Problem 1 report

In [1]:

```
import numpy as np
import cv2
import os
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from PIL import Image
from numpy.linalg import eig
from mpl_toolkits.mplot3d import Axes3D
```

Load and visualize data

load_images():
Each Image is of dimension 256. As data is 2-D we have 256x256 dimension for each image. Now we convert 2-D data to 1-D. Then image of 256x256 size is now converted into 1-D data of size 1x65536. Calculating covariance of large dimension takes more time. So lets resize the image. Resizing image do not lose information about image. So 2-D, 256x256 image is reshaped into 2-D data of 100x100 dimension. Now lets convert this 100x100 dimension data into 1-D data of size 1x10000. load_images does the above functionality.

Pre processing

pre_processing():
Pre processing technique used i this problem is COLUMN STANDARDIZATION.
Let out data set is of dimension mxn. where m=no. of datapoints, n=no. of dimensions/features.
A(i,j) be data value of ith images and jth dimension.
Now replace it with
$$\frac{(A(i,j) - mean(j))}{ standard\_deviation(j)}$$

mean(j) represent mean of jth column
standard_deviation(j) represent standard deviation of jth column
Advantages of standardization
On performing standardization, mean of every column becomes 0 and standard deviation of every column becomes 1. We will see how this will be helpful while calculating covariance of matrix. pre_processing method does this job in our function.

Compute covariance

covariance between any two column is defined as follows:
$$COV(X,Y) = \frac{\sum_{i=1}^n (X_i - mean_X)(Y_i - mean_Y)}{ n}$$
Now lets see how standardization is helping us in calculating covariance.
After standardization mean of a column will be 0.
therefore mean will be zero.
Now covariance is calculated as follows
$$COV(X,Y) = \frac{\sum_{i=1}^n (X_i - 0)(Y_i - 0)}{ n}$$
It is equivalent to
$$COV(X,Y) = \frac{\sum_{i=1}^n (X_i)(Y_i)}{ n}$$
where n=no of data samples
There we can get covariance by matrix multiplication. compute_covariance() method does this job in our function

In [2]:

```
class ques_1:
    def load_images(self,path):
        self.images=[]
        for filename in os.listdir(path):
            img = cv2.imread(path+"/"+filename,0)
            dim=(100,100)
```

```
            dim=(100,100)
            img=cv2.resize(img, dim, interpolation = cv2.INTER_AREA)
            self.images.append(np.array(img).ravel())
    def pre_processing(self):
        self.standardized_data = StandardScaler().fit_transform(self.images)
        return self.standardized_data

    def compute_covariance(self):
        self.covar_matrix = np.matmul(self.standardized_data.T , self.standardized_data)

    def get_eigens(self):
        self.values, self.vectors = eig(self.covar_matrix)
        return self.values, self.vectors
```

In [3]:

```
obj1=ques_1()
obj1.load_images("./dataset")
standardized_data=obj1.pre_processing()
obj1.compute_covariance()
eigen_values,eigen_vectors=obj1.get_eigens()
```

Reconstruct image using small components

In [4]:

```
def reduce_dimen(standardized_data,eigen_vectors,reduce_dim):
    reduce_eigen_vectors=eigen_vectors[:,:reduce_dim]
    reduce_data=standardized_data.dot(reduce_eigen_vectors)
    return reduce_eigen_vectors,reduce_data

def reconstruct_image(reduce_eigen_vectors,reduce_data):
    reconstructed_data=reduce_data.dot(reduce_eigen_vectors.T)
    reconstructed_data=reconstructed_data.real
    return reconstructed_data
```

In [6]:

```
reduce_eigen_vectors,reduce_data=reduce_dimen(standardized_data,eigen_vectors,500)
reconstructed_data=reconstruct_image(reduce_eigen_vectors,reduce_data)
```

The below function will display images in new window

In [7]:

```
for i in range(5):
    cv2.imshow('image',reconstructed_data[i].reshape(100,100))
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

How to identify % of data conserved due to PCA

For D-dimensional data we get, D eigen values and D eigen vectors(each vector of size D).
Let us suppose we have compressed D-dimension to z-dimension using PCA.
Now percentage of data conserved after reducing to z-dimension is
$$\frac{\sum_{i=1}^z (eigen\_value_i)}{ \sum_{i=1}^D (eigen\_value_i)}$$
$$error=1-\frac{\sum_{i=1}^z (eigen\_value_i)}{ \sum_{i=1}^D (eigen\_value_i)}$$

Let us plot a graph with different PCA

In [8]:

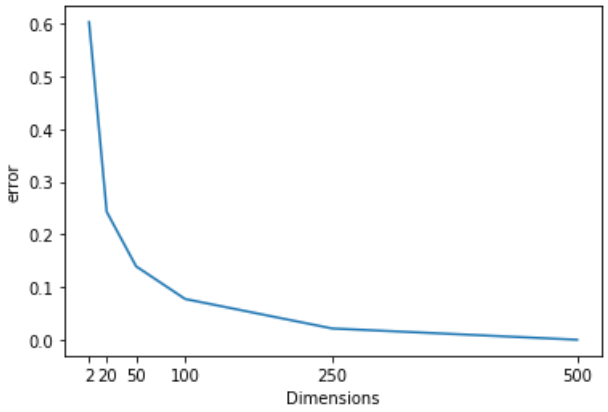
```
pca_values=[2,20,50,100,250,500]
error=[]
for i in range(0,len(pca_values)):
    e=np.sum(eigen_values[:pca_values[i]])/np.sum(eigen_values)
    error.append(1-e.real)
    #print(e)
```

In [9]:

```
#for i in range(len(error)):
plt.plot(pca_values,error)
plt.xticks(pca_values)
plt.xlabel('Dimensions')
plt.ylabel('error')
```

Out[9]:

```
Text(0, 0.5, 'error')
```



Decide "N" such that error is 20%

reducing to 28 dimension we get 20% error

In [10]:

```
e=np.sum(eigen_values[:28])/np.sum(eigen_values)
print(1-e.real)
```

```
0.20310728139125833
```

Labels of images

In [11]:

```
path="./dataset"
files=os.listdir(path)
labels=[int(i.split("_")[0]) for i in files]
labels=np.array(labels)
print(labels.shape)
```

```
(520,)
```

Reduce to 1-Dimension

Reduce images to 1-D

In [12]:

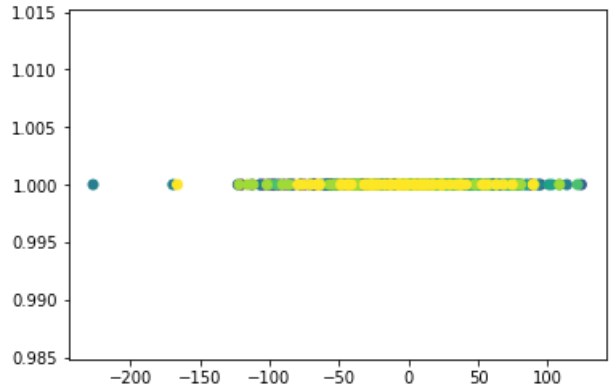
```
reduce_eigen_vectors,reduce_data=reduce_dimen(standardized_data,eigen_vectors,1)
```

```
reduce_data=reduce_data.real
print(reduce_data.shape)
```

```
(520, 1)
```

In [13]:

```
plt.scatter(reduce_data[:,0],520*[1],c=labels)
plt.show()
```



Reduce to 2-Dimension

Reduce images to 2-D

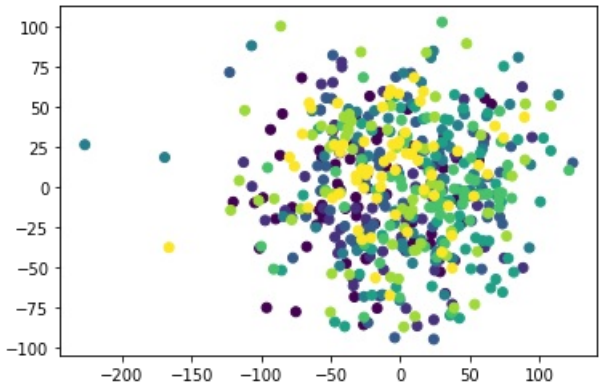
In [14]:

```
reduce_eigen_vectors,reduce_data=reduce_dimen(standardized_data,eigen_vectors,2)
reduce_data=reduce_data.real
print(reduce_data.shape)
```

```
(520, 2)
```

In [15]:

```
plt.scatter(reduce_data[:,0],reduce_data[:,1],c=labels)
plt.show()
```



Reduce to 3-Dimension

Reduce images to 3-D

In [16]:

```
reduce_eigen_vectors,reduce_data=reduce_dimen(standardized_data,eigen_vectors,3)
```

```
reduce_data=reduce_data.real
print(reduce_data.shape)
```

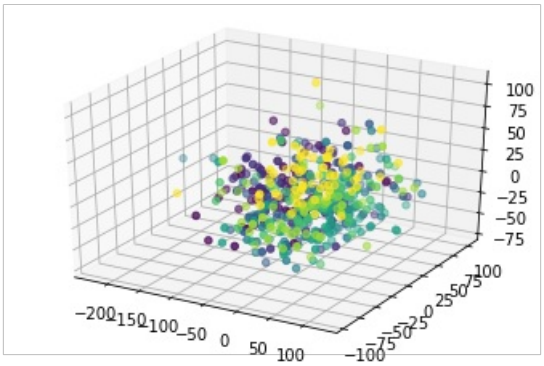
```
(520, 3)
```

In [19]:

```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(reduce_data[:,0],reduce_data[:,1],reduce_data[:,2],c=labels)
```

Out[19]:

```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x2a050e344c8>
```



In [22]:

```
!jupyter nbconvert --to html q1.ipynb
```

```
[NbConvertApp] Converting notebook q1.ipynb to html
[NbConvertApp] Writing 465530 bytes to q1.html
```

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```

In [75]:

```
import numpy as np
import pandas as pd
import seaborn as sns
import cv2
import os
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from PIL import Image
from numpy.linalg import eig
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report as cr
```

In [2]:

```
class PCA:
    def __init__(self,n_components=10):
        self.n_components = n_components

    def load_images(self,path):
        images=[]
        for filename in os.listdir(path):
            img = cv2.imread(path+"/"+filename,0)
            dim=(100,100)
            img=cv2.resize(img, dim, interpolation = cv2.INTER_AREA)
            images.append(np.array(img).ravel())
        images=np.array(images)
        return images

    def get_labels(self,path):
        files=os.listdir(path)
        labels=[int(i.split("_")[0]) for i in files]
        labels=np.array(labels)
        return labels

    def compute_mean(self,image_data):
        mean=np.mean(image_data.T, axis=1)
        return mean

    def compute_covariance(self,image_data, mean):
        return np.cov((image_data-mean).T)

    def get_eigens(self,covar_matrix):
        return np.linalg.eig(covar_matrix)

    def sort_eigens(self,eigen_values,eigen_vectors):
        index = eigen_values.argsort()[::-1]
        eigen_values = eigen_values[index]
        eigen_vectors = eigen_vectors[:,index]
        return eigen_values, eigen_vectors

    def reduce_dim(self,image_data, eigen_values, eigen_vectors):
        new_data = eigen_vectors[:,:self.n_components]
        return image_data.dot(new_data)

    def pca_(self,path):
        image_data=self.load_images(path)
        labels=self.get_labels(path)
        mean=self.compute_mean(image_data)
        covariance=self.compute_covariance(image_data, mean)
        eigen_values,eigen_vectors=self.get_eigens(covariance)
        eigen_values,eigen_vectors=self.sort_eigens(eigen_values,eigen_vectors)
        reduced_data=self.reduce_dim(image_data, eigen_values, eigen_vectors)
        return reduced_data,labels
```

In [3]:

```
obj1=PCA(n_components=500)
PCA_data,labels=obj1.pca_("./dataset")
```

reduced to 500 dimensions

In [50]:

```
print(PCA_data.shape)
```

```
(520, 500)
```

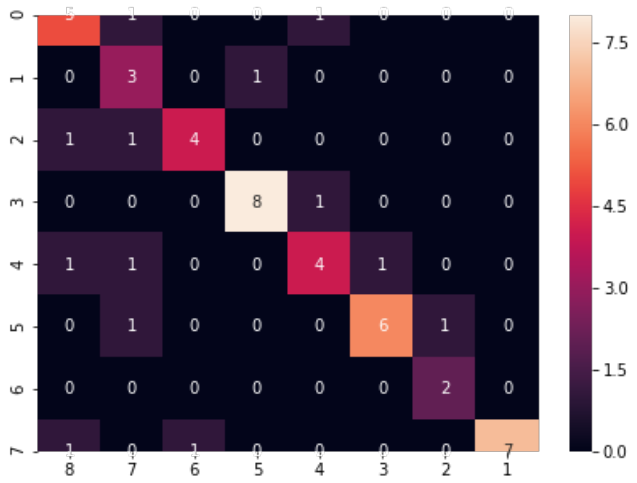
In [83]:

```
class logistic_regression:
    def split_data(self, PCA_data,labels):
        X_train, X_test, y_train, y_test = train_test_split(PCA_data, labels, test_size =
0.1,random_state=100)
        return X_train,X_test, y_train, y_test

    def sigmoid(self,z):
        return 1/(1+np.exp(-z))

    def costFunction(self,theta, x, y):
        m = len(y)
        h = self.sigmoid(x @ theta)
        gradient = 1/m * ((y-h) @ x)
        return gradient

    def fit(self,x, y, max_iter=5000, alpha=0.0001):
        x = np.insert(x, 0, 1, axis=1)
        thetas = []
        classes = np.unique(y)
        for c in classes:
            binary_y = np.where(y == c, 1, 0)
            theta = np.zeros(x.shape[1])
            for epoch in range(max_iter):
                grad = self.costFunction(theta, x, binary_y)
                theta = theta + alpha * grad
            thetas.append(theta)
        return thetas, classes

    def predict(self,classes, thetas, x):
        x = np.insert(x, 0, 1, axis=1)
        preds = [np.argmax(
            [self.sigmoid(xi @ theta) for theta in thetas]) for xi in x]
        return [classes[p] for p in preds]

    def log_reg(self,PCA_data,labels):
        X_train,X_test, y_train, y_test=self.split_data(PCA_data,labels)
        thetas, classes = self.fit(X_train, y_train, 3000)
        pred = self.predict(classes, thetas, X_test)

        cm=confusion_matrix(y_test, pred)
        df_cm = pd.DataFrame(cm, index = [i for i in "01234567"],
                             columns = [i for i in "87654321"])
        plt.figure(figsize = (7,5))
        sns.heatmap(df_cm, annot=True)
        plt.show()
        print("*************************************************************************")
        print(cr(y_test , pred))
        print("*************************************************************************")
        print("ACCURACY SCORE",end=" ")
        print(accuracy_score(y_test, pred))
```

In [84]:

```
obj2=logistic_regression()
obj2.log_reg(PCA_data,labels)
```

```
D:\Anaconda3\lib\site-packages\ipykernel_launcher.py:7: RuntimeWarning: overflow encountered in ex
p
  import sys
```




```
**************************************************************************
              precision    recall  f1-score   support

           0       0.62      0.71      0.67         7
           1       0.43      0.75      0.55         4
           2       0.80      0.67      0.73         6
           3       0.89      0.89      0.89         9
           4       0.67      0.57      0.62         7
           5       0.86      0.75      0.80         8
           6       0.67      1.00      0.80         2
           7       1.00      0.78      0.88         9

    accuracy                           0.75        52
   macro avg       0.74      0.76      0.74        52
weighted avg       0.78      0.75      0.76        52

**************************************************************************
ACCURACY SCORE 0.75
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [1]:

```
import torch
import torchvision
from torchvision import datasets
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
import torch.nn as nn
import torch.nn.functional as F
```

MLP

Reading data

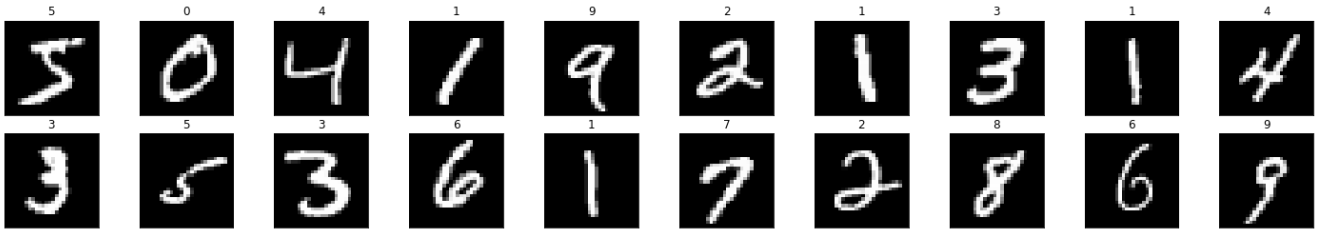
In [2]:

```
num_workers = 0
batch_size = 20
transform = transforms.ToTensor()
train_data = datasets.MNIST(root='data', train=True,
                                  download=True, transform=transform)
test_data = datasets.MNIST(root='data', train=False,
                                 download=True, transform=transform)

trainloader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
    num_workers=num_workers)
valloader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
    num_workers=num_workers)
```

Lets print some datapoints

In [3]:

```
dataiter = iter(trainloader)
images, labels = dataiter.next()
images = images.numpy()
fig = plt.figure(figsize=(25, 4))
for idx in np.arange(20):
    ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(images[idx]), cmap='gray')
    ax.set_title(str(labels[idx].item()))
```



In [4]:

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(28 * 28, 512)
        self.fc2 = nn.Linear(512, 512)
        self.fc3 = nn.Linear(512,10)
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        # flatten image input
        x = x.view(-1, 28 * 28)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
```

```
        x = F.relu(self.fc2(x))
        x=self.dropout(x)
        x = self.fc3(x)
        return x

# initialize the NN
model = Net()
print(model)
```

```
Net(
  (fc1): Linear(in_features=784, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=512, bias=True)
  (fc3): Linear(in_features=512, out_features=10, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

In [5]:

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
```

In [6]:

```
n_epochs = 30
model.train()
for epoch in range(n_epochs):
    train_loss = 0.0
    for data, target in trainloader:
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()*data.size(0)
    train_loss = train_loss/len(trainloader.dataset)

    print('Epoch: {} \tTraining Loss: {:.6f}'.format(
        epoch+1,
        train_loss
        ))
```

```
Epoch: 1  Training Loss: 0.287735
Epoch: 2  Training Loss: 0.113045
Epoch: 3  Training Loss: 0.080664
Epoch: 4  Training Loss: 0.062050
Epoch: 5  Training Loss: 0.051822
Epoch: 6  Training Loss: 0.042769
Epoch: 7  Training Loss: 0.037185
Epoch: 8  Training Loss: 0.032919
Epoch: 9  Training Loss: 0.030124
Epoch: 10  Training Loss: 0.025077
Epoch: 11  Training Loss: 0.022016
Epoch: 12  Training Loss: 0.021061
Epoch: 13  Training Loss: 0.018331
Epoch: 14  Training Loss: 0.017085
Epoch: 15  Training Loss: 0.015932
Epoch: 16  Training Loss: 0.016664
Epoch: 17  Training Loss: 0.014782
Epoch: 18  Training Loss: 0.012696
Epoch: 19  Training Loss: 0.011001
Epoch: 20  Training Loss: 0.010268
Epoch: 21  Training Loss: 0.011479
Epoch: 22  Training Loss: 0.009017
Epoch: 23  Training Loss: 0.009180
Epoch: 24  Training Loss: 0.009368
Epoch: 25  Training Loss: 0.009204
Epoch: 26  Training Loss: 0.008185
Epoch: 27  Training Loss: 0.008023
Epoch: 28  Training Loss: 0.006767
Epoch: 29  Training Loss: 0.006547
Epoch: 30  Training Loss: 0.006938
```

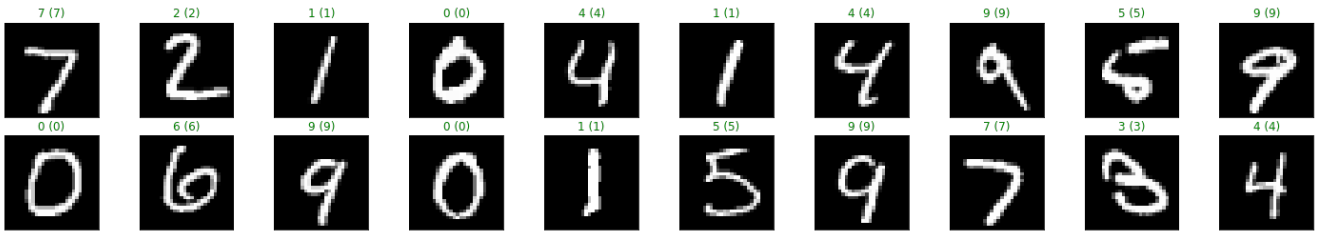
In [7]:

```
test_loss = 0.0
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))

model.eval() # prep model for *evaluation*

for data, target in valloader:
    output = model(data)
    loss = criterion(output, target)
    test_loss += loss.item()*data.size(0)
    _, pred = torch.max(output, 1)
    correct = np.squeeze(pred.eq(target.data.view_as(pred)))
    for i in range(batch_size):

        label = target.data[i]
        class_correct[label] += correct[i].item()
        class_total[label] += 1

test_loss = test_loss/len(valloader.dataset)
print('Test Loss: {:.6f}\n'.format(test_loss))

for i in range(10):
    if class_total[i] > 0:
        print('Test Accuracy of %5s: %2d%% (%2d/%2d)' % (
            str(i), 100 * class_correct[i] / class_total[i],
            np.sum(class_correct[i]), np.sum(class_total[i])))
    else:
        print('Test Accuracy of %5s: N/A (no training examples)' % (classes[i]))

print('\nTest Accuracy (Overall): %2d%% (%2d/%2d)' % (
    100. * np.sum(class_correct) / np.sum(class_total),
    np.sum(class_correct), np.sum(class_total)))
```

```
Test Loss: 0.067199

Test Accuracy of     0: 99% (975/980)
Test Accuracy of     1: 99% (1133/1135)
Test Accuracy of     2: 98% (1018/1032)
Test Accuracy of     3: 97% (989/1010)
Test Accuracy of     4: 97% (961/982)
Test Accuracy of     5: 99% (885/892)
Test Accuracy of     6: 98% (947/958)
Test Accuracy of     7: 98% (1008/1028)
Test Accuracy of     8: 98% (958/974)
Test Accuracy of     9: 97% (988/1009)

Test Accuracy (Overall): 98% (9862/10000)
```

In [8]:

```
dataiter = iter(valloader)
images, labels = dataiter.next()

output = model(images)
_, preds = torch.max(output, 1)
images = images.numpy()

fig = plt.figure(figsize=(25, 4))
for idx in np.arange(20):
    ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(images[idx]), cmap='gray')
    ax.set_title("{} ({})".format(str(preds[idx].item()), str(labels[idx].item())),
                 color=("green" if preds[idx]==labels[idx] else "red"))
```



In [9]:

```
!pip install python-mnist
```

```
Requirement already satisfied: python-mnist in d:\anaconda3\lib\site-packages (0.7)
```

CNN

In [11]:

```
from mnist import MNIST
from torch.utils.data import DataLoader, Dataset, TensorDataset
mnist = MNIST()
X_train,y_train = mnist.load_training()
X_test, y_test = mnist.load_testing()
X_train = torch.FloatTensor(X_train)
X_train = X_train.to(dtype=torch.float32)
X_train = X_train.reshape(X_train.size(0), -1)
X_train = X_train/128
y_train = torch.FloatTensor(y_train)
y_train = y_train.to(dtype=torch.long)
bs = 4
X_test = torch.FloatTensor(X_test)
X_test = X_test.to(dtype=torch.float32)
X_test = X_test.reshape(X_test.size(0), -1)
X_test = X_test/128
y_test = torch.FloatTensor(y_test)
y_test = y_test.to(dtype=torch.long)
train_ds = TensorDataset(X_train, y_train)
train_dl = DataLoader(train_ds, batch_size=bs, drop_last=False, shuffle=True)

test_ds = TensorDataset(X_test, y_test)
test_dl = DataLoader(test_ds, batch_size=bs * 2)

loaders={}
loaders['train'] = train_dl
loaders['test'] = test_dl
```

In [12]:

```
import torch.nn as nn
import torch.nn.functional as F
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 24, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(24, 48, 5)
        self.fc1 = nn.Linear(48 * 4 * 4, 64)
        self.fc2 = nn.Linear(64, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 48 * 4 * 4)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
```

In [13]:

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

In [14]:

```
import torch
for epoch in range(10):
    running_loss = 0.0
    for i, (data,target) in enumerate(loaders['train']):
        optimizer.zero_grad()
        data = data.view(-1, 1, 28, 28)
        outputs = net(data)

        loss = criterion(outputs, target)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
```

```
[1,  2000] loss: 0.762
[1,  4000] loss: 0.191
[1,  6000] loss: 0.143
[1,  8000] loss: 0.121
[1, 10000] loss: 0.100
[1, 12000] loss: 0.094
[1, 14000] loss: 0.080
[2,  2000] loss: 0.065
[2,  4000] loss: 0.065
[2,  6000] loss: 0.053
[2,  8000] loss: 0.052
[2, 10000] loss: 0.052
[2, 12000] loss: 0.047
[2, 14000] loss: 0.047
[3,  2000] loss: 0.036
[3,  4000] loss: 0.035
[3,  6000] loss: 0.042
[3,  8000] loss: 0.036
[3, 10000] loss: 0.039
[3, 12000] loss: 0.035
[3, 14000] loss: 0.040
[4,  2000] loss: 0.028
[4,  4000] loss: 0.022
[4,  6000] loss: 0.022
[4,  8000] loss: 0.029
[4, 10000] loss: 0.035
[4, 12000] loss: 0.026
[4, 14000] loss: 0.030
[5,  2000] loss: 0.021
[5,  4000] loss: 0.023
[5,  6000] loss: 0.023
[5,  8000] loss: 0.024
[5, 10000] loss: 0.025
[5, 12000] loss: 0.020
[5, 14000] loss: 0.021
[6,  2000] loss: 0.015
[6,  4000] loss: 0.019
[6,  6000] loss: 0.016
[6,  8000] loss: 0.018
[6, 10000] loss: 0.021
[6, 12000] loss: 0.022
[6, 14000] loss: 0.021
[7,  2000] loss: 0.010
[7,  4000] loss: 0.013
[7,  6000] loss: 0.015
[7,  8000] loss: 0.012
[7, 10000] loss: 0.018
[7, 12000] loss: 0.013
[7, 14000] loss: 0.019
[8,  2000] loss: 0.016
[8,  4000] loss: 0.007
```

```
[8,  6000] loss: 0.013
[8,  8000] loss: 0.010
[8, 10000] loss: 0.010
[8, 12000] loss: 0.012
[8, 14000] loss: 0.013
[9,  2000] loss: 0.005
[9,  4000] loss: 0.012
[9,  6000] loss: 0.011
[9,  8000] loss: 0.010
[9, 10000] loss: 0.006
[9, 12000] loss: 0.011
[9, 14000] loss: 0.015
[10,  2000] loss: 0.007
[10,  4000] loss: 0.005
[10,  6000] loss: 0.009
[10,  8000] loss: 0.008
[10, 10000] loss: 0.007
[10, 12000] loss: 0.013
[10, 14000] loss: 0.009
Finished Training
```

In [15]:

```
PATH = './mnist_net.pth'
torch.save(net.state_dict(), PATH)
```

In [16]:

```
dataiter = iter(loaders['test'])
images, labels = dataiter.next()
```

In [17]:

```
net = Net()
net.load_state_dict(torch.load(PATH))
```

Out[17]:

```
<All keys matched successfully>
```

In [18]:

```
images = images.view(-1, 1, 28, 28)
outputs = net(images)
```

In [19]:

```
correct = 0
total = 0
with torch.no_grad():
    for data in loaders['test']:
        images, labels = data
        images = images.view(-1,1,28,28)
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: {} %'.format(
    100 * correct / total))
```

```
Accuracy of the network on the 10000 test images: 99.11 %
```

SVM

In [20]:

```
from mnist import MNIST
```

```
import numpy as np
mnist = MNIST()
X_train,y_train = mnist.load_training()
X_test, y_test = mnist.load_testing()
```

In [21]:

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)
```

In [22]:

```
from sklearn.svm import SVC
svm  = SVC(C=10)
svm.fit(X_train, y_train)
predictions = svm.predict(X_test)
```

```
D:\Anaconda3\lib\site-packages\sklearn\svm\base.py:193: FutureWarning: The default value of gamma
will change from 'auto' to 'scale' in version 0.22 to account better for unscaled features. Set ga
mma explicitly to 'auto' or 'scale' to avoid this warning.
  "avoid this warning.", FutureWarning)
```

In [23]:

```
from sklearn.metrics import accuracy_score
accuracy_score(np.array(y_test), np.array(predictions))
```

Out[23]:

```
0.9614
```

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```

In [1]:

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
```

In [2]:

```
df = pd.read_csv('./datasets/household_power_consumption.txt', sep=';',
                 usecols=[2],na_values=['?','nan'], low_memory = False)
l = df[df['Global_active_power'].isnull()].index.tolist()
```

In [3]:

```
df.head()
```

Out[3]:

	Global_active_power
0	4.216
1	5.360
2	5.374
3	5.388
4	3.666

In [4]:

```
data = df.values
data
```

Out[4]:

```
array([[4.216],
       [5.36 ],
       [5.374],
       ...,
       [0.938],
       [0.934],
       [0.932]])
```

In [5]:

```
data = [j for sub in data for j in sub]
data = [i.tolist() for i in data]
np.shape(data)
```

Out[5]:

```
(2075259,)
```

In [6]:

```
trainX = list()
trainY = list()
for i in range(len(data)-60):
    data_check = np.isnan(data[i:i+61])
    if not np.sum(data_check):
        trainX.append(data[i:i+60])
        trainY.append(data[i+60])
trainX = np.array(trainX, dtype='float32')
trainY = np.array(trainY, dtype='float32')
```

```
trainY = np.array(trainY, dtype='float32')
```

In [7]:

```
train_sample , test_sample = train_test_split(np.array(range(trainX.shape[0])))
testX,testY = trainX[test_sample],trainY[test_sample]
trainX, trainY = trainX[train_sample],trainY[train_sample]
testX.shape, trainX.shape, testY.shape, trainY.shape
```

Out[7]:

```
((511278, 60), (1533832, 60), (511278,), (1533832,))
```

In [8]:

```
reg = LinearRegression()
reg.fit(trainX, trainY)
preds = reg.predict(testX)
```

In [9]:

```
r2_score(testY, preds)
```

Out[9]:

```
0.939110501363332
```

In [10]:

```
mean_squared_error(testY, preds)
```

Out[10]:

```
0.06781563
```

In [11]:

```
def percentage_error(actual, predicted):
    res = np.empty(actual.shape)
    for j in range(actual.shape[0]):
        if actual[j] != 0:
            res[j] = (actual[j] - predicted[j]) / actual[j]
        else:
            res[j] = predicted[j] / np.mean(actual)
    return res

def mean_absolute_percentage_error(y_true, y_pred):
    return np.mean(np.abs(percentage_error(np.asarray(y_true), np.asarray(y_pred)))) * 100
print("Percentage Error : ", percentage_error(testY, preds))
print("Mean Absolute Percentage Error : ", mean_absolute_percentage_error(testY, preds))
```

```
Percentage Error :  [-0.06380215 -0.00099456 -0.00898759 ...  0.03641921 -0.01743657
 -0.06626017]
Mean Absolute Percentage Error :  10.833005082646299
```

In [12]:

```
from tensorflow.python.keras.layers import Dense
from tensorflow.python.keras import Sequential
import matplotlib.pyplot as plt
import math

look_back = 60
model = Sequential()
model.add(Dense(200, input_dim=look_back, activation='relu'))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='SGD')
model.fit(trainX, trainY, epochs=1, batch_size=128, verbose=2)
trainScore = model.evaluate(trainX, trainY, verbose=0)
print('Train Score: %.2f MSE (%.2f RMSE)' % (trainScore, math.sqrt(trainScore)))
```

```
testScore = model.evaluate(testX, testY, verbose=0)
print('Test Score: %.2f MSE (%.2f RMSE)' % (testScore, math.sqrt(testScore)))
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
```

```
Train on 1533832 samples
1533832/1533832 - 27s - loss: 0.0877
Train Score: 0.07 MSE (0.26 RMSE)
Test Score: 0.07 MSE (0.26 RMSE)
```

In [13]:

```
testPredict = testPredict.ravel()
testPredict.shape
```

Out[13]:

```
(511278,)
```

In [14]:

```
r2_score(testY, testPredict)
```

Out[14]:

```
0.9381916676325132
```

In [15]:

```
mean_squared_error(testY, testPredict)
```

Out[15]:

```
0.068839
```

In [16]:

```
def percentage_error(actual, predicted):
    res = np.empty(actual.shape)
    for j in range(actual.shape[0]):
        if actual[j] != 0:
            res[j] = (actual[j] - predicted[j]) / actual[j]
        else:
            res[j] = predicted[j] / np.mean(actual)
    return res

def mean_absolute_percentage_error(y_true, y_pred):
    return np.mean(np.abs(percentage_error(np.asarray(y_true), np.asarray(y_pred)))) * 100
print("Percentage Error : ", percentage_error(testY, testPredict))
print("Mean Absolute Percentage Error : ", mean_absolute_percentage_error(testY, testPredict))
```

```
Percentage Error :  [-0.02525267  0.02667473  0.02929849 ...  0.05809522 -0.01401896
 -0.06440991]
Mean Absolute Percentage Error :  10.596146906299808
```

In [17]:

```
data=np.array(data)
nan_vals = list()
for i in l:
    testX = data[i-60:i]
    data[i] = model(testX.reshape(1,-1))[0][0]
    nan_vals.append(data[i])
```

```
WARNING:tensorflow:Layer dense is casting an input tensor from dtype float64 to the layer's dtype
of float32, which is new behavior in TensorFlow 2.  The layer has dtype float32 because it's dtype
defaults to floatx.

If you intended to run this layer in float32, you can safely ignore this warning. If in doubt, thi
s warning is likely only an issue if you are porting a TensorFlow 1.X model to TensorFlow 2
```

s warning is likely only an issue if you are porting a tensorflow 1.X model to tensorflow 2.

To change all layers to have dtype float64 by default, call
`tf.keras.backend.set_floatx('float64')`. To change just this layer, pass dtype='float64' to the l
ayer constructor. If you are the author of this layer, you can disable autocasting by passing auto
cast=False to the base Layer constructor.

In [18]:

```
len(nan_vals)
```

Out[18]:

```
25979
```

In [47]:

```
nan_vals
```

Out[47]:

```
[0.2727518379688263,
 0.3019287884235382,
 6.0962324142456055,
 6.135959148406982,
 3.3243801593780518,
 2.153040647506714,
 2.9623236656188965,
 2.8170433044433594,
 0.41989386081695557,
 0.48729971051216125,
 0.4854073226451874,
 0.47955751419067383,
 0.4750494062900543,
 0.460338294506073,
 0.45899996161460876,
 0.4604743719100952,
 0.5029863119125366,
 0.5319709777832031,
 0.5390459895133972,
 0.5479671359062195,
 0.5664830803871155,
 0.5839320421218872,
 0.5835678577423096,
 0.609924852848053,
 0.6522421836853027,
 0.67695152759552,
 0.6948468685150146,
 0.730141282081604,
 0.7969890236854553,
 0.8463677167892456,
 0.8902251124382019,
 0.9382801055908203,
 0.987629771232605,
 1.0183607339859009,
 1.0541071891784668,
 1.0672767162322998,
 1.0893012285232544,
 1.098063349723816,
 1.102694034576416,
 1.1235604286193848,
 1.135292649269104,
 1.1679562330245972,
 1.1942332983016968,
 1.242233157157898,
 1.2685171365737915,
 1.2765986919403076,
 1.3027678728103638,
 1.3197072744369507,
 1.3018693923950195,
 1.2899311780929565,
 1.2654837369918823,
 1.2592225074768066,
 1.2818952798843384,
```

1.2898224592208862,
1.2920030355453491,
1.281884789466858,
1.277136206626892,
1.2804445028305054,
1.2792164087295532,
1.2775593996047974,
1.3012160062789917,
1.3130955696105957,
1.3100825548171997,
1.3084372282028198,
1.312529444694519,
1.315608263015747,
1.318050742149353,
1.321891188621521,
1.3241735696792603,
1.324074625968933,
1.3214093446731567,
1.3204338550567627,
1.318140983581543,
1.3160816431045532,
1.3111201524734497,
1.3065701723098755,
1.3049532175064087,
1.307543158531189,
1.3104186058044434,
1.3159652948379517,
1.323165774345398,
1.3278441429138184,
1.330008864402771,
1.331587791442871,
1.334079384803772,
1.3370977640151978,
1.3425999879837036,
1.350983738899231,
1.3612343072891235,
1.372458815574646,
1.383742332458496,
1.3924188613891602,
1.4025384187698364,
1.412593960762024,
1.4230667352676392,
1.4346835613250732,
1.4461333751678467,
1.4566770792007446,
1.4670912027359009,
1.477601170539856,
1.4883989095687866,
1.4999068975448608,
1.5124167203903198,
1.5252543687820435,
1.5371679067611694,
1.5487374067306519,
1.5592745542526245,
1.569456696510315,
1.5795838832855225,
1.589740514755249,
1.5995776653289795,
1.6086090803146362,
1.617200493812561,
1.6261810064315796,
1.6348413228988647,
1.6435540914535522,
1.6527670621871948,
1.661863923072815,
1.6704128980636597,
1.6788045167922974,
1.6872693300247192,
1.695369839668274,
1.7032915353775024,
1.7110658884048462,
1.718775987625122,
1.7263482809066772,
1.7338112592697144,
1.7409487962722778,
1.7476083040237427,
1.7539480924606323,

1.7600374221801758,
1.7657004594802856,
1.7710429430007935,
1.7761512994766235,
1.7808398008346558,
1.7855323553085327,
1.7905043363571167,
1.7957788705825806,
1.8013641834259033,
1.8072621822357178,
1.8133103847503662,
1.8194643259048462,
1.825697898864746,
1.8321610689163208,
1.838908076286316,
1.8461220264434814,
1.8537449836730957,
1.8616966009140015,
1.870102047920227,
1.8787404298782349,
1.8874818086624146,
1.896275520324707,
1.905086636543274,
1.913896918296814,
1.9227814674377441,
1.93171226978302,
1.940679907798767,
1.9496368169784546,
1.9586029052734375,
1.967545986175537,
1.9764801263809204,
1.9854754209518433,
1.9945701360702515,
2.003750801086426,
2.0130035877227783,
2.0223381519317627,
2.0317540168762207,
2.041208267211914,
2.050693988800049,
2.0602049827575684,
2.0696792602539062,
2.079099178314209,
2.0884835720062256,
2.0978407859802246,
2.1071267127990723,
2.1163315773010254,
2.125450372695923,
2.1344611644744873,
2.143371343612671,
2.15219783782959,
2.160944938659668,
2.16964054107666,
2.178356647491455,
2.1870920658111572,
2.1958415508270264,
2.204610824584961,
2.2134006023406982,
2.2221851348876953,
2.2309694290161133,
2.2397570610046387,
2.24853253364563,
2.257290840148926,
2.266024589538574,
2.274711847305298,
2.28334641456604,
2.2919249534606934,
2.300450086593628,
2.3089261054992676,
2.317356586456299,
2.325744152069092,
2.334090232849121,
2.3424580097198486,
2.350855827331543,
2.3592910766601562,
2.3677730560302734,
2.3763060569763184,
2.3848936557769775,

2.3935494422912598,
2.402278423309326,
2.411083459854126,
2.419963836669922,
2.4289138317108154,
2.437922477722168,
2.4469802379608154,
2.456083297729492,
2.4652230739593506,
2.4743893146514893,
2.483576536178589,
2.4927802085876465,
2.501997470855713,
2.511225700378418,
2.5204644203186035,
2.529715061187744,
2.5389785766601562,
2.5482587814331055,
2.5575613975524902,
2.5668904781341553,
2.5762486457824707,
2.585638999938965,
2.5950636863708496,
2.604520797729492,
2.6140100955963135,
2.6235291957855225,
2.6330740451812744,
2.642639636993408,
2.652221918106079,
2.661813735961914,
2.6714091300964355,
2.681006669998169,
2.6906023025512695,
2.700191020965576,
2.7097702026367188,
2.7193398475646973,
2.728900671005249,
2.7384562492370605,
2.748008966445923,
2.7575600147247314,
2.7671122550964355,
2.7766690254211426,
2.7862329483032227,
2.7958078384399414,
2.8053958415985107,
2.814997673034668,
2.8246138095855713,
2.8342456817626953,
2.8438925743103027,
2.853553295135498,
2.863227605819702,
2.872913360595703,
2.8826088905334473,
2.8923137187957764,
2.902027130126953,
2.911748170852661,
2.921477794647217,
2.93121600151062,
2.9409637451171875,
2.9507217407226562,
2.9604921340942383,
2.9702768325805664,
2.9800777435302734,
2.9898970127105713,
2.9997220039367676,
3.0095505714416504,
3.0193850994110107,
3.029228687286377,
3.03908371925354,
3.0489511489868164,
3.0588297843933105,
3.068718194961548,
3.078615665435791,
3.088521957397461,
3.0984363555908203,
3.108358383178711,
3.1182878017425537,

3.1282238960266113,
3.138167142868042,
3.148118495941162,
3.1580777168273926,
3.1680455207824707,
3.1780219078063965,
3.18800687789917,
3.1980013847351074,
3.208005905151367,
3.2180216312408447,
3.228048801422119,
3.2380876541137695,
3.248138189315796,
3.258200168609619,
3.26827335357666,
3.27835750579834,
3.2884521484375,
3.2985565662384033,
3.30867338180542,
3.3188211917877197,
3.328998565673828,
3.339203357696533,
3.3494322299957275,
3.3596837520599365,
3.3699562549591064,
3.3802502155303955,
3.3905670642852783,
3.4009077548980713,
3.4112720489501953,
3.4216599464416504,
3.43207049369812,
3.442502498626709,
3.452956199645996,
3.463430881500244,
3.473926067352295,
3.4844419956207275,
3.4949779510498047,
3.5055346488952637,
3.5161123275756836,
3.5267105102539062,
3.5373287200927734,
3.5479660034179688,
3.558622360229492,
3.5692970752716064,
3.5799901485443115,
3.5907015800476074,
3.6014304161071777,
3.6121764183044434,
3.622938632965088,
3.633716583251953,
3.644509792327881,
3.6553189754486084,
3.6661431789398193,
3.676980972290039,
3.68782901763916,
3.6986875534057617,
3.7095577716827393,
3.720439910888672,
3.7313337326049805,
3.7422378063201904,
3.753152370452881,
3.764077663421631,
3.7750144004821777,
3.7859625816345215,
3.796922206878662,
3.8078927993774414,
3.818875551223755,
3.8298702239990234,
3.8408772945404053,
3.8518965244293213,
3.8629281520843506,
3.8739712238311768,
3.8850269317626953,
3.896094799041748,
3.907174587249756,
3.918266534805298,
3.929370880126953,

3.940488338470459,
3.9516196250915527,
3.9627645015716553,
3.973923921585083,
3.9850986003875732,
3.996288776397705,
4.0074944496154785,
4.018716335296631,
4.02995491027832,
4.041210174560547,
4.052482604980469,
4.063772678375244,
4.075079917907715,
4.086405277252197,
4.097748756408691,
4.109109878540039,
4.120489120483398,
4.1318864822387695,
4.143302917480469,
4.15473747253418,
4.1661906242370605,
4.177661895751953,
4.189152717590332,
4.200662612915039,
4.212191104888916,
4.223738193511963,
4.23530387878418,
4.246888160705566,
4.2584919929504395,
4.270114898681641,
4.2817559242248535,
4.2934160232543945,
4.3050947189331055,
4.316791534423828,
4.3285064697265625,
4.340240001678467,
4.351991653442383,
4.3637614250183105,
4.375548362731934,
4.38735294342041,
4.399174213409424,
4.411013126373291,
4.422868728637695,
4.434741020202637,
4.446630477905273,
4.458536148071289,
4.470458030700684,
4.482396125793457,
4.494351387023926,
4.506321907043457,
4.518309116363525,
4.5303120613098145,
4.542331695556641,
4.5543670654296875,
4.566418647766113,
4.578485488891602,
4.590568542480469,
4.602667331695557,
4.614782333374023,
4.626913070678711,
4.639059543609619,
4.65122127532959,
4.663399696350098,
4.675593852996826,
4.687804222106934,
4.70003080368042,
4.712273597717285,
4.7245330810546875,
4.736808776855469,
4.749101161956787,
4.761409759521484,
4.773736000061035,
4.786078453063965,
4.798439025878906,
4.810816287994385,
4.823211669921875,
4.835624217987061,

4.848053455352783,
4.860500335693359,
4.8729658126831055,
4.885448932647705,
4.897950172424316,
4.9104695320129395,
4.923007488250732,
4.935563564300537,
4.948138236999512,
4.960731029510498,
4.973341941833496,
4.9859724044799805,
4.998621463775635,
5.011288642883301,
5.023975372314453,
5.036680698394775,
5.049405097961426,
5.062148571014404,
5.074910640716553,
5.087691307067871,
5.100491523742676,
5.113310813903809,
5.1261491775512695,
5.139006614685059,
5.151883125305176,
5.164777755737305,
5.17769193649292,
5.190625190734863,
5.203577041625977,
5.216547966003418,
5.229537487030029,
5.242546081542969,
5.255573272705078,
5.268619537353516,
5.281684398651123,
5.294767379760742,
5.3078694343566895,
5.320990562438965,
5.33413028717041,
5.347289085388184,
5.360466480255127,
5.373661994934082,
5.386876583099365,
5.40010929107666,
5.413360595703125,
5.42663049697876,
5.439919471740723,
5.4532270431518555,
5.466552734375,
5.479897499084473,
5.493260383605957,
5.506641864776611,
5.520042419433594,
5.533461570739746,
5.546899795532227,
5.560356140136719,
5.573831558227539,
5.587325096130371,
5.600837707519531,
5.6143693923950195,
5.627920150756836,
5.6414899826049805,
5.655078887939453,
5.668686866760254,
5.682314395904541,
5.695960998535156,
5.709626197814941,
5.723311424255371,
5.737015724182129,
5.750739574432373,
5.764482498168945,
5.778245449066162,
5.792027950286865,
5.805830955505371,
5.819652557373047,
5.833494186401367,
5.84735488891601

5.861236572265625,
5.875138282775879,
5.889059543609619,
5.903001308441162,
5.916962623596191,
5.930944442749023,
5.9449462890625,
5.9589691162109375,
5.973011493682861,
5.98707389831543,
6.001156806945801,
6.015259742736816,
6.029382705688477,
6.043526649475098,
6.057690620422363,
6.07187557220459,
6.086080551147461,
6.100306510925293,
6.1145524978637695,
6.128818511962891,
6.143105506896973,
6.157413005828857,
6.171741008758545,
6.186089515686035,
6.200459003448486,
6.21484899520874,
6.229259490966797,
6.2436909675598145,
6.258143424987793,
6.272616386413574,
6.287109375,
6.301623344421387,
6.316158294677734,
6.330713272094727,
6.34528923034668,
6.359886646270752,
6.374504566192627,
6.389142990112305,
6.403801918029785,
6.418481826782227,
6.433182716369629,
6.447904586791992,
6.462647438049316,
6.477411270141602,
6.492196083068848,
6.507001876831055,
6.521827697753906,
6.536675930023193,
6.551544666290283,
6.566433906555176,
6.581343650817871,
6.596275806427002,
6.611228942871094,
6.6262030601501465,
6.641198635101318,
6.656215667724609,
6.6712541580200195,
6.686313629150391,
6.70139217376709,
6.71648645401001,
6.731595993041992,
6.7467217445373535,
6.76186466217041,
6.777024269104004,
6.792201042175293,
6.807394981384277,
6.822606086730957,
6.837834358215332,
6.8530802726745605,
6.868343830108643,
6.883625030517578,
6.898925304412842,
6.914243221282959,
6.929579734802246,
6.944934844970703,
6.960308074951172,
6.975699901580810

6.9736999015808105,
6.991110801696777,
7.006540298461914,
7.0219879150390625,
7.037454605102539,
7.052939414978027,
7.068443775177002,
7.083967208862305,
7.0995097160339355,
7.115071773529053,
7.130653381347656,
7.146254539489746,
7.161874771118164,
7.177514553070068,
7.193174362182617,
7.208853721618652,
7.224553108215332,
7.240273475646973,
7.256013870239258,
7.271773815155029,
7.287553787231445,
7.303353786468506,
7.319174766540527,
7.33501672744751,
7.350878715515137,
7.366761207580566,
7.382663249969482,
7.398586750030518,
7.4145307540893555,
7.430495738983154,
7.446479797363281,
7.462483882904053,
7.478509426116943,
7.4945549964904785,
7.510622024536133,
7.52670955657959,
7.542818546295166,
7.558948516845703,
7.575098991394043,
7.591270446777344,
7.607462406158447,
7.62367582321167,
7.6399102210998535,
7.656166076660156,
7.672442436218262,
7.688739776611328,
7.705057144165039,
7.721395492553711,
7.7377543449401855,
7.754134178161621,
7.770534038543701,
7.7869553565979,
7.8033976554870605,
7.819860935211182,
7.836344242095947,
7.852848529815674,
7.869373798370361,
7.885919570922852,
7.9024858474731445,
7.919073581695557,
7.93568229675293,
7.9523115158081055,
7.968961715698242,
7.985633373260498,
8.002326965332031,
8.019041061401367,
8.035775184631348,
8.052530288696289,
8.069306373596191,
8.086104393005371,
8.102924346923828,
8.11976432800293,
8.136625289916992,
8.153507232666016,
8.170411109924316,
8.187336921691895,
8.204282760620117,
8.221249580383

8.22124958033,
8.238239288330078,
8.255249977111816,
8.272281646728516,
8.289335250854492,
8.30640983581543,
8.323506355285645,
8.340624809265137,
8.357766151428223,
8.37492847442627,
8.392111778259277,
8.409317016601562,
8.426543235778809,
8.443791389465332,
8.461061477661133,
8.478353500366211,
8.495667457580566,
8.513004302978516,
8.530363082885742,
8.54774284362793,
8.565145492553711,
8.582569122314453,
8.600014686584473,
8.617483139038086,
8.634974479675293,
8.652487754821777,
8.670022964477539,
8.687580108642578,
8.705160140991211,
8.722763061523438,
8.740387916564941,
8.758034706115723,
8.775703430175781,
8.793395042419434,
8.81110954284668,
8.828845977783203,
8.84660530090332,
8.864387512207031,
8.88219165802002,
8.900018692016602,
8.917867660522461,
8.935739517211914,
8.953634262084961,
8.971550941467285,
8.989490509033203,
9.007452964782715,
9.02543830871582,
9.04344654083252,
9.061477661132812,
9.079530715942383,
9.097606658935547,
9.115706443786621,
9.133829116821289,
9.151975631713867,
9.170145034790039,
9.188336372375488,
9.206550598144531,
9.224788665771484,
9.243049621582031,
9.261333465576172,
9.279641151428223,
9.297971725463867,
9.316326141357422,
9.334701538085938,
9.35310173034668,
9.371524810791016,
9.389970779418945,
9.408441543579102,
9.426935195922852,
9.445451736450195,
9.463991165161133,
9.48255443572998,
9.501141548156738,
9.519752502441406,
9.538385391235352,
9.557043075561523,
9.575723648071289,
9.594428062438965,

9.594428062438965,
9.613157272338867,
9.63191032409668,
9.650686264038086,
9.669485092163086,
9.688307762145996,
9.707155227661133,
9.72602653503418,
9.74492073059082,
9.763839721679688,
9.782783508300781,
9.801750183105469,
9.820741653442383,
9.839756965637207,
9.858796119689941,
9.877859115600586,
9.89694595336914,
9.916057586669922,
9.93519401550293,
9.954354286193848,
9.973539352416992,
9.992748260498047,
10.011981964111328,
10.031240463256836,
10.050521850585938,
10.069828033447266,
10.089158058166504,
10.108513832092285,
10.127893447875977,
10.147296905517578,
10.166725158691406,
10.186178207397461,
10.205656051635742,
10.22515869140625,
10.244686126708984,
10.264238357543945,
10.283815383911133,
10.30341625213623,
10.323043823242188,
10.342695236206055,
10.362371444702148,
10.382072448730469,
10.401799201965332,
10.421550750732422,
10.441327095031738,
10.461128234863281,
10.480955123901367,
10.500805854797363,
10.520682334899902,
10.540584564208984,
10.560511589050293,
10.580463409423828,
10.60044002532959,
10.620443344116211,
10.640471458435059,
10.660524368286133,
10.68060302734375,
10.700706481933594,
10.720836639404297,
10.740991592407227,
10.7611722946167,
10.781378746032715,
10.801610946655273,
10.821868896484375,
10.842151641845703,
10.862460136413574,
10.882793426513672,
10.903154373168945,
10.923540115356445,
10.943952560424805,
10.964390754699707,
10.984854698181152,
11.005343437194824,
11.025858879089355,
11.04640007019043,
11.066967010498047,
11.087562561035156,

11.108182907104492,
11.128829002380371,
11.14950180053711,
11.17020034790039,
11.190925598144531,
11.211675643920898,
11.232452392578125,
11.253255844116211,
11.274085998535156,
11.294940948486328,
11.31582260131836,
11.33673095703125,
11.357666015625,
11.37862777709961,
11.399616241455078,
11.42063045501709,
11.441671371459961,
11.462738990783691,
11.483835220336914,
11.50495719909668,
11.526105880737305,
11.547280311584473,
11.568482398986816,
11.589712142944336,
11.610968589782715,
11.632251739501953,
11.653560638427734,
11.674896240234375,
11.696260452270508,
11.717650413513184,
11.739068031311035,
11.760513305664062,
11.78198528289795,
11.803483963012695,
11.825010299682617,
11.846564292907715,
11.868144989013672,
11.889753341674805,
11.91139030456543,
11.933052062988281,
11.954741477966309,
11.976459503173828,
11.998205184936523,
12.019978523254395,
12.041778564453125,
12.063606262207031,
12.08546257019043,
12.107345581054688,
12.129257202148438,
12.151195526123047,
12.173162460327148,
12.195157051086426,
12.217179298400879,
12.239230155944824,
12.261308670043945,
12.283414840698242,
12.305548667907715,
12.32771110534668,
12.34990119934082,
12.372119903564453,
12.394367218017578,
12.416641235351562,
12.438943862915039,
12.461275100708008,
12.483633995056152,
12.506021499633789,
12.528437614440918,
12.550882339477539,
12.573355674743652,
12.595857620239258,
12.618387222290039,
12.640946388244629,
12.663534164428711,
12.686149597167969,
12.708794593811035,
12.731468200683594,
12.754169464111328,

12.776899337768555,
12.799659729003906,
12.822447776794434,
12.845264434814453,
12.868110656738281,
12.890985488891602,
12.913888931274414,
12.936821937561035,
12.959783554077148,
12.98277473449707,
13.005794525146484,
13.028843879699707,
13.051921844482422,
13.075028419494629,
13.098165512084961,
13.121332168579102,
13.14452838897705,
13.167754173278809,
13.191007614135742,
13.214292526245117,
13.237605094909668,
13.260948181152344,
13.284320831298828,
13.307722091674805,
13.331153869628906,
13.3546142578125,
13.378106117248535,
13.401626586914062,
13.425176620483398,
13.44875717163086,
13.472367286682129,
13.496006965637207,
13.519678115844727,
13.543376922607422,
13.567107200622559,
13.590867042541504,
13.614657402038574,
13.638477325439453,
13.662327766418457,
13.686208724975586,
13.710119247436523,
13.734061241149902,
13.75803279876709,
13.782033920288086,
13.806065559387207,
13.830129623413086,
13.854223251342773,
13.878347396850586,
13.902502059936523,
13.926685333251953,
13.95090103149414,
13.975147247314453,
13.99942398071289,
14.023731231689453,
14.04806900024414,
14.072439193725586,
14.096839904785156,
14.121271133422852,
14.145732879638672,
14.170225143432617,
14.194747924804688,
14.2193021774292,
14.243888854980469,
14.268506050109863,
14.293155670166016,
14.317835807800293,
14.342546463012695,
14.367290496826172,
14.39206314086914,
14.416868209838867,
14.441703796386719,
14.466571807861328,
14.491472244262695,
14.516402244567871,
14.541364669799805,
14.566359519958496,
14.591384887695312,

```
 14.61644172668457,
 14.641530990600586,
 14.666650772094727,
 14.691802978515625,
 14.716986656188965,
 14.742202758789062,
 14.767452239990234,
 14.792732238769531,
 14.818044662475586,
 14.843390464782715,
 14.868767738342285,
 14.894177436828613,
 14.919618606567383,
 14.945093154907227,
 14.970598220825195,
 14.996135711669922,
 15.021707534790039,
 15.047308921813965,
 15.072943687438965,
 15.098609924316406,
 15.124309539794922,
 15.150041580200195,
 15.175806999206543,
 15.201604843139648,
 ...]
```

In [17]:

```
!jupyter nbconvert --to html q4.ipynb
```

```
[NbConvertApp] Converting notebook q4.ipynb to html
[NbConvertApp] Writing 677949 bytes to q4.html
```

In [ ]:

```

```

In [ ]:

```

```