# lab-3-pandas-pytorch–data-handling-LR

February 23, 2023

[206]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

[207]:
```
pd.__version__
```

[207]:
```
'1.4.2'
```

Pandas objects can be thought of as enhanced versions of NumPy structured arrays in which the rows and columns are identified with labels rather than simple integer indices.

[230]:
```
# loading data using pandas
df = pd.read_csv("spam_or_not_spam.csv")
```

[242]:
```
len(df[df['label'] == 0]) # not spam
```

[242]:
```
2500
```

[243]:
```
len(df[df['label'] == 1]) # spam
```

[243]:
```
500
```

[253]:
```
data = pd.read_csv('data/president_heights.csv')
heights = np.array(data['height(cm)'])
#print(heights)
```

[251]:
```
#df.head()
#data['height(cm)']
```

[254]:
```
print("Mean height:       ", heights.mean())
print("Standard deviation:", heights.std())
print("Minimum height:    ", heights.min())
print("Maximum height:    ", heights.max())
```

```
Mean height:        179.73809523809524
Standard deviation: 6.931843442745892
Minimum height:     163
Maximum height:     193
```

[256]:
```python
data[data['height(cm)'] == 193]
```

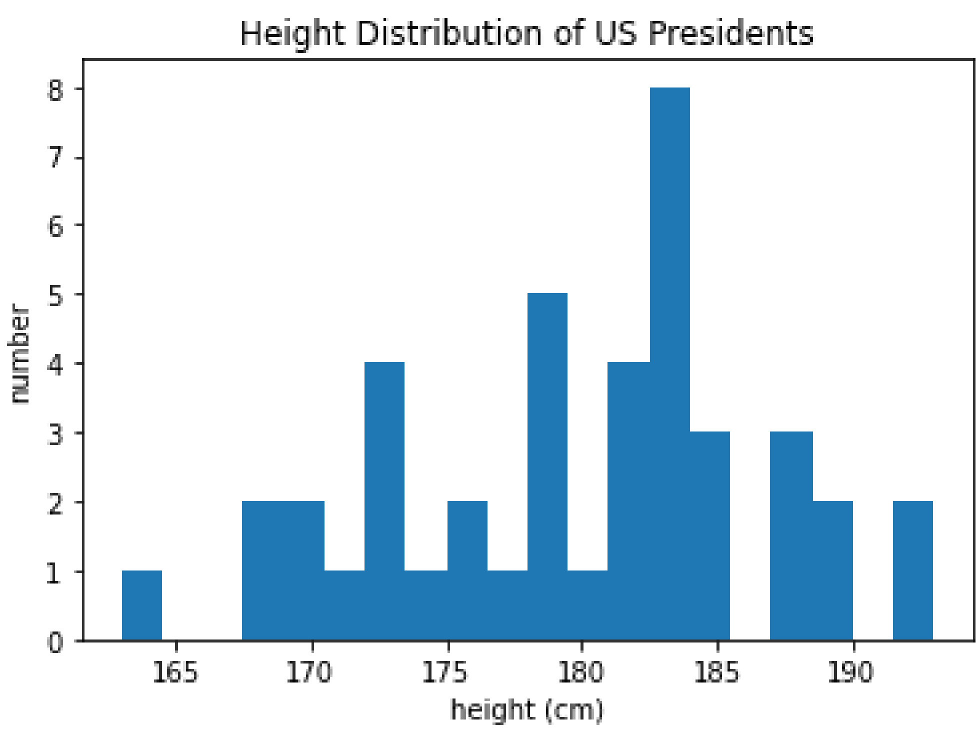
[256]:

| | order | name | height(cm) |
|---|---|---|---|
| 15 | 16 | Abraham Lincoln | 193 |
| 33 | 36 | Lyndon B. Johnson | 193 |

[263]:
```python
print("25th percentile:   ", np.percentile(heights, 25))
print("Median:            ", np.median(heights))
print("75th percentile:   ", np.percentile(heights, 75))
```

```
25th percentile:    174.25
Median:             182.0
75th percentile:    183.0
```

[264]:
```python
plt.hist(heights, bins=20)
plt.title('Height Distribution of US Presidents')
plt.xlabel('height (cm)')
plt.ylabel('number');
```




[265]:
```python
file = "P00009"
filelocation = "../../../rishiraj/Documents/thermal-project-data/cycling/
↪Full_Study/"+file+"/"+file+".TXT"
```

```
data_ = pd.read_csv(filelocation,skiprows=1, sep='\t',encoding = "utf-16",␣
 ↪header=0)
```

[277]:
```
np.mean(np.array(data_['Rf'][1:]).astype('int'))
```

[277]: 18.22568093385214

[270]:
```
np.array(data_['Rf'])[6]
```

[270]: '26'

[13]:
```
# Correlation
np.corrcoef
```

# 1 Pytorch

A 3D matrix is a tensor.

[23]:
```
# install pytorch
```

[ ]:
```
# [1,2,3,4] # vetor
# [[1,2], [3,4], [6,6]] # 3 x 2 matrix
#
```

[278]:
```
import torch
```

[281]:
```
t = torch.tensor([[1,2,3],[4,5,6]])
t
```

[281]:
```
tensor([[1, 2, 3],
        [4, 5, 6]])
```

[282]:
```
t.shape
```

[282]: torch.Size([2, 3])

[283]:
```
t.ndim
```

[283]: 2

[286]:
```
# Pytorch vs Numpy
a = np.array([1.,2.,3.])
print(a.dtype)

b = torch.tensor([1.,2.,3.])
print(b.dtype)
```

```
# Can you tell the advantage and disadvantage of both?
```

```
float64
torch.float32
```

[289]:
```
a.dot(a)
```

[289]:
```
14.0
```

[290]:
```
b.dot(b), b.matmul(b), b@b
```

[290]:
```
(tensor(14.), tensor(14.), tensor(14.))
```

[15]:
```
# convert tensor to numpy array
b.numpy()
```

[15]:
```
array([1., 2., 3.], dtype=float32)
```

[294]:
```
b.dtype
```

[294]:
```
torch.float32
```

[295]:
```
b.to(torch.double) # not an inplace operation
```

[295]:
```
tensor([1., 2., 3.], dtype=torch.float64)
```

[296]:
```
b.double()
```

[296]:
```
tensor([1., 2., 3.], dtype=torch.float64)
```

## 1.1 Why do we care about Pytorch?

- GPU support, which means better parallelism
- Pyorch has support for automatic differentiation
- DL convenience function

[297]:
```
print(torch.cuda.is_available())
```

```
False
```

[300]:
```
# matrix multiplication
a = torch.arange(6).view(2,3)
print(a.shape)
b = torch.tensor([1,2,3])
print(b.shape)
torch.matmul(a, b.view(-1,1))
```

```
torch.Size([2, 3])
torch.Size([3])
```

[300]:
```
tensor([[ 8],
        [26]])
```

[303]:
```
# broadcasting
te = torch.tensor([[4,5,6],[7,8,9]]) #(2,3)
te
```

[303]:
```
tensor([[4, 5, 6],
        [7, 8, 9]])
```

[30]:
```
te+torch.tensor([1,2,3])
```

[30]:
```
tensor([[ 5,  7,  9],
        [ 8, 10, 12]])
```

## 2 Data Standardisation and Data Normalisation

[311]:
```
x = np.random.randint(10,100, 10000)
#plt.plot(x)
```

[306]:
```
np.mean(x), np.var(x) # 0
```

[306]:
```
(54.5208, 670.9025673599999)
```

[ ]:
```
# height(ft) 4-7, age 18-70,
```

[307]:
```
# standardisation -> mean centering.
x = (x-np.mean(x))/np.var(x)
```

[309]:
```
np.mean(x), np.var(x)
```

[309]:
```
(-1.9040324872321433e-18, 0.0014905293982328878)
```

[312]:
```
# normalisation
x, x/np.max(x) # -2,2
```

[312]:
```
(array([60, 42, 67, …, 31, 77, 45]),
 array([0.60606061, 0.42424242, 0.67676768, …, 0.31313131, 0.77777778,
        0.45454545]))
```

[ ]:
```

```

## 3 Linear Regression

Recall the closed form solution.

$$Y = X\theta^T$$

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

[325]:
```
X = np.random.randn(500,1)
y = 2*X + 1 + 1.2*np.random.randn(500,1)
print(X.shape, y.shape)
```

```
(500, 1) (500, 1)
```

[328]:
```
def find_theta(X, y):
    m = X.shape[0] # Number of training examples.     # Appending a cloumn of␣
 ↪ones in X to add the bias term.
    X = np.append(np.ones((X.shape[0],1)), X, axis=1)       # reshaping y to␣
 ↪(m,1)
    y = y.reshape(m,1)

    # The Normal Equation
    theta = np.dot(np.linalg.inv(np.matmul(X.T, X)), np.matmul(X.T, y))

    return theta
```

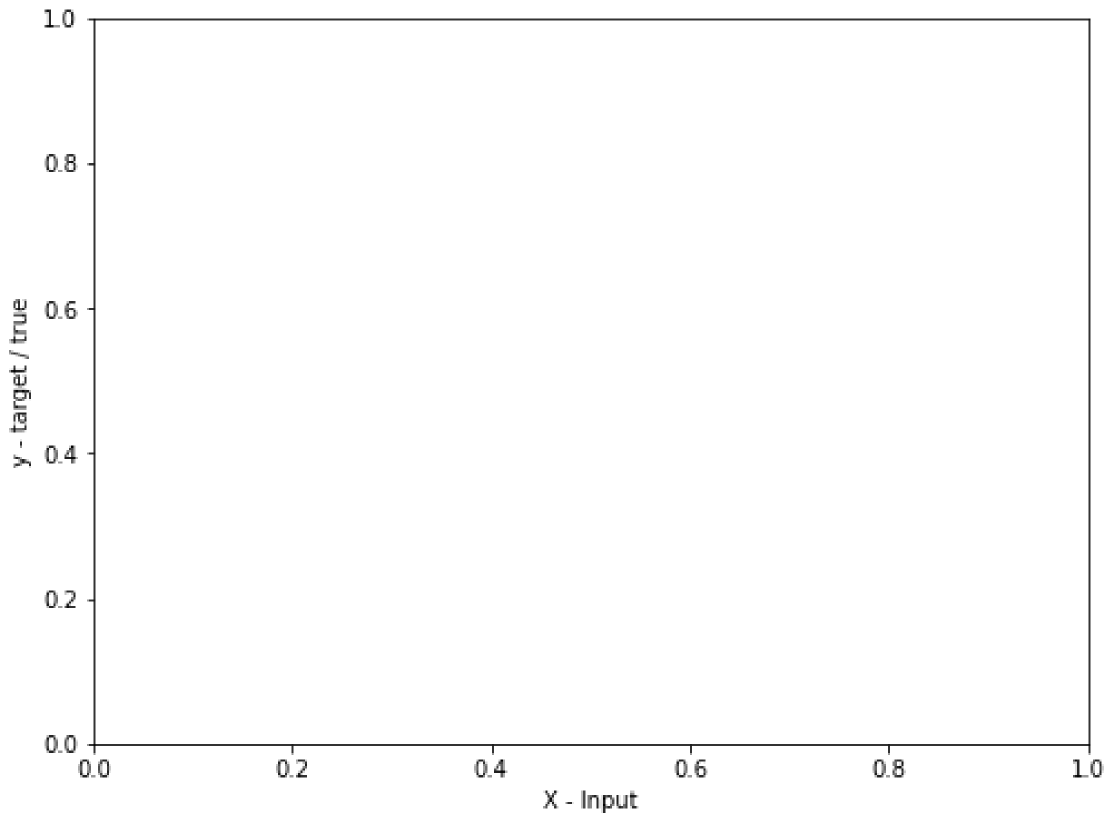
[331]:
```
def predict(X):

    # Appending a cloumn of ones in X to add the bias term.
    X = np.append(X, np.ones((X.shape[0],1)), axis=1)

    # preds is y_hat which is the dot product of X and theta.
    preds = np.dot(X, theta)

    return preds
```

[336]:
```
X.shape, theta.T.shape
```

[336]:
```
((500, 1), (1, 2))
```

[340]:
```
# Getting the Value of theta using the find_theta function.
theta = find_theta(X, y)
#preds = predict(X)# Plotting the predictions.
fig = plt.figure(figsize=(8,6))
#plt.plot(X, y, 'k.')
#plt.plot(X, np.dot(X, theta.T), 'r-')
plt.xlabel('X - Input')
plt.ylabel('y - target / true')
```

[340]: Text(0, 0.5, 'y - target / true')




## 4 Linear Regression

Gradient Descent Method

$$Y = X\theta^T$$

When we don't know the true $\theta$, we will have,

$$\hat{Y} = X\hat{\theta^T}$$

with an error

$$\epsilon = Y - \hat{Y}$$

[343]:
```
X = np.random.randn(500,1)
y = 2*X + 1 + 1.2*np.random.randn(500,1)
X = np.append(np.ones((X.shape[0], X.shape[1])),X, axis=1)
print(X.shape, y.shape)
```

```
(500, 2) (500, 1)
```

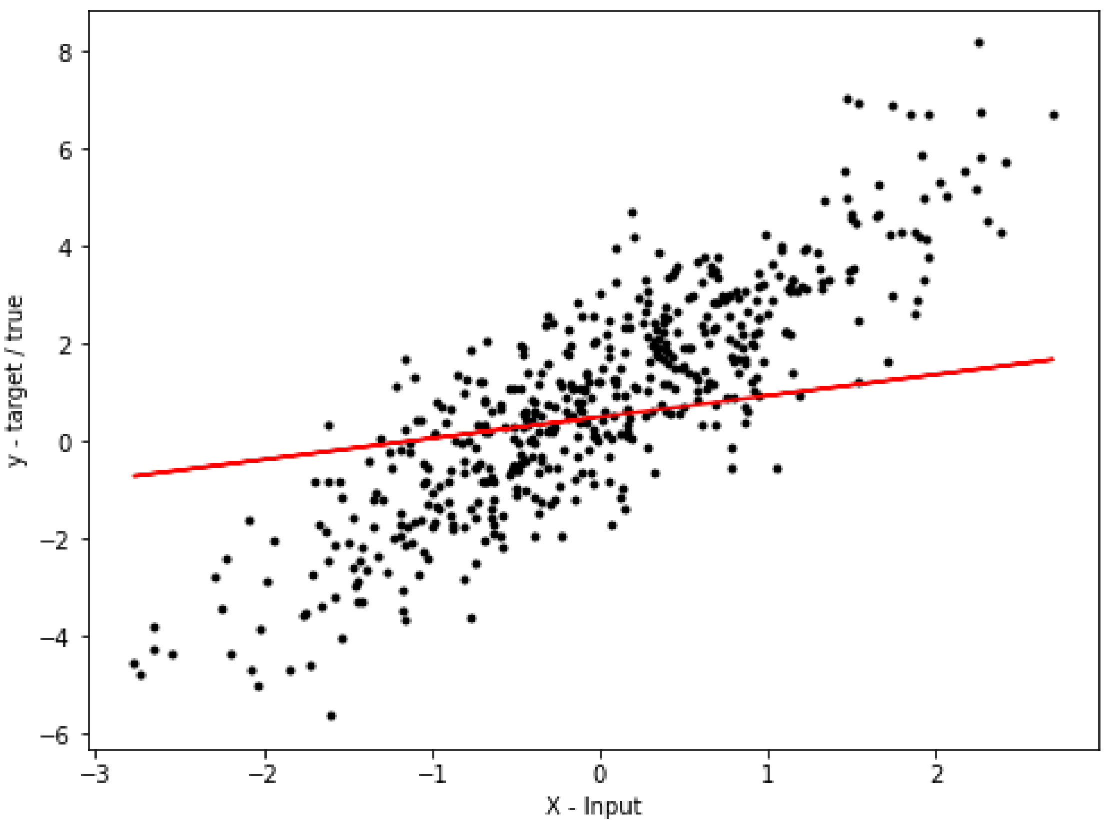
[359]:
```
np.random.seed(0)
theta = np.random.normal(0,1,(1,X.shape[1]))
```

[362]:
```
theta
```

[362]: array([[1.76405235, 0.40015721]])

[383]:
```
fig = plt.figure(figsize=(8,6))
plt.plot(X[:,1], y, 'k.')
plt.plot(X[:,1], np.dot(X, theta.T), 'r-')
plt.xlabel('X - Input')
plt.ylabel('y - target / true')
```

[383]: Text(0, 0.5, 'y - target / true')




[183]:
```
def predictGD(X, theta):
    return np.dot(X, theta.T)
```

[184]:
```
def error(predicted, true):
    '''
    Computed MSE
```

```python
    '''
    return np.mean((true-predicted)**2)
```

[384]:
```python
error(predictGD(X, theta), y)
```

[384]: 3.779278125788885

We are interested in reducing the error ($\epsilon$) by changing the $\theta$

$$\epsilon = Y - \hat{Y}$$

or

$$\epsilon_i = y_i - \hat{y}_i$$

In the case where we have one feature only.

$$\sum {\epsilon_i}^2 = \sum (y_i - (\theta_0 - \theta_1 x_i))^2$$

**Gradient Descent Algorithm**

$$\theta_0 = \theta_0 - \frac{\delta}{\delta\theta_0}(\sum \epsilon^2)$$

$$\theta_1 = \theta_1 - \frac{\delta}{\delta\theta_1}(\sum \epsilon^2)$$

$$\frac{\delta}{\delta\theta_0}(\sum \epsilon^2) = 2\sum(y_i - (\theta_0 + \theta_1 x_i))(-1)$$

$$\frac{\delta}{\delta\theta_1}(\sum \epsilon^2) = 2\sum(y_i - (\theta_0 + \theta_1 x_i))(-x_1)$$

[171]:
```python
lr = 0.1
predicted = predictGD(X, theta)
theta[:,0] = theta[:,0] - 2*lr*np.mean((y-predicted)*(-X[:,0]))
theta[:,1] = theta[:,1] - 2*lr*np.mean((y-predicted)*(-X[:,1]))
```

[382]:
```python
# take advantage of vectorisation
lr = 0.1
theta = theta - 2*lr*np.mean(y-predicted)*(-np.mean(X, axis=0))
```

[179]:
```python
#ex1 = np.random.randint(4,5, (1,2))
#print(ex1)
#ex2 = ex1 - np.mean(np.random.randint(1,2,(500,2)), axis=0)
#print(ex2)
```

```
[[4 4]]
[[3. 3.]]
```

### 4.1 Fully Connected Layer In Pytorch

[31]:
```
X = torch.arange(50, dtype=torch.float).view(10,5)
X
```

[31]:
```
tensor([[ 0.,  1.,  2.,  3.,  4.],
        [ 5.,  6.,  7.,  8.,  9.],
        [10., 11., 12., 13., 14.],
        [15., 16., 17., 18., 19.],
        [20., 21., 22., 23., 24.],
        [25., 26., 27., 28., 29.],
        [30., 31., 32., 33., 34.],
        [35., 36., 37., 38., 39.],
        [40., 41., 42., 43., 44.],
        [45., 46., 47., 48., 49.]])
```

[34]:
```
fc_layer = torch.nn.Linear(in_features=X.shape[1], out_features=3)
```

[35]:
```
fc_layer.weight
```

[35]:
```
Parameter containing:
tensor([[ 0.0583,  0.0581, -0.2820, -0.0436, -0.1690],
        [ 0.2762,  0.2168,  0.1687,  0.4142,  0.1183],
        [-0.0519,  0.0223, -0.2577, -0.1936, -0.0816]], requires_grad=True)
```

[36]:
```
fc_layer.bias
```

[36]:
```
Parameter containing:
tensor([-0.0504,  0.1824, -0.3723], requires_grad=True)
```

[40]:
```
A = fc_layer(X)
A.shape
```

[40]:
```
torch.Size([10, 3])
```

[ ]:
```

```