8.1 - Regression Scratch

September 17, 2020

0.1 Regression from Scratch

[1]:
```
import numpy as np
from sklearn.datasets import load_boston
```

Let's load some boston data as our regression case study

[2]:
```
#type - Bunch
#Bunch - dictionary of numpy data
#boston.feature_names
#print(boston)
boston = load_boston()
```

0.1.1 Step 1: Prepare your data

1.1 Get your X and y in the right shape

[3]:
```
X = boston.data
X.shape #number of samples, number of features

m = X.shape[0]  #number of samples
n = X.shape[1]  #number of features
```

[4]:
```
y = boston.target
```

[5]:
```
#number of rows in X is the same as number of rows in y
#because so we have yhat for all y
assert m == y.shape[0]
```

1.2 Feature scale your data to reach faster convergence

[6]:
```
#I want to standardize my data so that mean is 0, variance is 1
#average across each feature, NOT across each sample
#Why we need to standardize
#Because standardizing usually allows us to reach convergence faster
#Why -> because the values are within smaller range
#Thus, the gradients are also within limited range, and NOT go crazy

from sklearn.preprocessing import StandardScaler
```

```
#1. StandardScaler.fit(X)  #this scaler (or self) knows the mean and std so now
# it knows how to transform data
#2  X = StandardScaler.transform(X)  #not in place; will return something

#1. StandardScaler.fit_transform(X) -> 1 and 2 sequentially

#create an object of StandardScaler
#StandardScaler is a class
#scaler is called instance/object

#ALMOST always, feature scale your data using normalization or standardization
#If you assume your data is gaussian, use standardization, otherwise, you do␣
 ↪the normalization

scaler = StandardScaler()

X = scaler.fit_transform(X)
```

1.3 Train test split your data

[7]:
```
#what is the appropriate size for test data
#70/30 (small dataset); 80/20 (medium dataset); 90/10 (large dataset);
#why large dataset, can set test size to 10, because
#10% of large dataset is already enough for testing accuracy
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3)

assert len(X_train)  == len(y_train)
assert len(X_test) == len(y_test)
```

1.4 Add intercepts

[8]:
```
#What is the shape of X they want
#(number of samples, number of features) --> correct shape
# for closed form formula
#How about the intercept
#w0 is OUR intercept
#what is the shape of w -->(n+1, )
#What is the shape of intercept --->(m, 1)
#X = [1 2 3     @  [w0
#     1 4 6          w1
#     1 9 1          w2
#     1 10 2 ]

#np.ones((shape))
```

```
intercept = np.ones((X_train.shape[0], 1))

#concatenate the intercept based on axis=1
X_train = np.concatenate((intercept, X_train), axis=1)

#np.ones((shape))
intercept = np.ones((X_test.shape[0], 1))

#concatenate the intercept based on axis=1
X_test = np.concatenate((intercept, X_test), axis=1)
```

1.5. Feature Engineering (optional) It is sometimes useful to engineer new features (e.g., polynomial, kernels) so to create some non-linear relationships with your target.

Here we gonna skip

0.1.2 Algorithm 1: Closed Form

The closed form is a normal equations derived from setting the derivatives = 0. By performing only some inverse operations and matrix multiplication, we will be able to get the theta.

When closed form is available, is doable (can be inversed - can use pseudoinverse), and with not many features (i.e., inverse can be slow), it is recommended to always use closed form.

$$\theta = (X^T X)^{-1} X^T Y$$

1. Define your algorithm

[9]:
```
#Closed form
#How to get Closed Form
#Simple; Set the d(cost function) = 0
#And find the \theta that satisfy the equation
#When we can do such a thing in which we set the d(cost function) = 0
#--->When its strictly concave, or strictly convex
#----> They have only one local maximum (concave), minimum (convex)
from numpy.linalg import inv

#run it, and return me the theta
#which one do first DOES NOT MATTER
#But don't flip y before X^T for example
def closed_form(X, y):
    return inv(X.T @ X) @ X.T @ y
```

[10]:
```
#let's use the closed_form to find the theta
theta = closed_form(X_train, y_train)
theta  #<------this is our model
```

[10]:
```
array([22.64917743, -1.05547574,  1.22332735, -0.1144123 ,  0.75230475,
       -2.03959642,  2.69724278, -0.06751152, -3.528664  ,  2.94806347,
       -2.10031456, -2.15895157,  0.88454597, -3.93938543])
```

2. Compute accuracy/loss

[11]:
```
#Compute the accuracy/loss

#6.1 predict --> \theta^T x
yhat = X_test @ theta #==> X (m, n+1)  @ (n+1, ) w ==> (m, ) y

#if I want to compare yhat and y, I need to make sure they are the same shape
assert y_test.shape == yhat.shape
```

[12]:
```
#get the errors
errors = ((y_test - yhat)**2).sum()
```

[13]:
```
print(errors)
```

```
2710.09911982039
```

[14]:
```
X_train.shape
```

[14]:
```
(354, 14)
```

[15]:
```
y_train.shape
```

[15]:
```
(354,)
```

0.1.3 Algorithm 2: Batch Gradient Descent

The gradient descent has the following steps:

1. Prepare your data
   - add intercept
   - X and y in the right shape
   - train-test split
   - feature scale
   - clean out any missing data
   - (optional) feature engineering
2. Predict and calculate the loss
   - The loss function is the mean squared error

$$J = \frac{\Sigma_{i=1}^{m}(h-y)^2}{m}$$

   where h is simply

$$h = \theta^T x$$

3. Calculate the gradient based on the loss

- The gradient of the loss function is

$$\frac{\partial J}{\partial \theta_j} = \Sigma_{i=1}^{m}(h^{(i)} - y^{(i)})x_j$$

4. Update the theta with this update rule

$$\theta_j := \theta_j - \alpha * \frac{\partial J}{\partial \theta_j}$$

   where α is a typical learning rate range between 0 and 1
5. Loop 2-4 until max_iter is reached, or the difference between old loss and new loss are smaller than some predefined threshold tol

1. Define your algorithm

[16]:
```
from time import time

#Step 1: Prepare your data
#X_train, X_test have intercepts that are being concatenated to the data
#[1, features
# 1, features....]

#making sure our X_train has same sample size as y_train
assert X_train.shape[0] == y_train.shape[0]

#initialize our w
#We don't have to do X.shape[1] + 1 because our X_train already has the
#intercept
#w = theta/beta/coefficients
theta = np.zeros(X_train.shape[1])

#define the learning rate
#later on, you gonna know that it should be better to make it slowly decreasing
#once we perform a lot of iterations, we want the update to slow down, so it␣
 ↪converges better
alpha = 0.0001

#define our max_iter
#typical to call it epochs <---ml people likes to call it
max_iter = 1000

loss_old = 10000

tol = 0.0001

iter_stop = 0

def h_theta(X, theta):
    return X @ theta
```

```
def mse(yhat, y):
    return ((yhat - y)**2 / yhat.shape[0]).sum()

def delta_loss(new, old, tol):
    return np.abs(loss_new - loss_old) < tol

def gradient(X, error):
    return X.T @ error

start = time()

#define your for loop
for i in range(max_iter):

    #1. yhat = X @ w
    #prediction
    #yhat (m, ) = (m, n) @ (n, )
    yhat = h_theta(X_train, theta)

    #2. error = yhat - y_train
    #error for use to calculate gradients
    #error (m, ) = (m, ) - (m, )
    error = yhat - y_train

    #2.1 early stopping
    #so we don't go through all max_iter iterations
    # )yi_hat - yi )^2 / m  <--- mse
    #loss_new (scalar) = ((m, ) - (m, ) **2 / m).sum()
    loss_new = mse(yhat, y_train)
    if delta_loss(loss_new, loss_old, tol):  #np.allclose
        iter_stop = i
        break
    loss_old = loss_new

    #3. grad = X.T @ error
    #grad (n, ) = (n, m) @ (m, )
    #grad for each feature j
    grad = gradient(X_train, error)

    #4. w = w - alpha * grad
    #update w
    #w (n, ) = (n, ) - scalar * (n, )
    theta = theta - alpha * grad

time_taken = time() - start
```

2. Compute accuracy/loss

[17]:
```
#we got our lovely w
#now it's time to check our accuracy
#1. Make prediction
yhat = h_theta(X_test, theta)

#2. Calculate mean squared errors
mse = mse(yhat, y_test)

#print the mse
print("MSE: ", mse)
print("Stop at iteration: ", iter_stop)
print("Time used: ", time_taken)
```

```
MSE:  17.819515132071835
Stop at iteration:  797
Time used:  0.017093181610107422
```

0.1.4 Algorithm 3: Stochastic Gradient Descent

The gradient descent has the following steps:

1. Prepare your data
   - add intercept
   - X and y in the right shape
   - train-test split
   - feature scale
   - clean out any missing data
   - (optional) feature engineering
2. Predict and calculate the loss
3. Calculate the gradient based on the loss
   - **This differs from batch gradient descent that it only uses one sample to estimate the loss and gradient**

     $$\frac{\partial J}{\partial \theta_j} = (h^{(i)} - y^{(i)})x_j$$

     where i is some random number
4. Update the theta
5. Loop 2-4 until max_iter is reached, or the difference between old loss and new loss are smaller than some predefined threshold tol

1. Define your algorithm

[18]:
```
#Stochastic version

def h_theta(X, w):
    '''
    Input:
        X shape (m, n)
```

```
        w shape (n, )
    Returns:
        (m, )
    '''
    return X @ w

def mse(yhat, y):
    return ((yhat - y)**2 / yhat.shape[0]).sum()

def singe_mse(yhat, y):
    #yhat (1, ) - (1, ) ** 2
    return (yhat - y)**2

def delta_loss(new, old, tol):
    return np.abs(loss_new - loss_old) < tol

def gradient(X, error):
    return X.T @ error

#initialize our w
#We don't have to do X.shape[1] + 1 because our X_train already has the
#intercept
#w = theta/beta/coefficients
theta = np.zeros(X_train.shape[1])

#define the learning rate
#later on, you gonna know that it should be better to make it slowly decreasing
#once we perform a lot of iterations, we want the update to slow down, so it␣
 ↪converges better
alpha = 0.01
loss_old = 10000
tol = 0.0001
iter_stop = 0
max_epochs = 10000

start = time()

#define your for loop
for epoch in range(max_epochs):  #max_iter is the same as epochs

    #we have indices for all samples
    i = np.random.randint(X_train.shape[0])

    #1. yhat = X_i @ w
    #X_i (1, n)
    X_i = X_train[i, :].reshape(1, -1)
```

```
    #prediction
    #yhat (1, ) = (1, n) @ (n, )
    yhat = h_theta(X_i, theta)

    #2. error = yhat - y_i
    #y_i (1, )
    y_i = y_train[i]
    #error for use to calculate gradients
    #error (1, ) = (1, ) - (1, )
    error = yhat - y_i

    #2.1 early stopping
    #so we don't go through all max_iter iterations
    # (yi_hat - yi )^2 / m  <--- mse
    #loss_new (scalar) = ((m, ) - (m, ) **2 / m).sum()
    loss_new = singe_mse(yhat, y_i)
    if delta_loss(loss_new, loss_old, tol):  #np.allclose
        iter_stop = epoch
        break
    loss_old = loss_new

    #3. grad = X.T @ error
    #grad (n, ) = (n, 1) @ (1, )
    #grad for each feature j
    grad = gradient(X_i, error)

    #4. w = w - alpha * grad
    #update w
    #w (n, ) = (n, ) - scalar * (n, )
    theta = theta - alpha * grad

time_taken = time() - start
```

2. Compute accuracy/loss

[19]:
```
#we got our lovely w
#now it's time to check our accuracy
#1. Make prediction
yhat = h_theta(X_test, theta)

#2. Calculate mean squared errors
mse = mse(yhat, y_test)

#print the mse
print("MSE: ", mse)
print("Stop at iteration: ", iter_stop)
print("Time used: ", time_taken)
```

```
#the time taken is strangely higher than gradient descent
#perhaps using stochastic with decay learning rate may give us
#a better chance
```

```
MSE:  19.677024613338126
Stop at iteration:  0
Time used:  0.2084660530090332
```

0.1.5 Algorithm 4: Mini-Batch Gradient Descent

The gradient descent has the following steps:

1. Prepare your data
   - add intercept
   - X and y in the right shape
   - train-test split
   - feature scale
   - clean out any missing data
   - (optional) feature engineering
2. Predict and calculate the loss
3. Calculate the gradient based on the loss
   - **This differs from batch gradient descent that it only uses a subset of samples to estimate the loss and gradient**

     $$\frac{\partial J}{\partial \theta_j} = \Sigma_{i=start}^{batch}(h^{(i)} - y^{(i)})x_j$$

     where start is a randomized number within the range of m and batch is a predefined batch size, typically around 100 to 500
4. Update the theta
5. Loop 2-4 until max_iter is reached, or the difference between old loss and new loss are smaller than some predefined threshold tol

I will not implement this, but leave to your exercise. Enjoy!

[ ]: