(y - \hat{y}) . x_1$$

-2 error. Input value

constant

$$\frac{\partial L}{\partial w_2} = \frac{\partial(y-(w_2x_2+w_1x_1+w_0))^2}{\partial w_2}$$

$$= 2(y-\hat{y}).\frac{\partial(-\hat{y})}{\partial w_2}$$

$$= 2(y-\hat{y}).-x_2$$

$$= -2(y-\hat{y}).x_2$$

-2 error. Input value

Notice a pattern?

The partial derivative

-2 . Error . Input value of wt.

For m points

$$\frac{\partial L}{\partial w_0} = \frac{1}{m}\sum_{i=1}^{m} -2(y-\hat{y})$$

$$\frac{\partial L}{\partial w_d} = \frac{1}{m}\sum_{i=1}^{m} -2(y-\hat{y})x_d$$

How to update the weights ??

$$w_0 = w0 - \alpha \frac{\partial L}{\partial w_0}$$

$$\vdots$$

$$w_d = wd - \alpha \frac{\partial L}{\partial w_d}$$

Learning rate ⇒ How big of a step you want to take?

How to decide the right Linear Regression ??

Takes too long to reach downhill

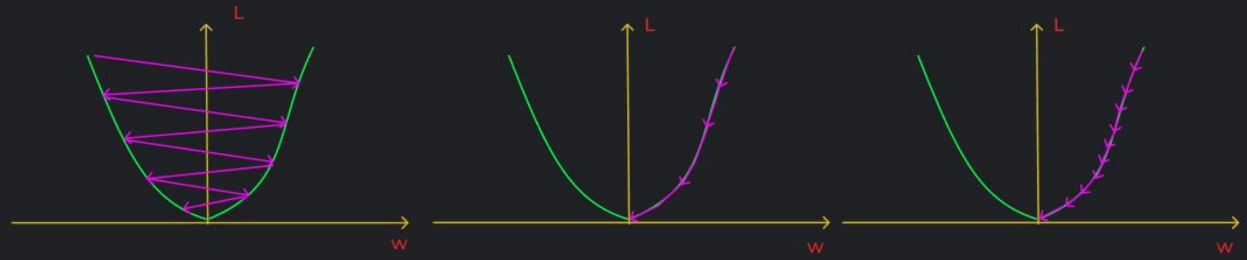
ADVICE $\Rightarrow$

Start with smaller α

If model is trained, increase α

How to decide the right Linear Regression ??

Run (α = 1) Jog(α = 0.1) Walk(α = 0.01)



Takes too long to reach downhill

ADVICE $\Rightarrow$

Start with smaller $\boxed{\alpha}$

If model is trained, increase $\boxed{\alpha}$

Implementation of Wt.s update and GD

Weights update

```
[ ] def update_weights(self):
      Y_pred = self.predict( self.X )
      # calculate gradients
      dW = - (2*(self.X.T ).dot(self.Y - Y_pred))/self.m
      db = - 2*np.sum(self.Y - Y_pred)/self.m
      # update weights
      self.W = self.W - self.learning_rate * dW
      self.b = self.b - self.learning_rate * db
      return self

    LinearRegression.update_weights=update_weights
```

Note:

Using 1 point of updation is called stochastic gradient descent.

Whereas if we update the weights after m points, it's called batch gradient descent.

Define the fit fn

```python
def fit(self, X, Y):
  # no_of_training_examples, no_of_features
  self.m, self.d = X.shape
  # weight initialization
  self.W = np.zeros(self.d)
  self.b = 0
  self.X = X
  self.Y = Y
  self.error_list=[]
  # gradient descent learning
  for i in range(self.iterations):
    self.update_weights()
    Y_pred=X.dot(self.W)+self.b
    error=np.square(np.subtract(Y,Y_pred)).mean()
    self.error_list.append(error)
  return self

LinearRegression.fit=fit
```

Let's finally train our model

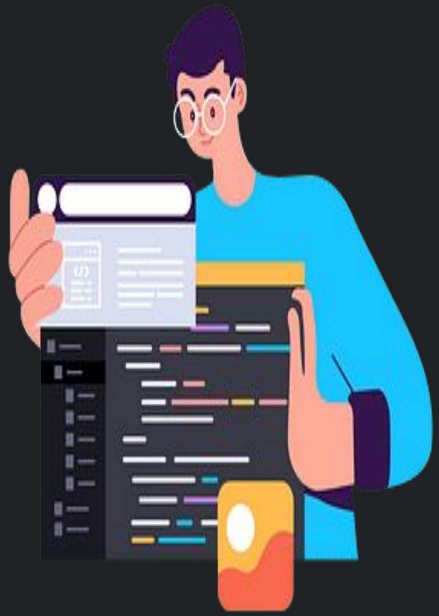
Importing necessary libraries

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

!gdown 1UpLnYA48Vy_lGUMMLG-uQElgf_Jel2Lh
```

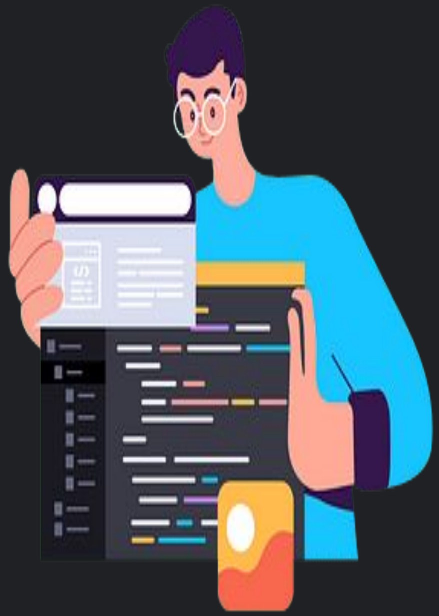
Reading the data set

```
df = pd.read_csv('cars24-car-price-clean.csv')
df.head()
```



Defining our model

```
lr = LinearRegression(iterations=100)
lr.fit(X_train, y_train)
```



Prediction on test data and evaluation

```
lr.predict(X_test)
```

```
array([-0.84308986, -0.27745439,  2.44849382, ...,  0.65540967,
       -0.54931517,  0.10246191])
```

```
lr.score(X_train, y_train)
```

```
0.9101033692790105
```

```
lr.score(X_test, y_test)
```

```
0.9075347558295035
```

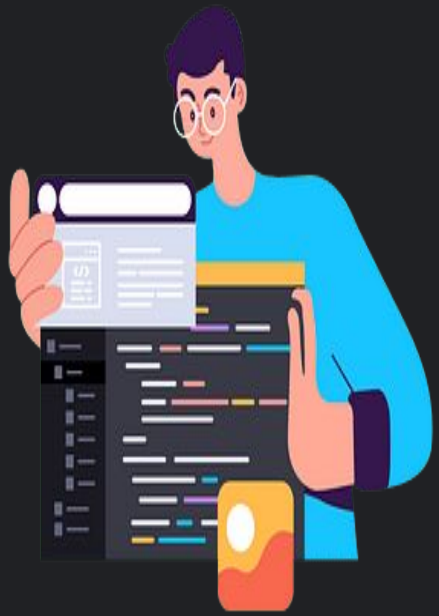
Wts and Bias

```
lr.W
```

```
year                 0.125722
km_driven           -0.047847
mileage             -0.050197
engine               0.093654
max_power            0.154132
age                 -0.125722
make                 0.189528
model                0.371064
Individual          -0.025478
Trustmark Dealer    -0.004997
Diesel               0.045118
Electric             0.016967
LPG                  0.002790
Petrol              -0.042832
Manual              -0.105770
5                   -0.005216
>5                   0.003182
dtype: float64
```

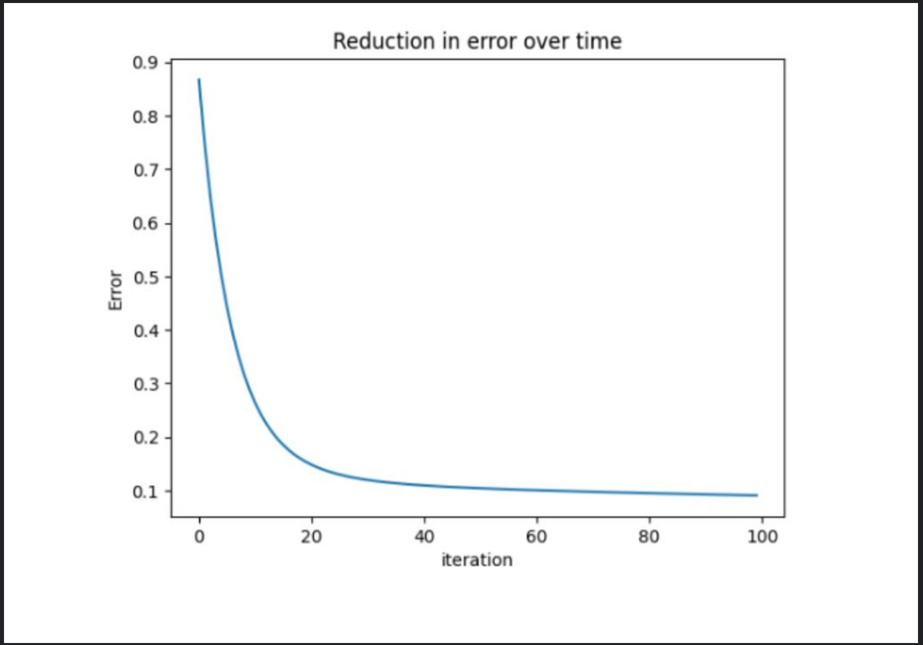
```
lr.b
```

```
0.0011317414350942184
```



Plotting the error vs time plot

```
%matplotlib inline
fig = plt.figure()
plt.plot(lr.error_list)
plt.title("Reduction in error over time")
plt.xlabel("iteration")
plt.ylabel("Error")
plt.show()
```

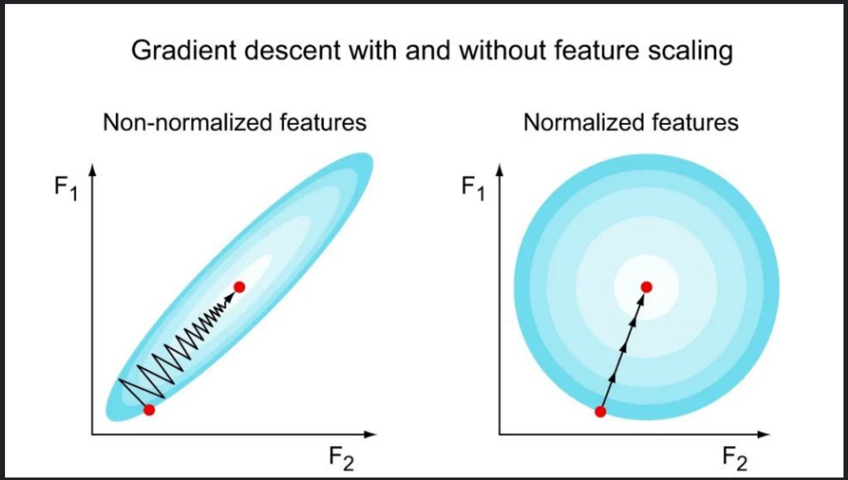


How does feature scaling help ??

Remember feature scaling helps in predicting the feature importance of predictors

1. Ensures all features are normalized and scaled equally
    - ML considers features with wide ranges to be important
      (It’s not how it should be)
    - Scaling ensures no feature dominates the training process

2. G.D converges faster



Problem with R- Square

What if we add one extra feature to our model

N features ⇒ R-Square

+

1 feature

$$W_0X_0 + W_1X_0 + W_2X_2 + \dots\dots\dots W_nX_n + W_{n+1}X_{n+1}$$

How will R-quare vary ?

CASE 1 Feature is relevant . R- Square increases (trivial)

CASE 2 Feature is not relevant . Will the R-squared decrease, increase or remain same?

R- square

↑ ✗ This is not fine

↓ ✗ Wn+d ⇒ 0 ensures that

No change — ✓ This is fine with us

When will R- square increase (irrelevant features) ?

Model makes some “spurious” assumptions

↓

Performance increase “by chance”

Adjusted R- Square

Solution ??

Add a penalty for increase in “d”

$$Adj.R-sq. = 1 - \left[\frac{(1-R^2)(m-1)}{(m-d-1)}\right]$$

Adjusted R- square has a quality

It decreases if dimension increases.

R2 won't increase.

Scenario 1

D increases : (m-d-1) decreases , R2 (no changes) → Adjusted R- Square decreases

(Minimal change in R2)

Scenario 2

D increases : (m-d-1) decreases , R2 (increase) → Adjusted R- Square increases

(notable change in R2)

```
y_hat = lr.predict(X_test)
Adj_R = 1 - (1-lr.score(X_test, y_test)*(len(y_test)-1)/(len(y_test)-X_test.shape[1]-1))
print("Adjusted R-squared:", Adj_R )
```

```
Adjusted R-squared: 0.9114445609103706
```