1a (i)

```
from keras.datasets import mnist
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import random
import pandas as pd
import seaborn as sns
```

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

```
x_train.shape
```

```
(60000, 28, 28)
```

```
y_train.shape
```

```
(60000,)
```

```
y_train
```

```
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

```
from google.colab.patches import cv2_imshow
cv2_imshow(x_train[0])
cv2_imshow(x_train[2])
```

```
my_list = np.array(y_train)
indices_all=[]
for i in range(0,10):
  indices = np.where(my_list == i)[0]
  indices = indices.tolist()
  indices_all.append(indices)
# display result
print(indices_all)
```

```
[[1, 21, 34, 37, 51, 56, 63, 68, 69, 75, 81, 88, 95, 108, 114, 118, 119, 121,
```

```
n = 5
indices_store=[]
for i in range(0,10) :
  indices_store.append(random.sample(indices_all[i], n))
```

```
indices_store
```

```
[[30187, 51669, 18093, 53290, 53424],
 [38538, 41282, 21357, 49680, 16914],
 [27451, 18206, 44226, 48877, 25150],
 [12920, 31140, 41048, 51701, 54285],
 [33554, 50656, 31111, 14027, 41020],
 [31584, 7848, 44713, 56695, 16558],
 [25171, 37829, 5141, 3736, 46632],
 [21693, 23127, 16809, 20105, 22350],
 [53737, 47498, 19582, 34933, 36974],
 [32848, 8510, 27440, 19914, 49227]]
```

```
indices_store[0][0]
```

```
30187
```

```
for i in range(0,10):
  for c in range(0,5):
    cv2_imshow(x_train[indices_store[i][c]])
```

1a(ii)(A)

```
x_train.shape
```

```
(60000, 28, 28)
```

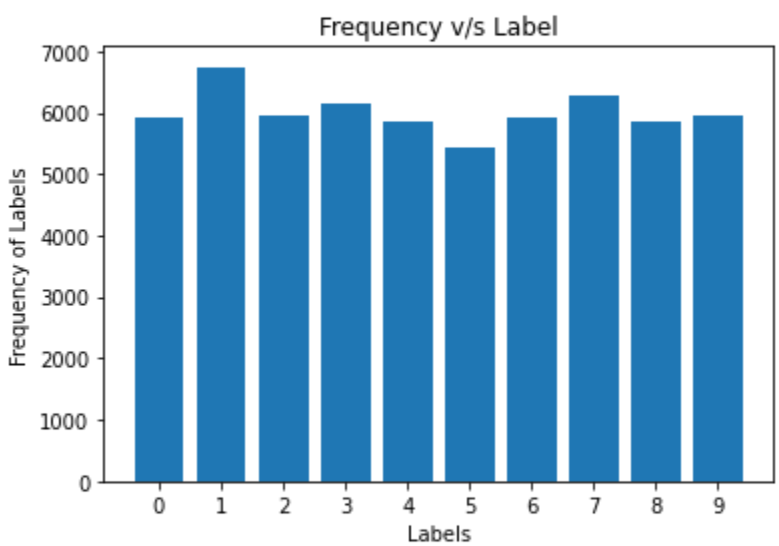
```
x_test.shape
```

```
(10000, 28, 28)
```

We observe in trainX, there are 60000 images of shape 28x28 and in testX there are 10000 images of shape 28x28. Thus, the shape of all images is 28x28. Hence, yes, all images are of the same size.

1a(ii)(B)

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()

labels, count = np.unique(y_train, return_counts=True)

plt.bar(labels, count, align='center')
plt.xticks(labels)
plt.xlabel('Labels')
plt.ylabel('Frequency of Labels')
plt.title('Frequency v/s Label')
plt.show()
```



Since there is no significant difference in the frequencies, there is no substantial class imbalance as the number of images for all labes is approximately equal to 6000.

1a(ii)(C)

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
print("x_train before normalization: ", x_train[0])
x_train = x_train.astype('float32')/255
x_test = x_test.astype('float32')/255
print("x_train after normalization: ", x_train[0])
```

```
x_train before normalization:  [[  0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   3  18  18  18 126 136
  175  26 166 255 247 127   0   0   0   0]
 [  0   0   0   0   0   0   0   0  30  36  94 154 170 253 253 253 253 253
  225 172 253 242 195  64   0   0   0   0]
 [  0   0   0   0   0   0   0  49 238 253 253 253 253 253 253 253 253 251
   93  82  82  56  39   0   0   0   0   0]
 [  0   0   0   0   0   0   0  18 219 253 253 253 253 253 198 182 247 241
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  80 156 107 253 253 205  11   0  43 154
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0  14   1 154 253  90   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 139 253 190   2   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0  11 190 253  70   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  35 241 225 160 108   1
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0  81 240 253 253 119
   25   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0  45 186 253 253
  150  27   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  16  93 252
  253 187   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 249
  253 249  64   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0  46 130 183 253
  253 207   2   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  39 148 229 253 253 253
  250 182   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0  24 114 221 253 253 253 253 201
   78   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  23  66 213 253 253 253 253 198  81   2
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0  18 171 219 253 253 253 253 195  80   9   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0  55 172 226 253 253 253 253 244 133  11   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0 136 253 253 253 212 135 132  16   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

```
    0   0   0   0   0   0   0   0   0   0]]
x_train after normalization:  [[0.         0.         0.         0.         0
  0.         0.         0.         0.         0.         0.
```

We can normalise x_train by dividing it by the maximum pixel value, i.e. 255. To visualise, I have printed the first element of x_train before and after normalisation. The value before normalisation ranges from 0 to 255, whereas after normalisation it ranges from 0 to 1.

1b

```
from google.colab import drive
drive.mount('/content/drive')
```

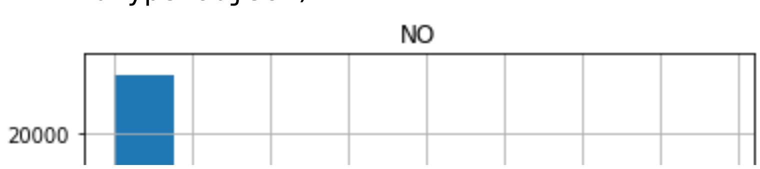
```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
```

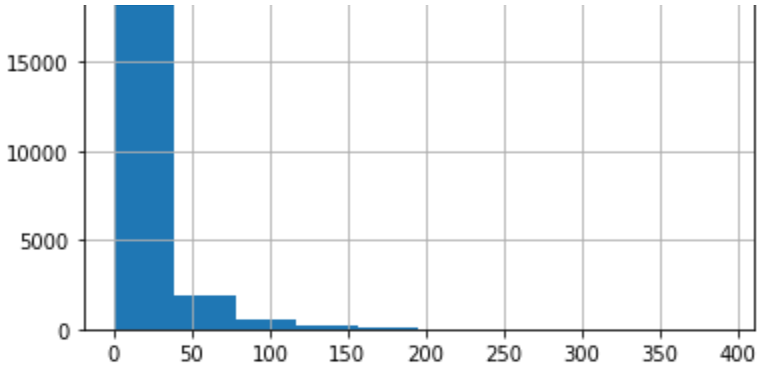
```
data = pd.read_csv("/content/drive/MyDrive/Datasets/city_day.csv")
data.head()
```

	City	Date	PM2.5	PM10	NO	NO2	NOx	NH3	CO	SO2	O3
0	Ahmedabad	2015-01-01	NaN	NaN	0.92	18.22	17.15	NaN	0.92	27.64	133.36
1	Ahmedabad	2015-01-02	NaN	NaN	0.97	15.69	16.46	NaN	0.97	24.55	34.06
2	Ahmedabad	2015-01-03	NaN	NaN	17.40	19.30	29.70	NaN	17.40	29.07	30.70
3	Ahmedabad	2015-01-04	NaN	NaN	1.70	18.48	17.97	NaN	1.70	18.59	36.08
4	Ahmedabad	2015-01-05	NaN	NaN	22.10	21.42	37.76	NaN	22.10	39.33	39.31

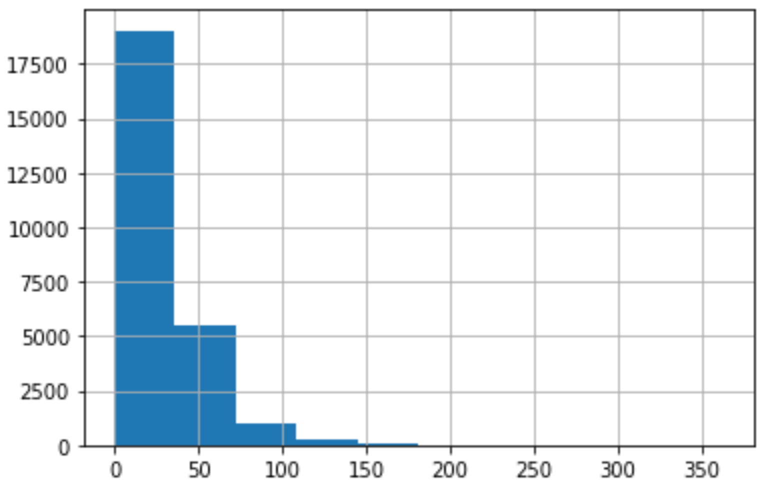
```
data.hist(column=["NO"])
data.hist(column=["NO2"])
data.hist(column=["NOx"])
data.hist(column=["CO"])
data.hist(column=["SO2"])
data.hist(column=["O3"])
data.hist(column=["Benzene"])
data.hist(column=["Toluene"])
data.hist(column=["Xylene"])
```

```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7f851a15f640>]],
      dtype=object)
```

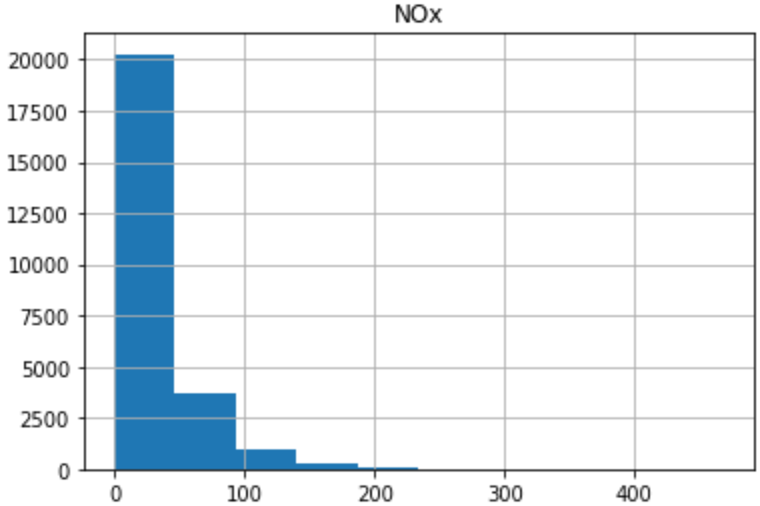




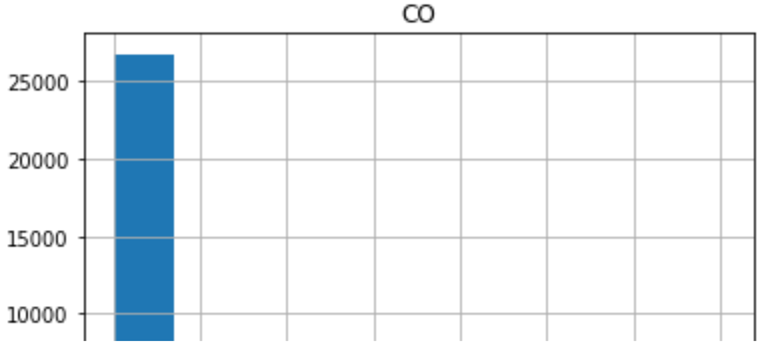
NO2



NOx



CO




```
data['Year'] = pd.to_datetime(data['Date']).dt.year
data_new = data[(data.Year >= 2015) & (data.Year <= 2018)]
print(data_new.describe())
```

```
              PM2.5         PM10           NO          NO2          NOx  \
count  13319.000000  7961.000000  14300.000000  14354.000000  14029.000000
mean      79.914880   137.369947     18.282866     31.070509     33.021959
std       74.004494   100.585837     23.589004     24.744464     33.528014
min        1.720000     0.010000      0.020000      0.010000      0.000000
25%       34.695000    69.710000      5.960000     13.900000     11.830000
50%       57.380000   107.770000     10.160000     24.575000     23.330000
75%       96.200000   170.790000     20.040000     40.890000     41.800000
max      949.990000   917.080000    287.140000    362.210000    467.630000

              NH3            CO           SO2            O3       Benzene  \
count  9141.000000  15905.000000  14100.000000  13995.000000  13619.000000
mean     28.458301      2.530362     14.435597     35.187022      2.585403
std      31.722477      7.686014     19.112619     23.047512      8.866921
min       0.010000      0.000000      0.040000      0.010000      0.000000
25%      10.390000      0.430000      5.170000     18.515000      0.090000
50%      20.090000      0.940000      8.390000     30.690000      0.850000
75%      35.480000      1.600000     14.842500     46.360000      2.940000
max     352.890000    175.810000    193.860000    257.730000    391.880000

           Toluene       Xylene           AQI          Year
count  13103.000000  6803.000000  13358.000000  17439.000000
mean       7.461200     2.990104    189.250262   2016.850393
std       14.545896     5.934221    152.539902      1.091172
min        0.000000     0.000000     13.000000   2015.000000
25%        0.435000     0.020000     91.000000   2016.000000
50%        2.570000     0.650000    136.000000   2017.000000
75%        7.950000     3.230000    253.000000   2018.000000
max      411.520000   125.180000   2049.000000   2018.000000
```


```
print("Number of null values in the dataset: ",(data_new.isnull().sum().sum()))
```

```
Number of null values in the dataset:  66801
```

```
#Normalization
for i in data.columns:
  if i not in ["City", "Date", "AQI_Bucket"]:
    data[i] = (data[i] - data[i].min()) / (data[i].max())
```

```
data
```

	City	Date	PM2.5	PM10	NO	NO2	NOx	N
0	Ahmedabad	2015-01-01	NaN	NaN	0.002304	0.050275	0.036674	N
1	Ahmedabad	2015-01-02	NaN	NaN	0.002432	0.043290	0.035199	N
2	Ahmedabad	2015-01-03	NaN	NaN	0.044487	0.053256	0.063512	N
3	Ahmedabad	2015-01-04	NaN	NaN	0.004300	0.050993	0.038428	N
4	Ahmedabad	2015-01-05	NaN	NaN	0.056517	0.059109	0.080748	N

...	...	...	...	...	...	...	...		
29526	Visakhapatnam	2020-06-27	0.015769	0.05093	0.019607	0.069159	0.041785	0.0353
29527	Visakhapatnam	2020-06-28	0.025621	0.07408	0.008703	0.071920	0.035348	0.0339
29528	Visakhapatnam	2020-06-29	0.024074	0.06572	0.008780	0.081500	0.039198	0.0303
29529	Visakhapatnam	2020-06-30	0.017474	0.04996	0.010315	0.080754	0.040203	0.0283
29530	Visakhapatnam	2020-07-01	0.015748	0.06599	0.000973	0.074101	0.030045	0.0147

29531 rows × 17 columns

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 29531 entries, 0 to 29530
Data columns (total 17 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   City        29531 non-null  object
 1   Date        29531 non-null  object
 2   PM2.5       24933 non-null  float64
 3   PM10        18391 non-null  float64
 4   NO          25949 non-null  float64
 5   NO2         25946 non-null  float64
 6   NOx         25346 non-null  float64
 7   NH3         19203 non-null  float64
 8   CO          27472 non-null  float64
 9   SO2         25677 non-null  float64
 10  O3          25509 non-null  float64
 11  Benzene     23908 non-null  float64
 12  Toluene     21490 non-null  float64
 13  Xylene      11422 non-null  float64
 14  AQI         24850 non-null  float64
 15  AQI_Bucket  24850 non-null  object
 16  Year        29531 non-null  float64
dtypes: float64(14), object(3)
memory usage: 3.8+ MB
```

3c: Linear Regression

```
import pandas as pd
from sklearn import datasets
from sklearn import linear_model
from sklearn import model_selection
from sklearn.model_selection import train_test_split
# Load the dataset
from sklearn.model_selection import KFold
diabetes = datasets.load_diabetes()
```

```
diabetes = datasets.load_diabetes()
from matplotlib import pyplot as plt
import numpy as np
```

```
diabetes = datasets.load_diabetes(as_frame=True)
```

```
print(type(diabetes['data']))
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
data = pd.read_csv("/content/drive/MyDrive/Datasets/diabetes-dataset.csv")
```

```
data
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	Diabete
0	2	138	62	35	0	33.6	
1	0	84	82	31	125	38.2	
2	0	145	0	0	0	44.2	
3	0	135	68	42	250	42.3	
4	1	139	62	41	480	40.7	
...	...	...	...	...	...	...	
1995	2	75	64	24	55	29.7	
1996	8	179	72	42	130	32.7	
1997	6	85	78	0	0	31.2	
1998	0	129	110	46	130	67.1	
1999	2	81	72	15	76	30.1	

2000 rows × 9 columns

```
df_max_scaled = data.copy()
```

```
# apply normalization techniques
for column in df_max_scaled.columns:
    df_max_scaled[column] = df_max_scaled[column]  / df_max_scaled[column].abs().ma
```

```
df_max_scaled
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	Dial
0	0.117647	0.693467	0.508197	0.318182	0.000000	0.416873	
1	0.000000	0.422111	0.672131	0.281818	0.168011	0.473945	

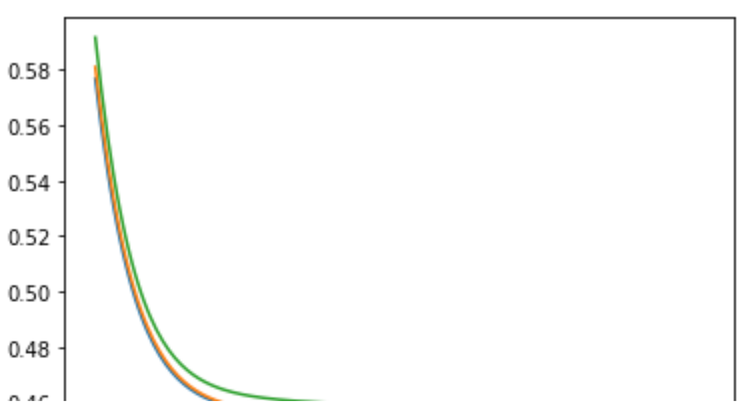
2	0.000000	0.728643	0.000000	0.000000	0.000000	0.548387
3	0.000000	0.678392	0.557377	0.381818	0.336022	0.524814
4	0.058824	0.698492	0.508197	0.372727	0.645161	0.504963
...	...	...	...	...	...	...
1995	0.117647	0.376884	0.524590	0.218182	0.073925	0.368486
1996	0.470588	0.899497	0.590164	0.381818	0.174731	0.405707
1997	0.352941	0.427136	0.639344	0.000000	0.000000	0.387097
1998	0.000000	0.648241	0.901639	0.418182	0.174731	0.832506
1999	0.117647	0.407035	0.590164	0.136364	0.102151	0.373449

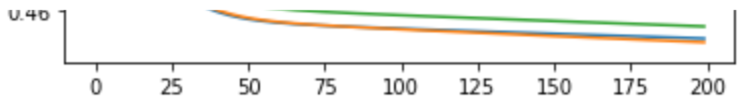
2000 rows × 9 columns

```
data = df_max_scaled
train, test = train_test_split(data, test_size=0.1)
```

```
import math
class Linear_Regression:
    # Initiliase the class with user defined learning rate and number of epochs
    def __init__(self, learning_rate, n_epochs):
        self.learning_rate = learning_rate
        self.n_epochs = n_epochs


    def gradient_descent(self,x,y):
        theta_curr = np.zeros(x.shape[1]) #weights
        bias = 0 #bias
        n_samples = y.size
        cost_l = [0] * self.n_epochs
        rmse = [0] * self.n_epochs

        for i in range(self.n_epochs):
            y_pred = x.dot(theta_curr) + bias

            #computing gradients
            dm = (-2/n_samples)*x.T.dot(y-y_pred)
            dc = (-2/n_samples)*np.sum(y-y_pred)

            #Updating weights
            theta_curr = theta_curr - self.learning_rate*dm
            bias = bias -  self.learning_rate*dc

            #Calculating the cost associated with each iteration
            cost = (1/(2*n_samples))*np.sum((y_pred-y)**2)
            cost_l[i] = cost
```

```
            cost_l[i] = cost
            mse = ((1/(n_samples))*np.sum((y_pred-y)**2))
            mse = math.sqrt(mse)
            rmse[i] = mse
        return theta_curr, bias, cost_l, rmse

    def predict(self,X,theta,bias):
        return X.dot(theta) + bias


model = Linear_Regression(learning_rate=0.01,n_epochs=200)
Y= train['Outcome']
X= train.drop(['Outcome'],axis=1)

X=X.to_numpy()
Y=Y.to_numpy()

kf = KFold(n_splits=3)

index =0
rmse_sum =[]
t_min=[]

for train_index, test_index in kf.split(train):
    X_train, X_test = X[train_index], X[test_index]
    Y_train, Y_test = Y[train_index], Y[test_index]

    #t: Theta obtained after gradient descent, b: Bias obtained after gradient desc
    t,b,c,rmse = model.gradient_descent(X_train,Y_train)
    y_pred_train = X_train.dot(t)+b
    y_pred = model.predict(X_test,t,b) #validation set
#     fig, axes = plt.subplots(1, 3)
#     axes[0].plot(rmse,'g')
    plt.plot(rmse)
    t_len = len(t)
    t_min.extend(t)
    rmse_sum.append(sum(rmse))
min_ind = rmse_sum.index(min(rmse_sum))
t_best = t_min[min_ind*t_len:(min_ind*t_len+t_len)]
```

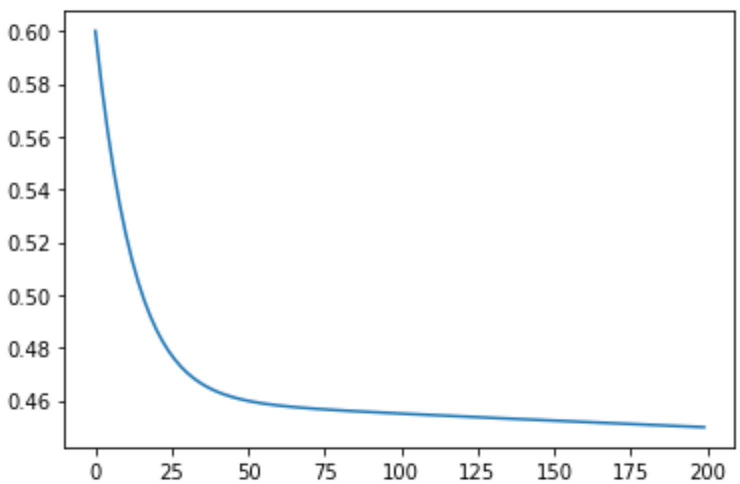




Performed 3 fold k cross validation to obtain better results

```
Y_test = test['Outcome']
X_test = test.drop(['Outcome'],axis=1)
Y_test = Y_test.to_numpy()
X_test = X_test.to_numpy()

y_pred_test = X_test.dot(t_best)

t,b,c,rmse = model.gradient_descent(X_test,Y_test)
y_pred_train = X_test.dot(t_best)+b
y_pred = model.predict(X_test,t_best,b)
plt.plot(rmse)

print("RMSE using Linear Regression from Scratch: ",min(rmse))
```

```
RMSE using Linear Regression from Scratch:  0.44993638716413276
```




```
from sklearn.linear_model import LinearRegression
from sklearn import metrics
regr = LinearRegression()
rmse_sci= []
t_reg = []
kfold_datasets_stored=[]

for train_index, test_index in kf.split(train):
    X_train, X_test = X[train_index], X[test_index]
    Y_train, Y_test = Y[train_index], Y[test_index]
    kfold_datasets_stored.append([X_train, X_test,Y_train, Y_test])
    regr = LinearRegression()
    regr.fit(X_train, Y_train)
    t_reg.extend(regr.coef_)
```

```
    y_pred = regr.predict(X_test)
    rmse_sci.append(np.sqrt(metrics.mean_squared_error(Y_test,y_pred)))

min_index = np.argmin(np.array(rmse_sci))
X_train_kf, X_test_kf, Y_train_kf, Y_test_kf= kfold_datasets_stored[min_index]
model=LinearRegression()
model.fit(X_train_kf,Y_train_kf)
y_pred_kf= model.predict(X_test_kf)
np.sqrt(metrics.mean_squared_error(Y_test_kf,y_pred_kf))
rmse_sci = np.sqrt(metrics.mean_squared_error(Y_test_kf,y_pred_kf))
print("RMSE using Linear Regression (SciKit Learn): ", rmse_sci)
```

```
RMSE using Linear Regression (SciKit Learn):  0.39157319306864125
```

RMSE using Linear Regression (SciKit Learn) is 0.3974252185078301

RMSE using Linear Regression from Scratch is 0.4552361135405836

Q4: Naive Bayes from Scratch

```
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split

# Load Iris dataset
iris = datasets.load_iris()
data1 = pd.DataFrame(data= np.c_[iris['data'], iris['target']],
                     columns= iris['feature_names'] + ['target'])
data1
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0.0
1	4.9	3.0	1.4	0.2	0.0
2	4.7	3.2	1.3	0.2	0.0
3	4.6	3.1	1.5	0.2	0.0
4	5.0	3.6	1.4	0.2	0.0
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	2.0
146	6.3	2.5	5.0	1.9	2.0
147	6.5	3.0	5.2	2.0	2.0

147	6.5	3.0	5.2	2.0	2.0
148	6.2	3.4	5.4	2.3	2.0
149	5.9	3.0	5.1	1.8	2.0

150 rows × 5 columns

```
X = data1.iloc[:, :4]
y = data1.iloc[:, 4:]
print(X.shape, y.shape)
print(X)
```

```
(150, 4) (150, 1)
     sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                  5.1               3.5                1.4               0.2
1                  4.9               3.0                1.4               0.2
2                  4.7               3.2                1.3               0.2
3                  4.6               3.1                1.5               0.2
4                  5.0               3.6                1.4               0.2
..                 ...               ...                ...               ...
145                6.7               3.0                5.2               2.3
146                6.3               2.5                5.0               1.9
147                6.5               3.0                5.2               2.0
148                6.2               3.4                5.4               2.3
149                5.9               3.0                5.1               1.8

[150 rows x 4 columns]
```

```
data1.describe()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
count	150.000000	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333	1.000000
std	0.828066	0.435866	1.765298	0.762238	0.819232
min	4.300000	2.000000	1.000000	0.100000	0.000000
25%	5.100000	2.800000	1.600000	0.300000	0.000000
50%	5.800000	3.000000	4.350000	1.300000	1.000000
75%	6.400000	3.300000	5.100000	1.800000	2.000000
max	7.900000	4.400000	6.900000	2.500000	2.000000

```
data1.dtypes
```

```
sepal length (cm)    float64
sepal width (cm)     float64
petal length (cm)    float64
petal width (cm)     float64
```

```
petal width (cm)      float64
target                float64
dtype: object
```

```
data1_notarget = data1.drop(['target'],axis=1)
```

```
data1.count()
```

```
sepal length (cm)    150
sepal width (cm)     150
petal length (cm)    150
petal width (cm)     150
target               150
dtype: int64
```

```
data1[(data1.max() - data1.min()).idxmax()]
```

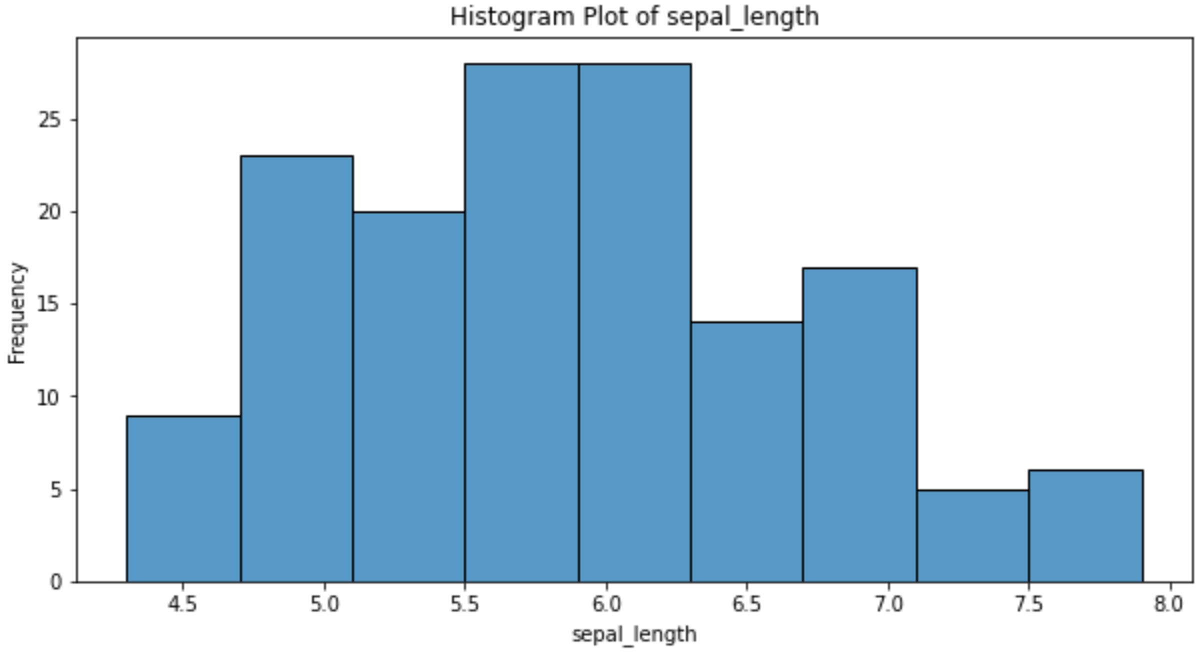
```
0      1.4
1      1.4
2      1.3
3      1.5
4      1.4
      ...
145    5.2
146    5.0
147    5.2
148    5.4
149    5.1
Name: petal length (cm), Length: 150, dtype: float64
```

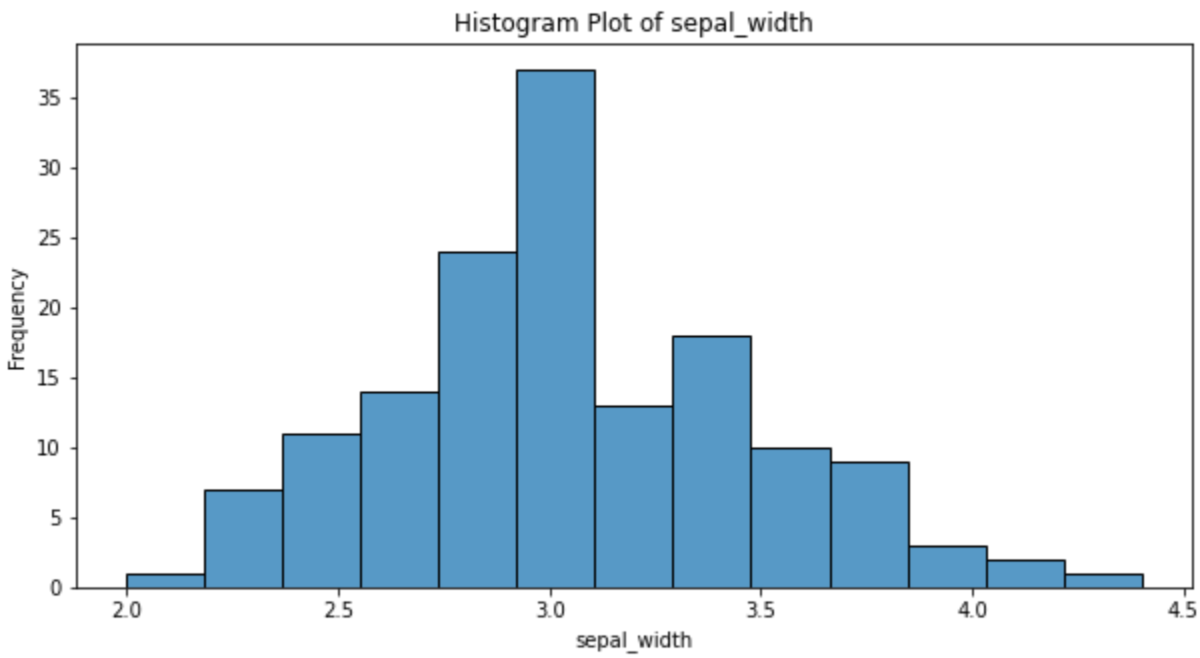
```
cols = ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width
for col in cols:
    print(data1[col].unique())
```

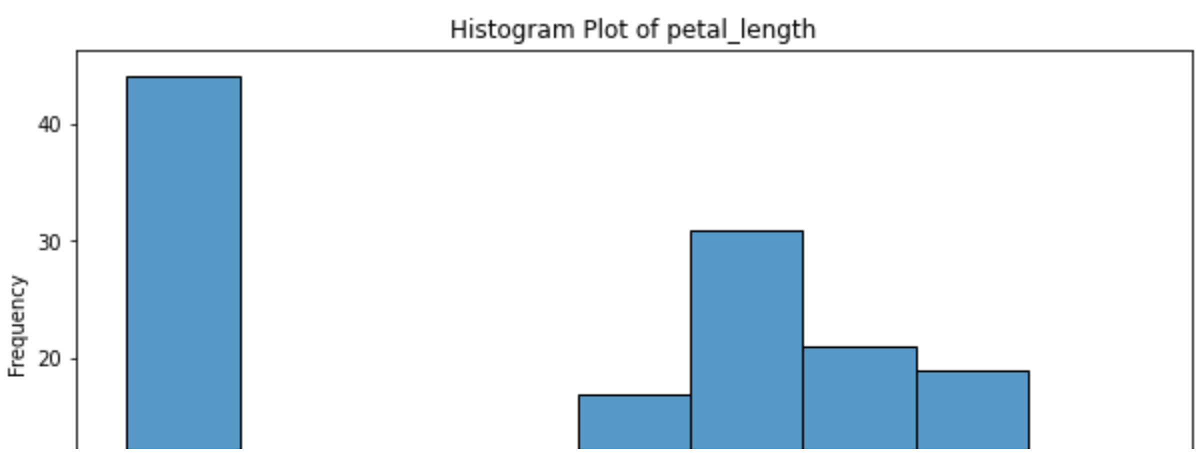
```
[5.1 4.9 4.7 4.6 5.  5.4 4.4 4.8 4.3 5.8 5.7 5.2 5.5 4.5 5.3 7.  6.4 6.9
 6.5 6.3 6.6 5.9 6.  6.1 5.6 6.7 6.2 6.8 7.1 7.6 7.3 7.2 7.7 7.4 7.9]
[3.5 3.  3.2 3.1 3.6 3.9 3.4 2.9 3.7 4.  4.4 3.8 3.3 4.1 4.2 2.3 2.8 2.4
 2.7 2.  2.2 2.5 2.6]
[1.4 1.3 1.5 1.7 1.6 1.1 1.2 1.  1.9 4.7 4.5 4.9 4.  4.6 3.3 3.9 3.5 4.2
 3.6 4.4 4.1 4.8 4.3 5.  3.8 3.7 5.1 3.  6.  5.9 5.6 5.8 6.6 6.3 6.1 5.3
 5.5 6.7 6.9 5.7 6.4 5.4 5.2]
[0.2 0.4 0.3 0.1 0.5 0.6 1.4 1.5 1.3 1.6 1.  1.1 1.8 1.2 1.7 2.5 1.9 2.1
 2.2 2.  2.4 2.3]
```

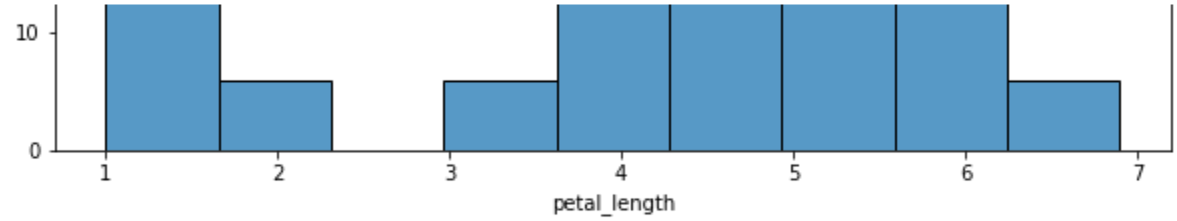
```
iris = sns.load_dataset("iris")

# Plotting histograms for each column
for col in iris.columns:
    plt.figure(figsize=(10, 5))
    sns.histplot(iris, x=col, kde=False)
```

```
plt.xlabel(col)
plt.ylabel('Frequency')
plt.title(f'Histogram Plot of {col}')
```










```
(data1.columns)
```

```
Index(['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
       'petal width (cm)', 'target'],
      dtype='object')
```

```
#Normalization
data1['sepal length (cm)']= data1['sepal length (cm)']/(data1['sepal length (cm)'].
data1['sepal width (cm)']= data1['sepal width (cm)']/(data1['sepal width (cm)'].ma›
data1['petal length (cm)']= data1['petal length (cm)']/(data1['petal length (cm)'].
data1['petal width (cm)']= data1['petal width (cm)']/(data1['petal width (cm)'].ma›
```

```
data1
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	0.645570	0.795455	0.202899	0.08	0.0
1	0.620253	0.681818	0.202899	0.08	0.0
2	0.594937	0.727273	0.188406	0.08	0.0
3	0.582278	0.704545	0.217391	0.08	0.0
4	0.632911	0.818182	0.202899	0.08	0.0
...	...	...	...	...	...
145	0.848101	0.681818	0.753623	0.92	2.0
146	0.797468	0.568182	0.724638	0.76	2.0
147	0.822785	0.681818	0.753623	0.80	2.0
148	0.784810	0.772727	0.782609	0.92	2.0
149	0.746835	0.681818	0.739130	0.72	2.0

150 rows × 5 columns

```
iris = datasets.load_iris()
```

```
X = iris.data
y = iris.target
```

```
# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Calculate class probabilities
class_probs = np.zeros((3, X.shape[1]))
for i in range(3):
    class_probs[i,:] = np.mean(X_train[y_train==i], axis=0)

# Calculate class priors
class_priors = np.zeros(3)
for i in range(3):
    class_priors[i] = np.mean(y_train==i)

# Define predict function
def predict(X, class_probs, class_priors):
    probs = np.zeros((X.shape[0], 3))
    for i in range(3):
        probs[:,i] = class_priors[i] * np.prod(np.power(class_probs[i,:], X), axis=
    return np.argmax(probs, axis=1)

# Predict class labels on test data
y_pred = predict(X_test, class_probs, class_priors)

# Calculate accuracy
accuracy = np.mean(y_pred==y_test)
print("Accuracy:", accuracy)
```

```
Accuracy: 0.8777777777777778
```

Q4: Naive Bayes using SK-Learn

```
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB

# Load Iris dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Train Naive Bayes classifier
clf = GaussianNB()
clf.fit(X_train, y_train)

# Predict class labels on test data
```

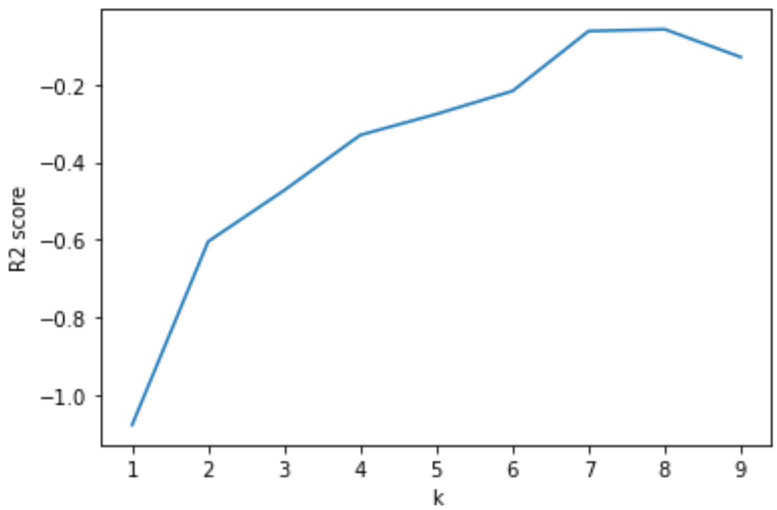
```
y_pred = clf.predict(X_test)

# Calculate accuracy
accuracy = np.mean(y_pred==y_test)
print("Accuracy:", accuracy)
```

```
Accuracy: 1.0
```

Q5: KNN

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

df = pd.read_csv("/content/drive/MyDrive/Datasets/bmd.csv") # Load the data


# Split the data into training and testing sets
X = df["age"].values.reshape(-1, 1)
y = df["bmd"].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Calculating the Euclidean distance between two points
def euclidean_distance(x1, x2):
    return np.sqrt(np.sum((x1 - x2)**2))

# Define a function to fit KNN Regressor
def knn_regressor(X_train, y_train, X_test, k):
    y_pred = []
    for x_test in X_test:
        distances = []
        for i, j in zip(X_train, y_train):
            distance = euclidean_distance(x_test, i)
            distances.append((distance, j))
        distances = sorted(distances, key=lambda x: x[0])
        k_neighbors = distances[:k]
        k_neighbors_y = [neighbor[1] for neighbor in k_neighbors]
        y_pred.append(np.mean(k_neighbors_y))
    return np.array(y_pred)

# Define a function to calculate R2 score
def r2_score(y_true, y_pred):
    mean_y_true = np.mean(y_true)
    sse = np.sum((y_true - y_pred)**2)
    sst = np.sum((y_true - mean_y_true)**2)
    return 1 - sse/sst
```

```
# Plot the results for different values of k
r2_scores = []
k_values = [1,2,3,4,5,6,7,8,9]
for k in k_values:
    y_pred = knn_regressor(X_train, y_train, X_test, k)
    r2 = r2_score(y_test, y_pred)
    r2_scores.append(r2)

plt.plot(k_values, r2_scores)
plt.xlabel("k")
plt.ylabel("R2 score")
plt.show()
```




```
ind = np.argmax(r2_scores, axis=None, out=None)
print("Best value of k: ", k_values[ind])
print("Corresponding R2 score: ", r2_scores[ind])
```

```
Best value of k:  8
Corresponding R2 score:  -0.05781999422048245
```