Homework #3A

1 Conceptual Questions

A1

a. True. k eigenvalues of covariance matrix are nonzero.

b. FALSE. It's not necessary that SVM has the lowest generalisation error.

c. TRUE. Because sampling is with replacement.

d. FALSE. It's the columns instead of rows.

e. FALSE. New PCA coordinate can be uninterpretable.

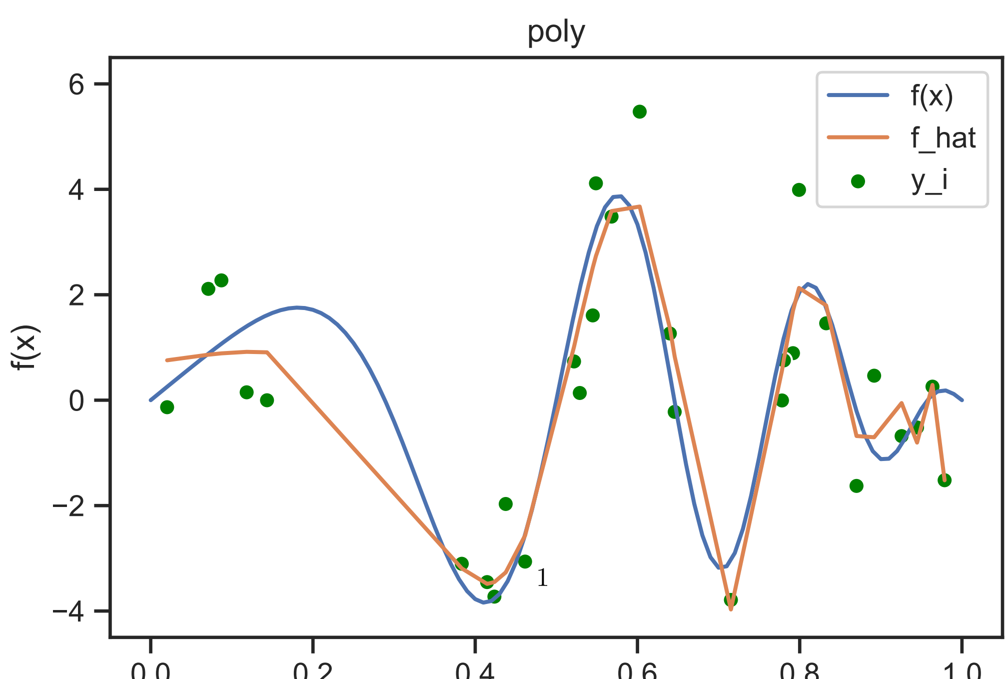
f. FALSE. In order to minimize objective we need to increase k and when each cluster has very few points, they can't show much similarities.

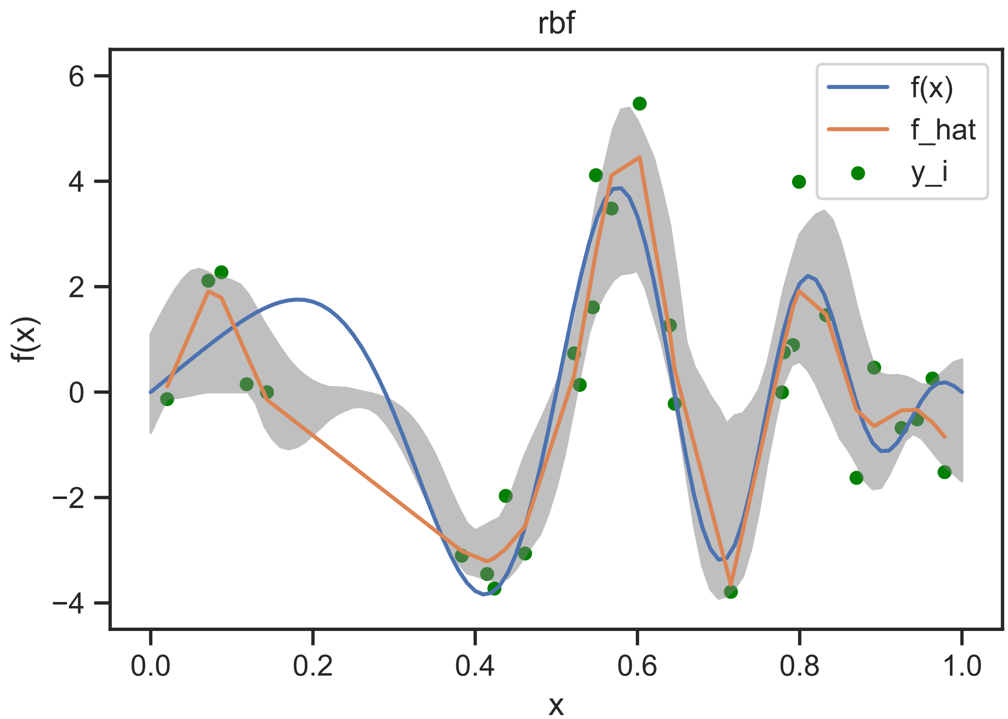
g. We need to decrease σ as shown in lecture notes.

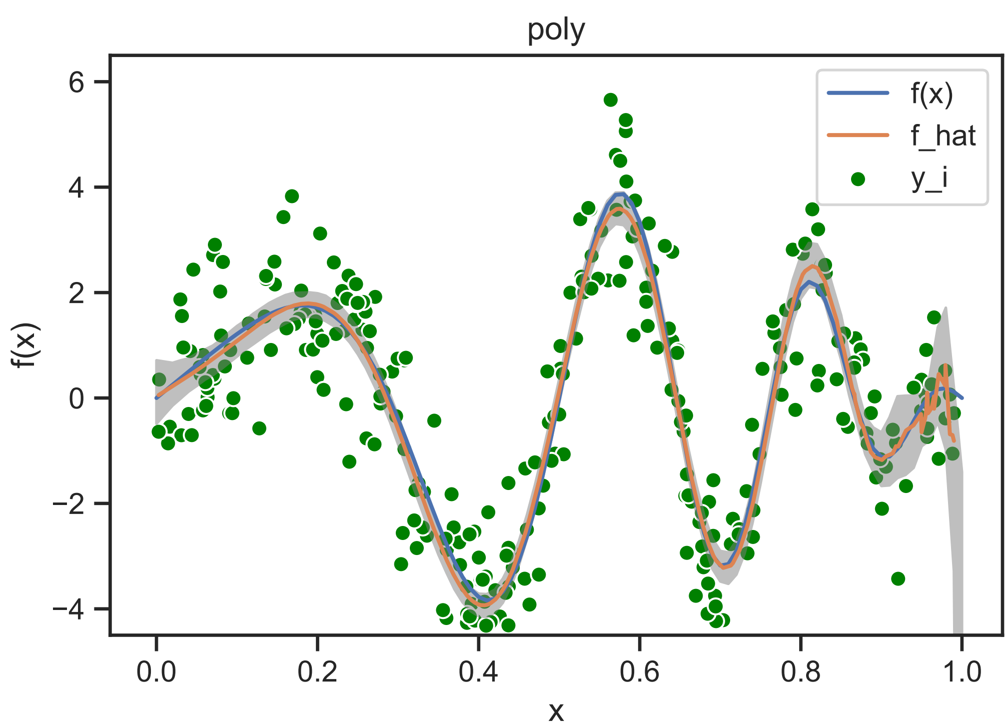
2 Kernels and the Bootstrap

A2
By definition of $\phi(x)$, we have:
$\phi(x)\phi(x') = \sum_{i=0}^{\infty} \frac{1}{\sqrt{i!}} e^{-\frac{x^2}{2}} x^i \frac{1}{\sqrt{i!}} e^{-\frac{x'^2}{2}} x'^i = e^{-\frac{x^2+x'^2}{2}} \sum_{i=0}^{\infty} \frac{1}{i!} (xx')^i = e^{-\frac{x^2+x'^2}{2}} e^{xx'} = e^{-\frac{(x-x')^2}{2}}$
A3

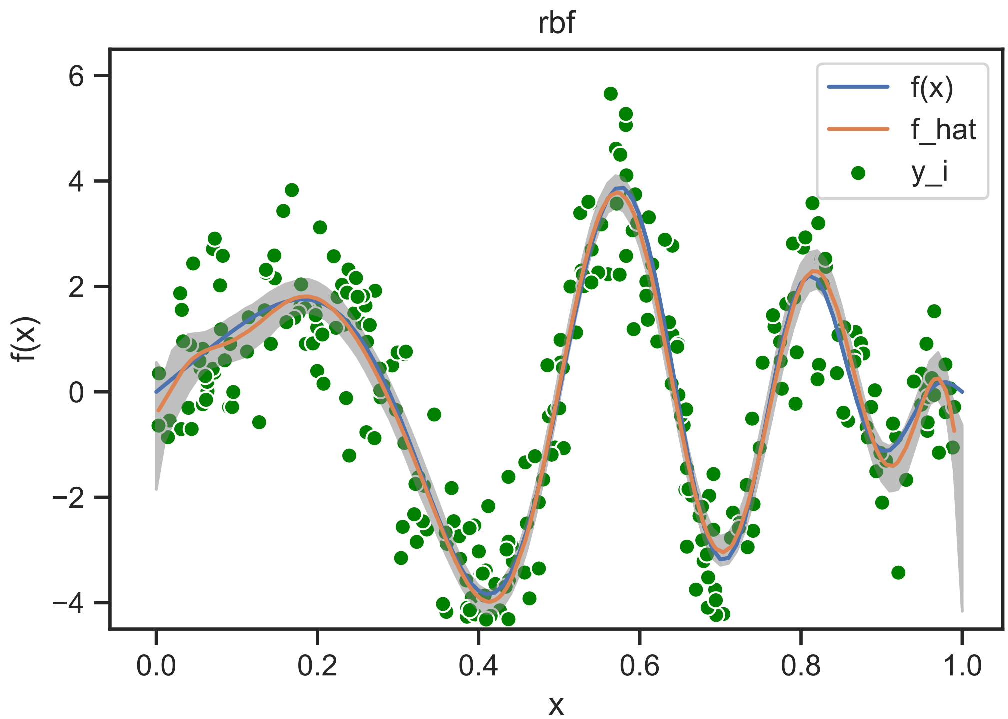
a. For rbf and $n = 30$ I used $\gamma = 177.8$ and $\lambda = 0.1$
For poly and $n = 30$ I used $d = 47$ and $\lambda = 0.316277$




rbf
f(x)
f_hat
y_i
6
4
2
0
−2
−4
0.0
0.2
0.4
0.6
0.8
1.0
x
f(x)


poly
f(x)
f_hat
y_i
6
4
2
0
−2
−4
0.0
0.2
0.4
0.6
0.8
1.0
x
f(x)

c.



d. For rbf and $n = 300$ I used $\gamma = 5.6234$ and $\lambda = 1.778 * 10^{-12}$
For poly and $n = 300$ I used $d = 41$ and $\lambda = 0.017782$
graph results are included in b and c.

e. The confidence interval is [-3.508038449929596, -2.731661579893631] and 0 is not included.

Listing 1: A3code

```
import numpy as np
import scipy.linalg as la
import matplotlib
matplotlib.use('agg')
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
sns.set_style("ticks")
np.random.seed(0)


def gen_data(n=30):
    x = np.random.uniform(0, 1, n).reshape(n, 1)
    fx = 4* np.sin(np.pi * x) * np.cos(6 * np.pi * x**2)
    eps = np.random.randn(n).reshape(n, 1)
    y = fx + eps
    return(x, fx, y)


def train(K, y, L=10**-6):
        a = la.solve( K + L * np.identity(K.shape[0]), y )
        return(a)
```

```python
def predict(x, y):
        f = x.dot(y)
        return(f)

def mse(fx, y):
        m = ((fx-y)**2).mean()
        return(m)

def poly(x, z, d):
        k = (1 + x.T.dot(z) )**d
        return(k)

def rbf(x,z,gamma):
        norm = np.linalg.norm(x-z, ord=2)**2
        k = np.exp( -gamma * norm)
        return(k)

def makeKernel(X, kernel, X2=None, hyper=100):
        if(X2 is None):
                X2 = X
        n = X.shape[0]
        m = X2.shape[0]
        K = np.zeros((m,n))
        for i in range(m):
                x = X2[i,:]
                for j in range(n):
                        z = X[j,:]
                        K[i, j] = kernel(x,z,hyper)
        return(K)

def kfold(Kvalue, y, k = 5):
        idx = np.random.permutation(Kvalue.shape[0])
        ktrainlist = []
        kvallist = []
        ytrainlist = []
        yvallist = []
        for i in range(k):
                start = int(i*X.shape[0]/k)
                end = int((i+1)*X.shape[0]/k)
                idx_val = idx[start:end]
                idx_train = np.concatenate( (idx[0:start], idx[end:]) )
                ktrainlist.append(Kvalue[idx_train, :][:, idx_train]  )
                kvallist.append(Kvalue[idx_val, :][:, idx_train])

                ytrainlist.append(y[idx_train, :])
                yvallist.append(y[idx_val, :])

        return(ktrainlist, kvallist, ytrainlist, yvallist)


def plotsinthisp(X, fx, y, kernel, hypa, L, x, p5, p95):
        K = makeKernel(X, kernel, hyper = hypa)
        alpha = train(K, y, L=L)
        f = predict(K, alpha)
```

```
        n = X.shape[0]

        name = "{}_{}.pdf".format(n, kernel.__name__)

        fx = 4* np.sin(np.pi * x) * np.cos(6 * np.pi * x**2)

        sns.scatterplot(X[:,0], y[:,0], color="green", label="y_i")
        sns.lineplot(x[:,0], fx[:,0], label="f(x)")
        sns.lineplot(X[:,0], f[:,0], label="f_hat")
        plt.title(kernel.__name__)
        plt.legend()
        plt.xlabel("x")
        plt.ylabel("f(x)")
        plt.ylim(-4.5, 6.5)
        sns.lineplot(x[:,0], p5, label="5_conf")
        sns.lineplot(x[:,0], p95, label="95_conf")
        plt.fill_between(x[:,0], p5, p95, color='grey', alpha=0.5)

        plt.clf()
        print(name, mse)

def DoCV(X, fx, y, kernel, k=10):
    n = X.shape[0]
    if(kernel == rbf):
        hypa = np.float_power( 10, np.arange(-3, 4, .25))
    elif(kernel == poly):
            hypa = np.arange(1, 100, 2)

    Ls = np.float_power( 10, np.arange(-10, 5, .25) )
    result_list = []
    for hyp in hypa:
        K = makeKernel(X, kernel, hyper = hyp)
        K_trains, y_trains, K_vals, y_vals =  kfold(K, y, k = k)
        for L in Ls:
            mse_list = []
            for i in range(k):
                K_train = K_trains[i]; y_train = y_trains[i]; K_val = K_vals[i]; y_val = y_v
                alpha = train(K_train, y_train, L=L)
                f = predict(K_val, alpha)
                m_s_e = mse(f, y_val)
                mse_list.append(m_s_e)
                result_list.append( ( np.mean(mse_list), hyp, L ) )

    best = np.inf
    bestidx = 0
    for idx, line in enumerate(result_list):
        if(line[0] < best):
            best = line[0]
            bestidx = idx
    mse, hyper, L = result_list[bestidx]
    print(mse, hyper, L)
    return(hyper, L)

def bootstrap(X, y, kernel, hyper, L, B=300):
        n = X.shape[0]
```

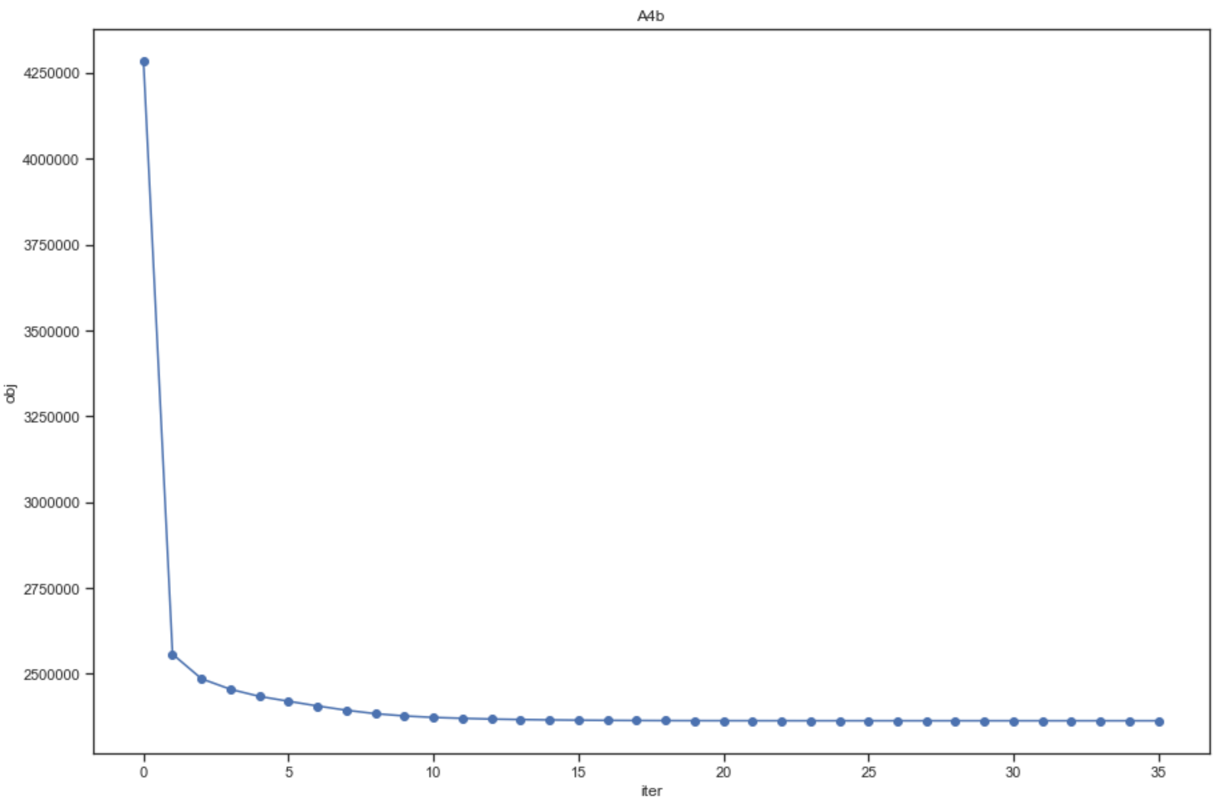
```
        step = .01
        x = np.arange(0, 1 + step, step)
        x = x.reshape(x.shape[0], 1)

        testn = 40

        fs = np.zeros((B, x.shape[0]))
        for i in range(B):
                if(i % 100 == 0 ):
                        print("bootstrap", i)
                idxs = np.random.choice(n, n)
                Xb = X[idxs,:]
                yb = y[idxs,:]
                K = makeKernel(Xb, kernel, hyper = hyper)
                alpha = train(K, yb, L=L)

                kx = makeKernel(Xb, kernel, X2=x, hyper = hyper)
                f = predict(kx, alpha)
                fs[i, :] = f[:,0]

        p5 = np.percentile(fs, 5, axis=0)
        p95 = np.percentile(fs, 95, axis=0)
        return(x, p5, p95)


hypers = [177.82794100389228, 47, 5.6234, 41]
Ls = [0.1, 0.316277, 1.778*(10**-12), 0.017782]
kernels = [rbf, poly, rbf, poly]



i=0
for n in [30, 300]:
        X, fx, y = gen_data(n);
        k = X.shape[0]
        if(k > 30):
                k = 10
        for kernel in [rbf, poly]:
                if(hypers is None):
                        hyper, L = DoCV(X, fx, y, kernel, k=k)
                else:
                        hyper = hypers[i]; L = Ls[i]
                print(hyper, L)
                x, p5, p95 = bootstrap(X,y, kernel, hyper, L)
                plotsinthisp(X, fx, y, kernel, hyper, L, x, p5, p95)
                i += 1

#part e
m=1000
X_e, fx_e, Y_e= gen_data(m);
K_poly, K_rbf=makeKernel(X_e, poly, X2=None, hyper=5.6234), makeKernel(X_e, rbf, X2=None, hy
a_poly, a_rbf=train(K_poly, Y_e, L=10**-6), train(K_rbf, Y_e, L=10**-6)
outputlist_e=[];
for i in range(300):
    a= np.random.choice(len(X_e), size=len(X_e), replace = True)
    X_a, y_a = X[a], y[a]
```

```
    poly_pred=predict(K_poly, a_poly)
    rbf_pred=predict(K_rbf, a_rbf)
    outputlist_e.append(np.mean((y_a - poly_pred)**2 - (y_a - rbf_pred)**2))

CI_lower = np.percentile(outputlist_e, 5)
CI_upper = np.percentile(outputlist_e, 95)
print(CI_lower, CI_upper)
```

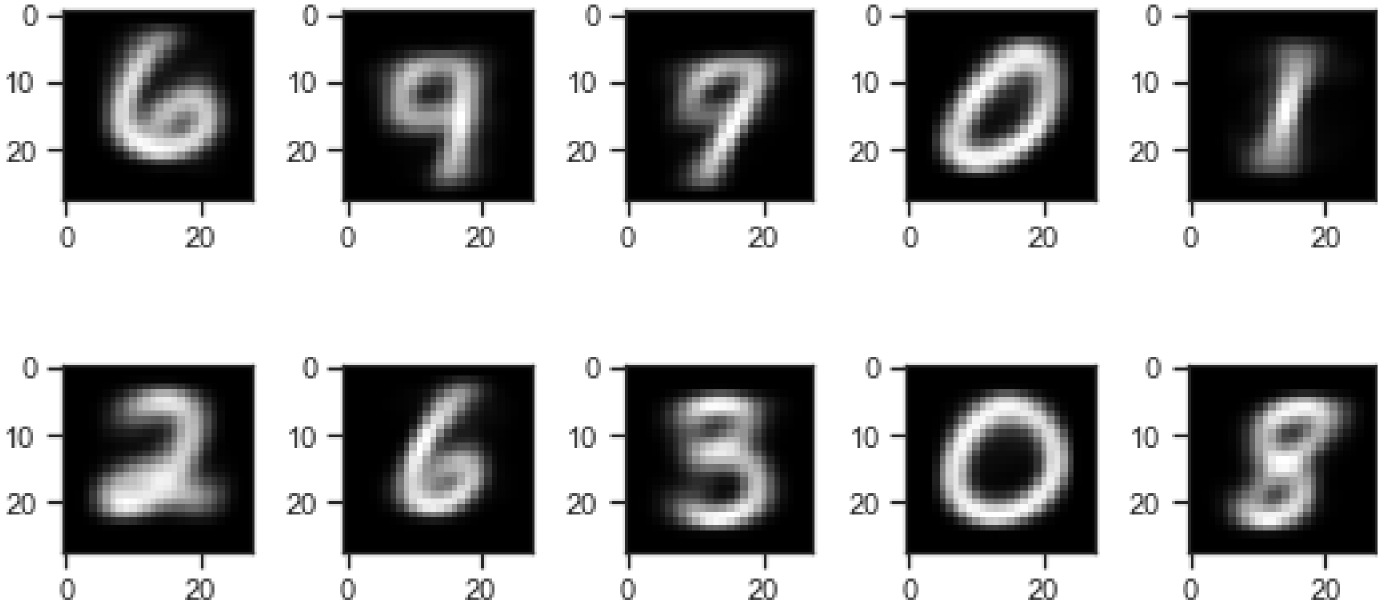
3 k-means clustering

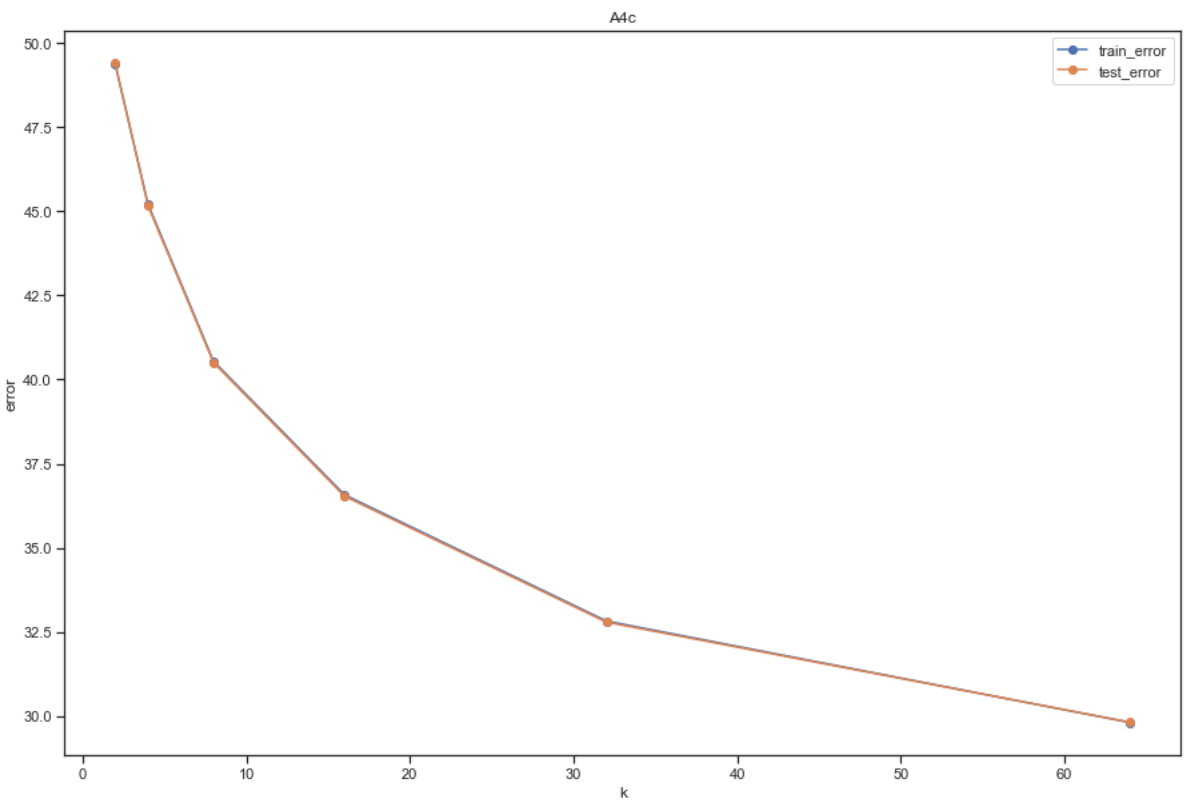
A4

a. see the code



b.





c.

Listing 2: A4code

```
import numpy as np
import matplotlib.pyplot as plt
from mnist import MNIST

def load_dataset():
    mndata = MNIST('./data/')
    mndata.gz=True
```

```python
    X_train, labels_train = map(np.array, mndata.load_training())
    X_test, labels_test = map(np.array, mndata.load_testing())
    X_train = X_train/255.0
    X_test = X_test/255.0
    return X_train,labels_train, X_test, labels_test


class KMeans:

    def __init__(self, k):
        self.k = k
        self.centroids = None
        self.clusters_train = None
        self.obj_List = []

    def Lloyd(self, X, itera, eps):
        initial_centroids = X[np.random.choice(len(X), size=self.k, replace = False)]
        # init_centroids = np.random.normal(0.5, 0.5,init_centroids.shape).astype('float32',
        # init_centroids = 10+np.random.randn(self.k, X.shape[1]).astype('float32')

        centroids = np.copy(initial_centroids)
        centroidspre = initial_centroids + np.inf

        distance = np.zeros((len(X),self.k))

        i = 0

        while np.linalg.norm(centroids - centroidspre) > eps and i < itera:
            i += 1
            centroidspre = np.copy(centroids)

            #compute the distance
            for j in range(self.k):
                distance[:,j] = np.linalg.norm(X - centroids[j], axis=1)**2

            partition = np.argmin(distance, axis = 1)
            assert len(partition) == len(X)

            newlist = []
            obj = 0
            for j in range(self.k):
                cluster = X[partition == j]
                obj += np.sum(np.linalg.norm(cluster - centroids[j], axis = 1)**2)
                centroid = np.mean(cluster, axis = 0)
                newlist.append(centroid)
            centroids = np.copy(np.array(newlist))
            self.obj_List.append(obj)
        self.centroids = centroids
        self.clusters_train = partition
        return


    def predict(self, X):
        distance = np.zeros((len(X),self.k))
        for j in range(self.k):
```

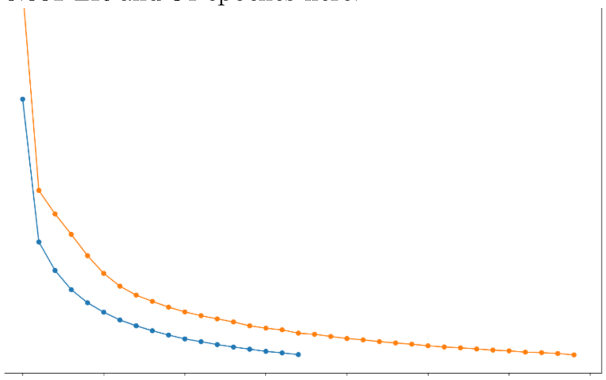
```
                distance[:,j] = np.linalg.norm(X - self.centroids[j], axis=1)**2
        partition = np.argmin(distance, axis = 1)
        pred = self.centroids[partition]
        return pred



if __name__ == "__main__":
    X_train, y_train, X_test, y_test = load_dataset()

    kmeans = KMeans(k = 10)

    kmeans.Lloyd(X_train, itera = 100, eps=0.01)

    plt.figure(figsize = (15,10))
    plt.plot(kmeans.obj_List, '-o')
    plt.title('A4b')
    plt.xlabel('iter')
    plt.ylabel('obj')
    plt.show()

    fig, axes = plt.subplots(2, 5, figsize=(1.5*5,2*2))
    for i, axe in enumerate(axes.flatten()):
        axe.imshow(kmeans.centroids[i].reshape(28,28), cmap='blue')
    plt.tight_layout()
    plt.show()

    k_range = 2**np.arange(1,7)

    trainerrlist = []
    testerrlist = []
    for k in k_range:
        kmeanss = KMeans(k = k)
        kmeanss.Lloyd(X_train, itera = 40, eps = 1e-1)
        train_pred = kmeanss.predict(X_train)
        trainerrlist.append(np.mean(np.linalg.norm(X_train - train_pred, axis = 1)**2))
        test_pred = kmeanss.predict(X_test)
        testerrlist.append(np.mean(np.linalg.norm(X_test - test_pred, axis = 1)**2))

    plt.figure(figsize = (15,10))
    plt.plot(k_range, trainerrlist, '-o', label = 'train_error')
    plt.plot(k_range, testerrlist, '-o', label = 'test_error')
    plt.title('A4c')
    plt.legend()
    plt.xlabel('k')
    plt.ylabel('error')
    plt.show()
```

4 Neural Networks for MNIST

A5

a. I take the learning rate as 0.001 and when epoch =18 it reaches the accuracy¿0.99.

b. 0.001 LR and 34 epoches here.



c. 50890 for the wide one and 26506 for the first one

Listing 3: A5code

```
import numpy as np
import matplotlib.pyplot as plt
import torch
import torchvision.datasets as datasets
import torchvision.transforms as transforms
from tqdm import tqdm
import torch.nn as nn
To_Tensor = transforms.ToTensor()

mnist_trainset = datasets.MNIST(
        root='./data', train=True, download=True, transform=To_Tensor)
mnist_testset = datasets.MNIST(
        root='./data', train=False, download=True, transform=To_Tensor)
train_loader = torch.utils.data.DataLoader(mnist_trainset,batch_size=128,shuffle=True)
test_loader = torch.utils.data.DataLoader(mnist_testset,batch_size=128, shuffle=True)

def  ws(input, W0, W1, b0, b1):
        return torch.matmul(nn.functional.relu(torch.matmul(input, W0.T)+b0), W1.T) +b1

def dn(input, W0, W1, W2, b0, b1, b2):
        temp = torch.matmul(input, W0.T)+b0
        return torch.matmul(nn.functional.relu(torch.matmul(nn.functional.relu(temp), W1.T)

def accuracyandloss(test_loader, net_type, W0=None, W1=None, W2=None, b0=None, b1=None, b2=N
        a = 0
        loss = 0
        for i, j in tqdm(iter(test_loader)):
                i = torch.flatten(i, 1, 3)
                if net_type == 'wide':
                        result = ws(i, W0, W1, b0, b1)
                        pred = torch.argmax(result,1)
                elif net_type == 'deep':
                        result = dn(i, W0, W1, W2, b0, b1, b2)
                        pred = torch.argmax(result,1)

                a += torch.sum(pred == j)
                loss += torch.nn.functional.cross_entropy(result, j, size_average = False)
```

```
        return a.to(dtype=torch.float)/len(test_loader.dataset), loss/len(test_loader.datase


def trainws(dataload, LR, epochs, h , m):
        alpha = 1/np.sqrt(m)
        W0 = -2*alpha* torch.rand(h, m) + alpha
        W1 = -2*alpha* torch.rand(10, h) + alpha
        b0 = -2*alpha* torch.rand(h) + alpha
        b1 = -2*alpha* torch.rand(10) + alpha
        parameters = [W0, W1, b0, b1]
        opt = torch.optim.Adam(parameters, LR)
        loss_list = []
        for epoch in range(epochs):
                a = 0
                loss_list.append(0)
                for i, j in dataload:
                        result = ws(i, W0, W1, b0, b1)
                        pred = torch.argmax(result,1)
                        a += torch.sum(pred == j)
                        loss = torch.nn.functional.cross_entropy(result, j, size_average = 
                        opt.zero_grad()
                        loss.backward()
                        opt.step()
                        loss_list[epoch] += loss
                loss_list[epoch] = loss_list[epoch]/len(dataload.dataset)
                a = a.to(dtype=torch.float)/len(dataload.dataset)
                if a>0.99:
                        return loss_list, W0, W1, b0, b1
        return loss_list, W0, W1, b0, b1

loss_list_ws, W0_ws, W1_ws, b0_ws, b1_ws = trainws(train, 0.001, 500, 64, 784)

def traindn(dataload, LR, epochs, h , m):
    alpha = 1/np.sqrt(m)
    W0 = -2*alpha* torch.rand(h, m) + alpha
    W0.requires_grad = True
    W1 = -2*alpha* torch.rand(h, h) + alpha
    W1.requires_grad = True
    W2 = -2*alpha* torch.rand(10, h) + alpha
    W2.requires_grad = True
    b0 = -2*alpha* torch.rand(h) + alpha
    b0.requires_grad = True
    b1 = -2*alpha* torch.rand(h) + alpha
    b1.requires_grad = True
    b2 = -2*alpha* torch.rand(10) + alpha
    b2.requires_grad = True
    p = [W0, W1, W2, b0, b1, b2]
    opt = torch.optim.Adam(p, LR)
    loss_list = []
    for i in range(epochs):
        loss_list.append(0)
        a = 0
        for i, j in tqdm(iter(dataload)):
            i = torch.flatten(i, 1, 3)
            result = dn(i, W0, W1, W2, b0, b1, b2)
```

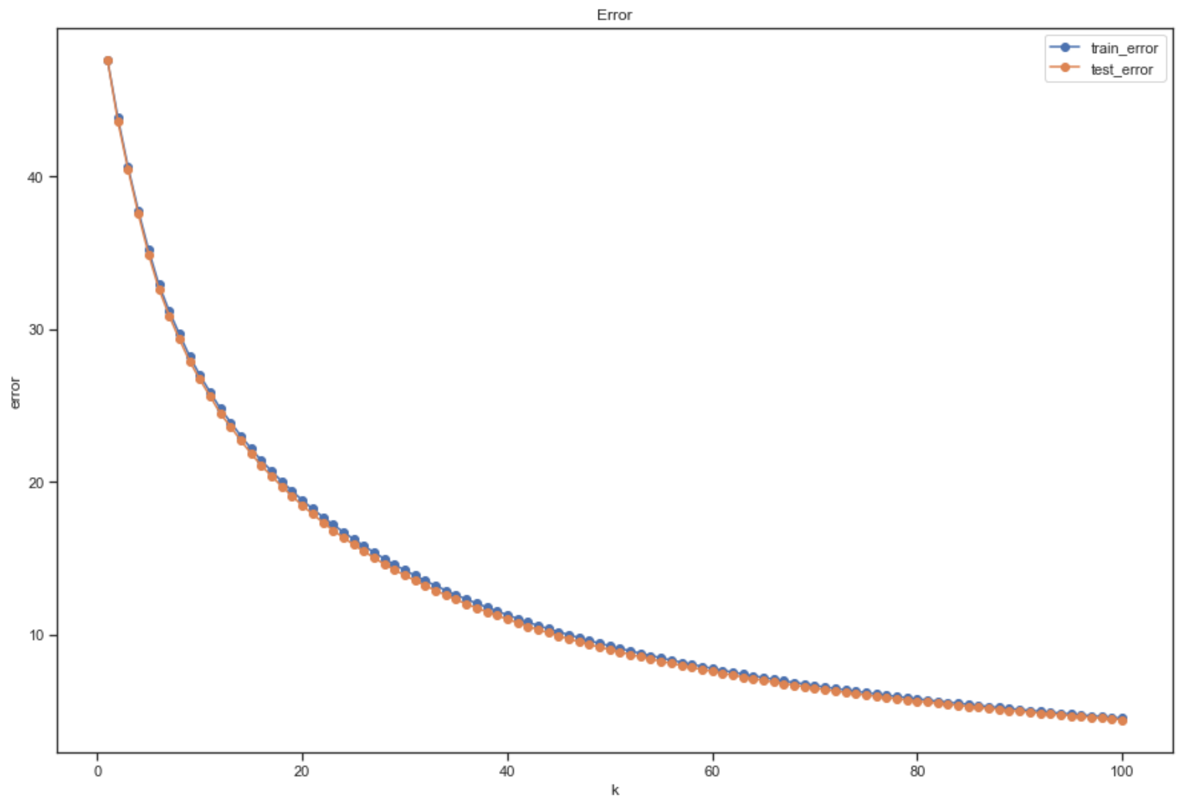
```
            pred = torch.argmax(result,1)
            a += torch.sum(pred == j)
            loss = torch.nn.functional.cross_entropy(result, j, size_average = False)
            opt.zero_grad()
            loss.backward()
            opt.step()
            loss_list[i] += loss
            loss_list[i] = loss_list[i]/len(dataload.dataset)
            a = a.to(dtype=torch.float)/len(dataload.dataset)
    if a>0.99:
        return loss_list, W0, W1, W2, b0, b1, b2
    return loss_list, W0, W1, W2, b0, b1, b2


#a
loss_list_ws, W0_ws, W1_ws, b0_ws, b1_ws = trainws(train, 0.001, 500, 64, 784)
print(loss_list_ws, W0_ws, W1_ws, b0_ws, b1_ws)
ws_acc, ws_loss = accuracyandloss(test, net_type ='wide', W0=W0_ws, W1=W1_ws, W2=None, b0=b(
print('accuracy for wide: ', ws_acc)
print('loss for wide: ', ws_loss)
plt.plot(range(len(loss_list_ws)), loss_list_ws, '-o', label = 'wide and shallow')
plt.xlabel('epoch')
plt.ylabel('loss')
#b
loss_list_dn, W0_dn, W1_dn, W2_dn, b0_dn, b1_dn, b2_dn = traindn(train, 0.001, 500, 32, 784)
dn_acc, dn_loss = accuracyandloss(test, net_type ='deep', W0=W0_dn, W1=W1_dn, W2=W2_dn, b0=b
print('accuracy for wide: ', dn_acc)
print('loss for wide: ', dn_loss)
plt.plot(range(len(loss_list_dn)), loss_list_dn, '-o', label = 'deep and narrow')
plt.xlabel('epoch')
plt.ylabel('loss')
#c
numberws = np.prod(W0_ws.shape) + np.prod(W1_ws.shape) + np.prod(b0_ws.shape) + np.prod(b1_w
numberdn = np.prod(W0_dn.shape) + np.prod(W1_dn.shape) + np.prod(W2_dn.shape) + np.prod(b0_d
print(numberws, numberdn)
```

5 PCA

A6

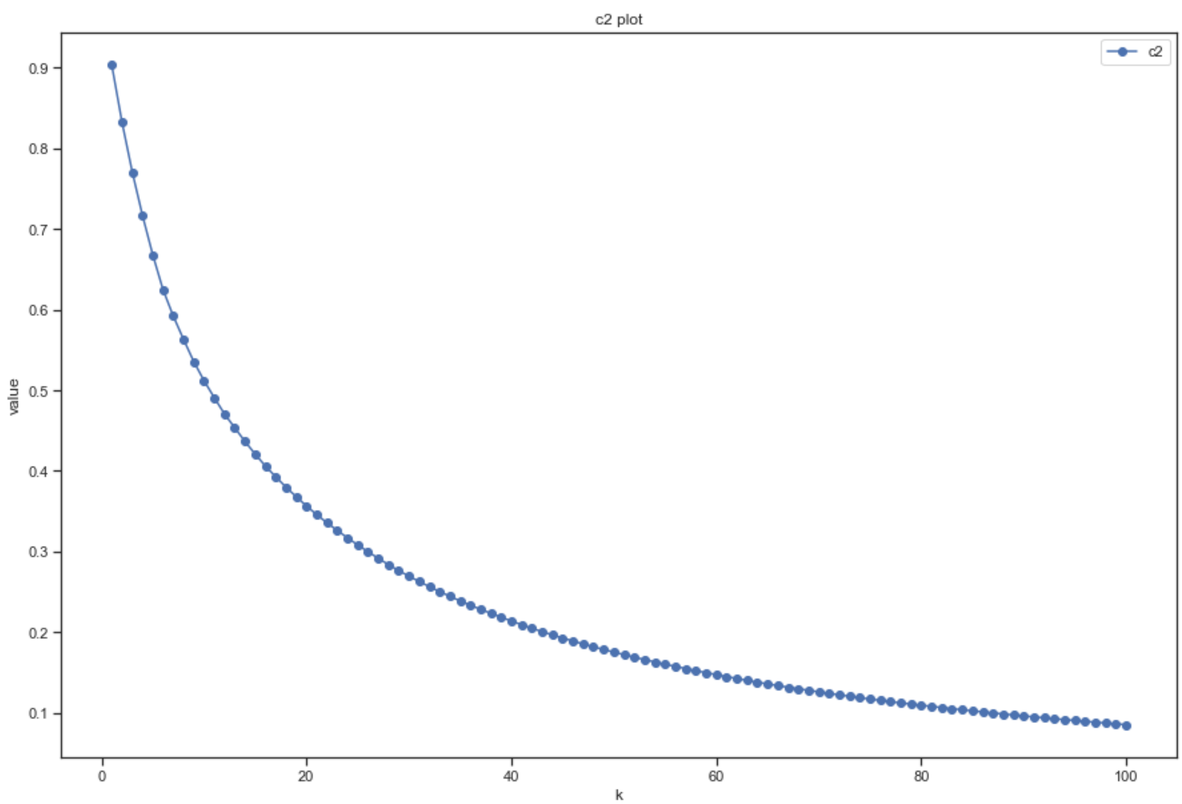
a. $[5.11678773+0.j3.74132848+0.j1.24272938+0.j0.36425572+0.j0.16970843+0.j]$ are eigenvalues and sum is 52.72503549512699+0j

b.
$$x^T = \mu^T + (x-\mu)^T V_k V_k^T$$

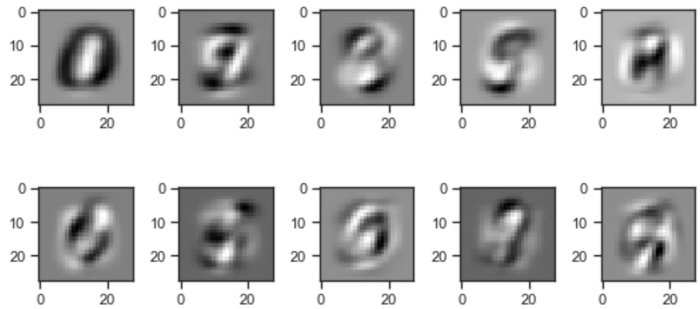


Error
train_error
test_error
40
30
20
10
error
0
20
40
60
80
100
k

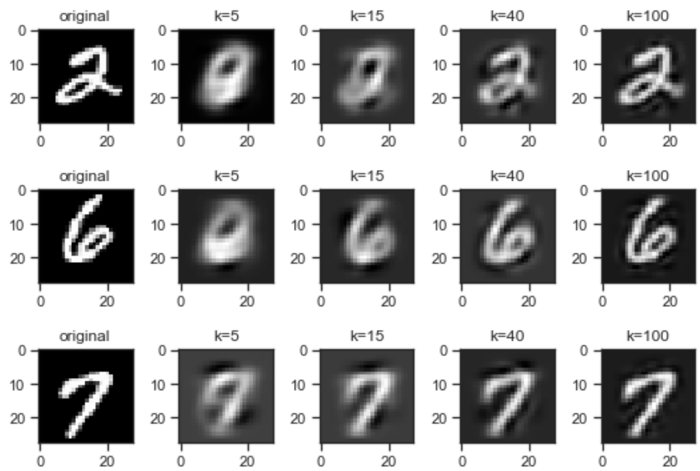
c.



c2 plot
c2
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
value
0
20
40
60
80
100
k

d. 

They include some basic characteristics, especially the first one which is for 0.

e. 

Listing 4: A6code

```
import numpy as np
import matplotlib.pyplot as plt
from mnist import MNIST

def load_dataset():
    mndata = MNIST('./data/')
    mndata.gz=True
    X_train, labels_train = map(np.array, mndata.load_training())
    X_test, labels_test = map(np.array, mndata.load_testing())
    X_train = X_train/255.0
    X_test = X_test/255.0
    return X_train,labels_train, X_test, labels_test


def Cov(X):
    mu = np.mean(X, axis = 0)
    return mu, (X-mu).T.dot(X-mu)/len(X)


X_train, y_train, X_test, y_test = load_dataset()
mu, Sigma = Cov(X_train)
l, V = np.linalg.eig(Sigma)
a_lambda = [0,1,9,29,49]
print(np.sort(l)[::-1][a_lambda], np.sum(l))
```

```python
sorted_indices = np.argsort(l)[::-1]
l = l[sorted_indices].astype('float')
V = V[:,sorted_indices].astype('float')

train_error = []
test_error = []
k_range = np.arange(1,101)
for k in k_range:
    train_pred = (X_train - mu).dot(V[:,:k]).dot(V[:,:k].T) + mu
    train_error.append(np.mean(np.linalg.norm(X_train - train_pred, axis = 1)**2))
    test_pred = (X_test - mu).dot(V[:,:k]).dot(V[:,:k].T) + mu
    test_error.append(np.mean(np.linalg.norm(X_test - test_pred, axis = 1)**2))

plt.figure(figsize = (15,10))
plt.plot(k_range, train_error, '-o', label = 'train_error')
plt.plot(k_range, test_error, '-o', label = 'test_error')
plt.title('Error')
plt.legend()
plt.xlabel('k')
plt.ylabel('error')
plt.show()

c2 = []
for k in k_range:
    c2.append(1 - np.sum(l[:k])/np.sum(l))

plt.figure(figsize = (15,10))
plt.plot(k_range, c2, '-o', label = 'c2')
plt.title('c2 plot')
plt.legend()
plt.xlabel('k')
plt.ylabel('value')
plt.show()




fig, axes = plt.subplots(2, 5, figsize=(1.5*5,2*2))
for i, axe in enumerate(axes.flatten()):
    axe.imshow(V[:,i].reshape(28,28), cmap='gray')
plt.tight_layout()
plt.show()


def display_digit_reconstruction(digit_num):
    digit = X_train[y_train == digit_num][0]
    k_range = [5, 15, 40, 100]
    names = ['original'] + ['k='+str(k) for k in k_range]
    recons = [digit] + [(digit - mu).dot(V[:,:k]).dot(V[:,:k].T) + mu for k in k_range]
    fig, axes = plt.subplots(1, 5, figsize=(1.5*5,2*1))
    for i, axe in enumerate(axes.flatten()):
        axe.imshow(recons[i].reshape(28,28), cmap='gray')
        axe.set_title(names[i])
```

```python
    plt.tight_layout()
    plt.show()
display_digit_reconstruction(2)
display_digit_reconstruction(6)
display_digit_reconstruction(7)
```