linear-regression-dl

March 12, 2024

1 Mathematical dive into Linear Regression using Gradient Descent

1.0.1 Author:

- `Sachin M Sabariram`
- Github
- Linkedin

In this Notebook we dive deep into the classical Linear Regression problem with the popular Boston housing dataset starting with the basic data analysis and solving it using various approaches mostly based on Single layer Perceptron approach and finally see it's similarity with Sklearn implementation of Linear Model.

1. Pure Mathematical Approach using Numpy (Here we compute the gradients of the weight and baises by partial derivates manually)
2. Using Pytorch tensors (Still step by step solving but using torch to do gradient heavy lifting for us)
3. Using Pytorch nn module (Here we go full library version to implement the linear regression with Pytorch libraries)
4. Finally we comapare the results with the classical Linear Regression model from the Sklearn Library

1.0.2 Data preprocessing

[1]:
```
import pandas as pd
import numpy as np
```

[2]:
```
df = pd.read_csv("./housing.csv")
```

[3]:
```
df.head()
```

[3]:
```
      RM  LSTAT  PTRATIO      MEDV
0  6.575   4.98     15.3  504000.0
1  6.421   9.14     17.8  453600.0
2  7.185   4.03     17.8  728700.0
3  6.998   2.94     18.7  701400.0
4  7.147   5.33     18.7  760200.0
```

```
[199]: df.isnull().sum()
```

```
[199]: RM         0
       LSTAT      0
       PTRATIO    0
       MEDV       0
       dtype: int64
```

```
[4]: cols = df.columns
     cols
```

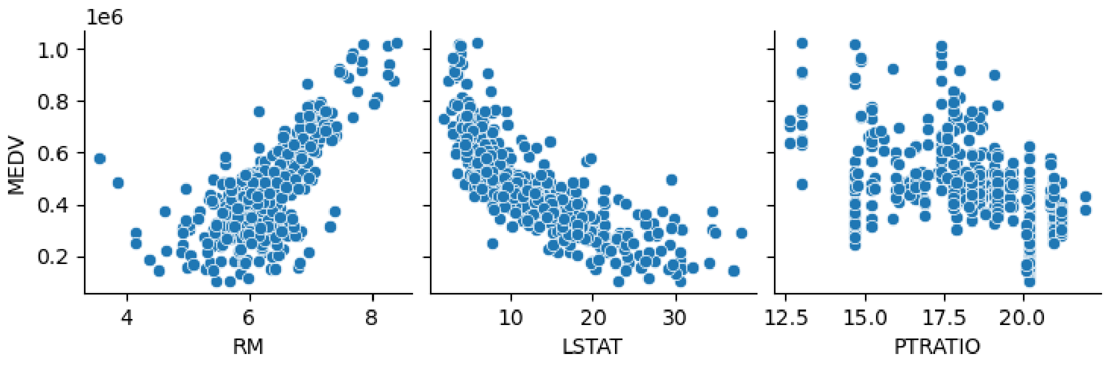
```
[4]: Index(['RM', 'LSTAT', 'PTRATIO', 'MEDV'], dtype='object')
```

```
[191]: import warnings
       warnings.filterwarnings('ignore')
```

```
[192]: import seaborn as sns
```

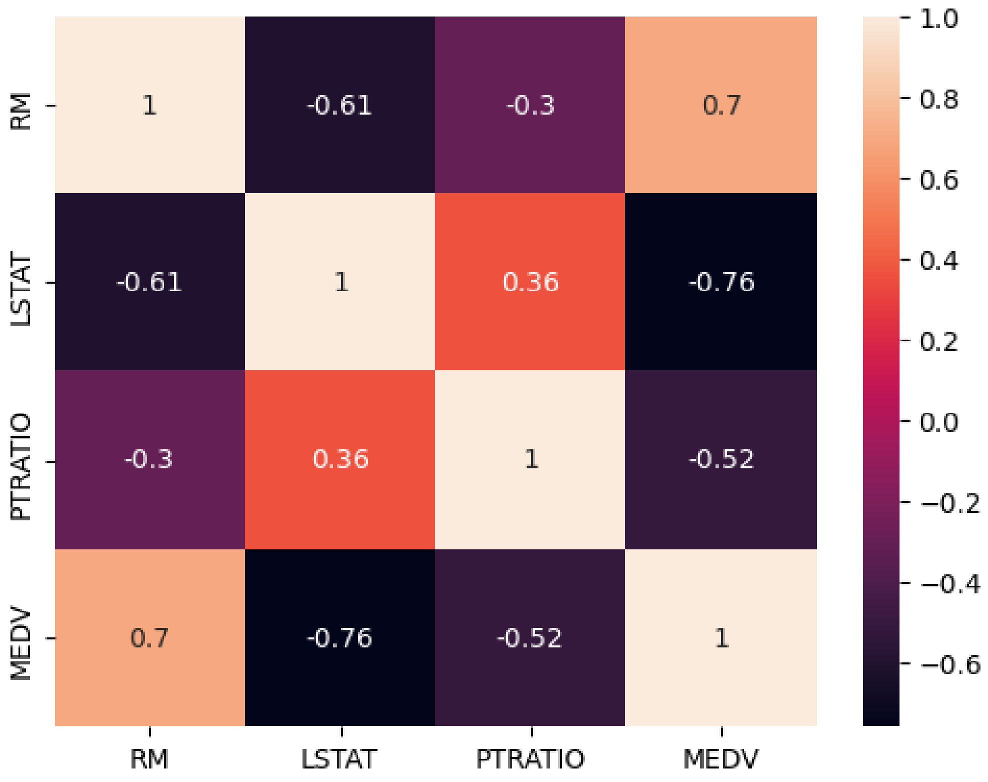
```
[6]: sns.pairplot(df, x_vars=cols[:3], y_vars=cols[-1])
```

```
[6]: <seaborn.axisgrid.PairGrid at 0x1463e5280>
```



```
[7]: sns.heatmap(df.corr(), annot=True)
```

```
[7]: <AxesSubplot:>
```



As for this example for the demostration of Multivariate Linear Regression we are gonna use all the 3 vaiables to predict 'MEDV'

1.1 Pure Mathematical Approach

As in this case of Boston Housing there are 3 independent variables ['RM', 'LSTAT', 'PTRATIO'] and 1 dependent ['MEDV'] variable

- Thus Y = X1.W1 * X2.W2 * X3.W3 + b

1. X -> n rows x 3 cols
2. Y -> n rows x 1 cols
3. W -> 1 rows x 3cols
4. b -> 1 (rows,cols)

Converting the dataframe to Numpy Arrays

```
[206]: x = df.iloc[:,:3]
       x = x.to_numpy()
       x.shape
```

```
[206]: (489, 3)
```

```
[207]: y = df.iloc[:,-1]
       y = y.to_numpy().reshape(489,1) #To match the output of y
       y.shape
```

```
[207]: (489, 1)
```

Initialising the Parameters (W,b) to random values

```
[267]: w = np.random.randn(1,3)
       b = np.random.randn(1)
       params = {'w':w, 'b':b}
       params
```

```
[267]: {'w': array([[-0.33398227,  0.12529013,  0.82622773]]),
        'b': array([-0.76626045])}
```

Modeling the function Y = X.W + b

```
[209]: def forward(x,params):
           w,b = params['w'], params['b']
           return x@w.T + b
```

```
[210]: pred = forward(x, params)
```

```
[211]: pred.shape
```

```
[211]: (489, 1)
```

Function to compute the loss (MSE)

```
[212]: def loss(pred,y):
           return np.sum(np.square(y-pred))/2*len(y)
```

```
[213]: loss(pred,y)
```

```
[213]: 2.794247326159067e+16
```

Partial derivatives of the Loss function

- Loss(l) = (y-pred)^2/2n

Partial derivatives with respect to model parameters

1. dl/dw = -1/n * (y-pred)*(X)
2. dl/db = -1/n * (y-pred)

Shape of y = n x 1 Shape of x = n x 3

[219]:
```
def grad(x,y,pred):
    l = y-pred
    #print(l.shape,x.shape)
    def dw():
        return -(1/len(y)) * (l.T@x)
    def db():
        return np.sum(-(1/len(y))*l)
    dw, db = dw(),db()
    return {"dw":dw, "db":db}
```

[220]:
```
grad(x,y,pred)
```

[220]:
```
{'dw': array([[-2909282.14153843, -4990157.23198805, -8232071.78530779]]),
 'db': -454343.69417215}
```

Implementing backpropagation Adjusting the parameters in the direction opposite to the gradient of the loss function in order to minimise the loss

[222]:
```
def backprop(x,y,pred,params,lr):
    w,b = params['w'], params['b']
    gradients = grad(x,y,pred)
    dw, db = gradients['dw'], gradients['db']
    w,b = w - (lr*dw), b - (lr*db)
    return {'w':w, 'b':b}
```

Putting the functions together

[286]:
```
def fit(x, y, params, epochs=10000000, lr=1e-3):
    for i in range(epochs):
        pred = forward(x, params)
        l = loss(pred,y)
        if i%(epochs/10)==0:
            print(f"Loss: {l}")
        params = backprop(x,y,pred,params,lr)
    return params
```

[298]:
```
params = fit(x,y,params,epochs=10000000, lr=1e-3)
```

```
Loss: 921039756400163.5
Loss: 921039756400163.5
Loss: 921039756400163.5
Loss: 921039756400163.5
Loss: 921039756400163.5
Loss: 921039756400163.5
Loss: 921039756400163.5
Loss: 921039756400163.5
Loss: 921039756400163.5
```

```
Loss: 921039756400163.5
```

It's still a huge loss and took millions of iterations with varying learning rates.

It might be because of various factors like not good params initialisation but majorly due to scaling issues between features.

1.1.1 Testing with Scaling/Standardising the data

Normalising the data

[277]:
```
x_mean = np.mean(x, axis=0)
x_std = np.std(x, axis=0)
print(x_mean,x_std, sep="\n")
```

```
[ 6.24028834 12.9396319  18.51656442]
[0.6429913  7.07474478 2.10910764]
```

[278]:
```
x_norm = (x-x_mean)/x_std
x_norm.shape
```

[278]:
```
(489, 3)
```

[281]:
```
x_norm
```

[281]:
```
array([[ 0.52055395, -1.1250769 , -1.5250831 ],
       [ 0.28104837, -0.53706982, -0.33974768],
       [ 1.46924486, -1.25935736, -0.33974768],
       …,
       [ 1.14420158, -1.03178731,  1.17748167],
       [ 0.86114953, -0.91305511,  1.17748167],
       [-0.32704695, -0.71516812,  1.17748167]])
```

[282]:
```
y_mean = np.mean(y,axis=0)
y_std = np.std(y,axis=0)
y_norm = (y-y_mean)/(y_std)
y_norm.shape
```

[282]:
```
(489, 1)
```

[283]:
```
y_norm[:5]
```

[283]:
```
array([[ 0.30064004],
       [-0.00449803],
       [ 1.66104726],
       [ 1.49576414],
       [ 1.85175855]])
```

Initialising news weights (again!)

[287]:
```
w_norm = np.random.randn(1,3)
b_norm = np.random.randn(1)
params1 = {'w':w_norm, 'b':b_norm}
params1
```

[287]:
```
{'w': array([[-0.6557215 ,  0.20712036, -0.41996082]]),
 'b': array([0.84776863])}
```

Using the same fit function we created (but with normalised data)

[309]:
```
params1 = fit(x_norm,y_norm,params1,epochs=100000,lr=1e-10)
```

```
Loss: 33760.59473981772
Loss: 33760.59473981772
Loss: 33760.59473981772
Loss: 33760.59473981772
Loss: 33760.59473981772
Loss: 33760.59473981772
Loss: 33760.59473981772
Loss: 33760.59473981772
Loss: 33760.59473981772
Loss: 33760.59473981772
```

Normalised data was far better in moving towards the down slope - convergence (still by varing the learning rates to avoid overshooting or vanishing gradients)

It's important to note that normalising requires re-scaling features for prediction and then rescaling the results (Don't worry, it's easier to use libraries for it.)

[310]:
```
params1
```

[310]:
```
{'w': array([[ 0.33698803, -0.46470781, -0.24889925]]),
 'b': array([6.65716784e-16])}
```

Weights of the normalised model

Making Predictions with Normalised model (ofcourse using normalised features!)

[311]:
```
print(x_norm[10])
print(y_norm[10])
```

```
[ 0.21261821  1.06157442 -1.57249652]
[-0.84362772]
```

[312]:
```
forward(x_norm[10],params1)
```

[312]:
```
array([-0.03027893])
```

- As said problem with normalised model is we need to scale the data before passing to the model and need to re-scale the predictions later

```
[324]: y_test_pred = forward(x_norm, params1)
```

Rescaling predictions to original scale and computing loss

```
[327]: y_test_pred = (y_test_pred*y_std)+y_mean
```

```
[328]: loss(y_test_pred, y)
```

```
[328]: 921039756400163.6
```

It's kinf of same error in both cases in both normalised data and actual data (But normalised one converages easier compared to original data) 1. 921039756400163.5

2. 921039756400163.6 (normalised)

2 Using Pytorch

2.0.1 Using functions to handle gradient heavy-lifting

```
[329]: import torch
```

```
[472]: # Converting numpy arrays to tensors
       x_tensor = torch.from_numpy(x).float()
       y_tensor = torch.from_numpy(y).float()
```

```
[473]: x_tensor.shape
```

```
[473]: torch.Size([489, 3])
```

```
[474]: y_tensor.shape
```

```
[474]: torch.Size([489, 1])
```

Initialising parameters and setting auto-grad to True

```
[495]: w = torch.randn(1,3, requires_grad=True)
       b = torch.randn(1, requires_grad=True)
```

```
[496]: print(w)
       print(b)
```

```
tensor([[-0.0410,  0.4363, -0.9232]], requires_grad=True)
tensor([-0.1517], requires_grad=True)
```

Manual Implementation

```
[477]: def model(x):
           return x@w.t() + b
```

[497]:
```
preds = model(x_tensor)
```

[479]:
```
preds.shape
```

[479]:
```
torch.Size([489, 1])
```

[537]:
```
def loss(y,pred):
    diff = y-pred
    return torch.sum(diff*diff)/(2*len(y))
```

[538]:
```
l = loss(y_tensor,preds)
```

[539]:
```
l
```

[539]:
```
tensor(3.8518e+09, grad_fn=<DivBackward0>)
```

Using backward function auto computes the grads for the params

[501]:
```
l.backward()
```

[502]:
```
w.grad
```

[502]:
```
tensor([[-2909351.2500, -4990292.5000, -8232276.5000]])
```

Using .no_grad() ensures that we aren't messing up with the gradients and keeping it track of while updating it

[503]:
```
with torch.no_grad():
    w -= w.grad * 1e-5
    b -= b.grad * 1e-5
```

[504]:
```
print(w)
print(b)
```

```
tensor([[29.0525, 50.3393, 81.3996]], requires_grad=True)
tensor([4.3919], requires_grad=True)
```

After each updation we reset the gradients for further computation

[505]:
```
w.grad.zero_()
b.grad.zero_()
```

[505]:
```
tensor([0.])
```

[506]:
```
preds = model(x_tensor)
l = loss(y_tensor, preds)
l
```

```
[506]: tensor(1.1585e+11, grad_fn=<DivBackward0>)
```

```
[507]: w.grad
```

```
[507]: tensor([[0., 0., 0.]])
```

```
[540]: for i in range(100000):
           preds = model(x_tensor)
           l = loss(y_tensor, preds)
           if i%10000==0:
               print(l)
           l.backward()
           with torch.no_grad():
               w -= (w.grad * 1e-3)
               b -= (b.grad * 1e-3)
           w.grad.zero_()
           b.grad.zero_()
```

```
tensor(3.8518e+09, grad_fn=<DivBackward0>)
tensor(3.8518e+09, grad_fn=<DivBackward0>)
tensor(3.8518e+09, grad_fn=<DivBackward0>)
tensor(3.8518e+09, grad_fn=<DivBackward0>)
tensor(3.8518e+09, grad_fn=<DivBackward0>)
tensor(3.8518e+09, grad_fn=<DivBackward0>)
tensor(3.8518e+09, grad_fn=<DivBackward0>)
tensor(3.8518e+09, grad_fn=<DivBackward0>)
tensor(3.8518e+09, grad_fn=<DivBackward0>)
tensor(3.8518e+09, grad_fn=<DivBackward0>)
```

- Note : It seem to have convergered but this point is reached after running it serval iterations with varying learning rates

```
[533]: params_pytorch_manual = {'w':w.detach().numpy(), 'b':b.detach().numpy()}
```

```
[534]: pred_pytorch_manual = forward(x, params_pytorch_manual)
```

```
[535]: np.sum(np.square(y-pred_pytorch_manual))/2*len(y)
```

```
[535]: 921048917510166.1
```

We can see it's a bit far off from the loss compared to our hand coded implementation, let's see whether using the modules and functions from pytorch helps

2.1 Using Modules

```
[363]: from torch.utils.data import TensorDataset
```

Converting the X and Y tensors to a Tensor Dataset

[364]:
```
train_ds = TensorDataset(x_tensor, y_tensor)
train_ds[:3]
```

[364]:
```
(tensor([[ 6.5750,  4.9800, 15.3000],
         [ 6.4210,  9.1400, 17.8000],
         [ 7.1850,  4.0300, 17.8000]]),
 tensor([[504000.],
         [453600.],
         [728700.]]))
```

[365]:
```
from torch.utils.data import DataLoader
```

Creating the Dataloader for our dataset

[366]:
```
batch_size = 5
train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

[367]:
```
for xb,yb in train_dl:
    print(xb,yb, sep="\n")
    break
```

```
tensor([[ 6.6300,  6.5300, 18.5000],
        [ 6.2090,  7.1400, 16.6000],
        [ 6.4330,  9.5200, 19.1000],
        [ 5.9830, 18.0700, 20.1000],
        [ 4.9060, 34.7700, 20.2000]])
tensor([[558600.],
        [487200.],
        [514500.],
        [285600.],
        [289800.]])
```

[368]:
```
import torch.nn as nn
```

Using in-built functions for the implementation

[375]:
```
model = nn.Linear(3,1)
print(model.weight)
print(model.bias)
```

```
Parameter containing:
tensor([[ 0.0597, -0.5588,  0.2815]], requires_grad=True)
Parameter containing:
tensor([0.2969], requires_grad=True)
```

[376]:
```
list(model.parameters())
```

```
[376]: [Parameter containing:
        tensor([[ 0.0597, -0.5588,  0.2815]], requires_grad=True),
        Parameter containing:
        tensor([0.2969], requires_grad=True)]
```

```
[377]: preds = model(x_tensor)
       preds[:3]
```

```
[377]: tensor([[2.2140],
                [0.5841],
                [3.4851]], grad_fn=<SliceBackward0>)
```

```
[378]: from torch.functional import F
```

```
[379]: loss_fn = F.mse_loss
```

```
[380]: loss = loss_fn(model(x_tensor),y_tensor)
       loss
```

```
[380]: tensor(2.3371e+11, grad_fn=<MseLossBackward0>)
```

```
[403]: optim = torch.optim.SGD(model.parameters(),lr=1e-25)
```

```
[404]: def fit(model, train_dl, loss_fn, optim, epochs):

           for i in range(epochs):

               for xb,yb in train_dl:
                   pred = model(xb)
                   loss = loss_fn(pred, yb)

                   loss.backward()

                   optim.step()
                   optim.zero_grad()

               if i%10==0:
                   print("Epoch {}/{} loss:{:.4f}".format(i+1, epochs, loss.item()))
```

```
[407]: fit(model, train_dl, loss_fn, optim, 100)
```

```
Epoch 1/100 loss:2973650944.0000
Epoch 11/100 loss:1214444032.0000
Epoch 21/100 loss:24651794432.0000
Epoch 31/100 loss:932059648.0000
Epoch 41/100 loss:3756283392.0000
Epoch 51/100 loss:31362609152.0000
Epoch 61/100 loss:693265728.0000
Epoch 71/100 loss:7747588096.0000
Epoch 81/100 loss:4008612864.0000
Epoch 91/100 loss:3438778112.0000
```

- Not so optimal loss as we can see flautuations in the loss even at very low learning rates.

```
[428]: m = list(model.parameters())
       w_pytorch = m[0].detach().numpy()
       b_pytorch = m[1].detach().numpy()
```

```
[429]: params_pytorch = {'w':w_pytorch, 'b':b_pytorch}
```

```
[430]: pred_pytorch = forward(x, params_pytorch)
```

```
[432]: np.sum(np.square(y-pred_pytorch))/2*len(y)
```

```
[432]: 1041863161861592.6
```

- Very low performance compared to our hand coded Pure mathematical one, might be depends on the choice of the optimiser and the loss function in addition to the accuracy of gradient calculation

2.1.1 Using the Sklearn ML library

```
[451]: from sklearn.linear_model import LinearRegression
```

```
[447]: df.head()
```

```
[447]:       RM  LSTAT  PTRATIO      MEDV
       0  6.575   4.98     15.3  504000.0
       1  6.421   9.14     17.8  453600.0
       2  7.185   4.03     17.8  728700.0
       3  6.998   2.94     18.7  701400.0
       4  7.147   5.33     18.7  760200.0
```

```
[448]: x_ml_lib = df.iloc[:,:3]
       x_ml_lib.head()
```

```
[448]:       RM  LSTAT  PTRATIO
       0  6.575   4.98     15.3
       1  6.421   9.14     17.8
       2  7.185   4.03     17.8
       3  6.998   2.94     18.7
       4  7.147   5.33     18.7
```

```
[459]: y_ml_lib = df.iloc[:,-1]
       y_ml_lib.head()
```

```
[459]: 0    504000.0
       1    453600.0
       2    728700.0
       3    701400.0
       4    760200.0
       Name: MEDV, dtype: float64
```

```
[460]: model_ml = LinearRegression()
```

```
[461]: model_ml.fit(X=x_ml_lib, y=y_ml_lib)
```

```
[461]: LinearRegression()
```

```
[462]: model_ml.score(x_ml_lib,y_ml_lib)
```

```
[462]: 0.7176275212982739
```

```
[455]: pred_ml_lib = model_ml.predict(x_ml_lib)
```

```
[466]: pred_ml_lib = pred_ml_lib.reshape(489,1)
```

```
[467]: np.sum(np.square(y-pred_ml_lib))/2*len(y)
```

```
[467]: 921039756400163.5
```

Summary: Loss we got for Linear Regression using Sklearn is exactly same as we attained for our model implemented purely using calculus, thus we had learnt the mathematical implementation of linear regression..

Though the pytorch implementation doesn't yield us the same result and it's the topic of dicussion in next notebook on our journey of Deep Leaning…

- Happy coding!