Homework #4

CSE 446/546: Machine Learning
Joseph David, worked with Kevin Liu
Due: **Friday** June 4, 2021 11:59pm Pacific Time

Conceptual Questions

A1. The answers to these questions should be answerable without referring to external materials. Briefly justify your answers with a few words.

a. This is true. If our data matrix has rank k, then projecting the data onto any rank k subspace that contains each data point will be able to achieve zero reconstruction error. Since PCA achieves the optimal reconstruction error, it must produce zero error.

b. This is false. We see that $X^TX = VS^2V^T$, or alternatively that $X^TXV = S^2V$. Thus it is actually the columns of V that are the eigenvectors of X^TX.

c. This is false. The choice of k should reflect some knowledge of the data (i.e. a reasonable choice for a number of clusters of the data). For instance, choosing k very large would decrease error by making each point its own cluster, which is not meaningful.

d. This is false, since we can reorder the singular value matrix S, as well as U, V appropriately to get a different decomposition.

e. This is false. For instance one can have a matrix with rank 1 but with all zero eigenvalues.

f. This is true. Since a neural network with a linear map from $\mathbb{R}^d \to \mathbb{R}^k$ and $\mathbb{R}^k \to \mathbb{R}^d$ is a generalization of PCA, it will capture at least as much variance as PCA. The nonlinear activation function will allow it to capture nonlinearities in the data.

Basics of SVD and subgradients

A2.

a. (a) We know from previously in this course that $\widehat{w} = (X^TX)^{-1}X^Ty$, whereas $\widehat{w}_R = (X^TX + \lambda I)^{-1}X^Ty$. Therefore, if $X = U\Sigma V^T$ is the SVD decomposition of X, then we see that

$$\begin{aligned}\widehat{w}_R &= ((U\Sigma V^T)^T(U\Sigma V^T) + \lambda I)^{-1}(U\Sigma V)^Ty \\ &= (V\Sigma U^TU\Sigma V^T + \lambda I)^{-1}(V\Sigma U^T)y \\ &= (V\Sigma^2V^T + \lambda I)^{-1}(V\Sigma U^T)y \\ &= V(\Sigma^2 + \lambda I)^{-1}V^T(V\Sigma U^T)y \\ &= V(\Sigma^2 + \lambda I)^{-1}\Sigma U^Ty.\end{aligned}$$

Thus, we see that $\widehat{w}_R$ is the image of y under the map $V(\Sigma^2 + \lambda I)^{-1}\Sigma U^T$, which is an SVD decomposition. Thus, if $\Sigma = diag(\sigma_1, \ldots, \sigma_k)$, then looking at the singular value matrix in the decomposition, we see that

$$(\Sigma^2 + \lambda I)^{-1}\Sigma = diag\left(\frac{\sigma_1}{\sigma_1^2 + \lambda}, \ldots, \frac{\sigma_n}{\sigma_n^2 + \lambda}\right).$$

Therefore, as λ increases, we see that the singular values here decrease and thus the ridge regression solution $\widehat{w}_R$ will "shrink".

(b) Let $U = W\Sigma V^T$ be the SVD decomposition for U. By definition of the SVD decomposition, we know that $WW^T = W^TW = I_n$ and $VV^T = V^TV = I_n$. Since the singular values of U are all equal to one, and since Σ is diagonal, this implies that $\Sigma = I_n$. Therefore

$$\begin{aligned}U^TU &= (W\Sigma V^T)^T\Sigma(W\Sigma V^T) \\ &= V\Sigma W^T\Sigma W\Sigma V^T \\ &= VW^TWV^T \quad (\text{since } \Sigma = I_n) \\ &= I_n.\end{aligned}$$

In addition, the above fact implies that, for any $x \in \mathbb{R}^n$,

$$||Ux||_2^2 = (Ux)^T(Ux) = x^TU^TUx = x^Tx = ||x||_2^2$$

so that U preserves Euclidean norms.

b. Compute the subgradient set of the following functions:

(a) Let $\mathbf{w} = (w_1, \ldots, w_n)$ be a subgradient vector at some $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$. Define the sign function $sgn(x) = -1$ if $x < 0$, $sgn(x) = 1$ if $x > 0$ and $sgn(x) = 0$ if $x = 0$. Then we have $w_i = sgn(x_i)$ if $x_i \neq 0$. If $x_i = 0$, then w_i can be anything in $[-1, 1]$. This characterizes all subgradients at any point $\mathbf{x} \in \mathbb{R}^n$.

(b) Let $\mathbf{x} \in \mathbb{R}^n$ and let $f_{j_1}, \ldots, f_{j_k}$ be such that $f_{j_i}(x) = f(x)$ (i.e. they attain the max over all f_i at $\mathbf{x}$). If g is a subgradient for some f_{j_k} at $\mathbf{x}$, then by definition of subgradients, for all $\mathbf{w} \in \mathbb{R}^n$,

$$f(\mathbf{w}) \geq f(\mathbf{x}) + g^T(\mathbf{x} - \mathbf{w}).$$

But then,

$$\begin{aligned}f(\mathbf{w}) &\geq f_{j_k}(\mathbf{w}) \quad (\text{since } f \text{ is the max of all such functions}) \\ &\geq f_{j_k}(\mathbf{x}) + g^T(\mathbf{x} - \mathbf{w}) \quad (\text{by def'n of subgradients}) \\ &= f(\mathbf{x}) + g^T(\mathbf{x} - \mathbf{w}).\end{aligned}$$

Thus, g is a subgradient of f at $\mathbf{x}$. Since this holds for any subgradient of any f_i that attains the maximum at $\mathbf{x}$, we see that

$$\partial f(\mathbf{x}) = \{\partial f_i(\mathbf{x}) : f_i(\mathbf{x}) = f(\mathbf{x})\}.$$

c. *[3 points]* For $i = 1, \dots, n$, define $f_i(x) = |x_i - (1 + \eta/i)|$. By part (b), we know that

$$(\partial f_i(x))_i = \begin{cases} -1 & x_i < 1 + \frac{\eta}{i} \\ 1 & x_i > 1 + \frac{\eta}{i} \\ [-1, 1] & x_i = 1 + \frac{\eta}{i} \end{cases}.$$

and $(\partial f_i(x))_j = 0$ for $j \neq i$, where $(\partial f_i(x))_j$ denotes the jth components of the subgradients. By the second part of (b), we know that all possible subgradients must have components bounded in absolute value by 1, and thus $||v||_\infty \leq 1$. Since $(1, 1, \dots, 1)$ is a subgradient vector of f at $(2 + \eta, \dots, 2 + \eta)$, we see that 1 is the best upper bound.

PCA

A3.

a. We found $\lambda_1 = 5.11679$, $\lambda_2 = 3.74133$, $\lambda_{10} = 1.24273$, $\lambda_{30} = 0.36426$, $\lambda_{50} = 0.16971$, and $\sum_{i=1}^{784} \lambda_i = 52.72504$. Below is the code used.

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

```python
def load_dataset():
    mndata = MNIST('/Users/joseph/Courses/cse546/hw2/mnist')
    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]

[X_train, labels_train, X_test, labels_test] = load_dataset()
```

```python
n=60000
d=784
mu = np.mean(X_train, axis=0) #take average across pixels
mu_matrix = np.zeros((n,d))
for j in range(d):
    mu_matrix[:,j] = mu[j]
```

```python
sigma = (np.transpose(X_train - mu_matrix) @ (X_train - mu_matrix))/n
```

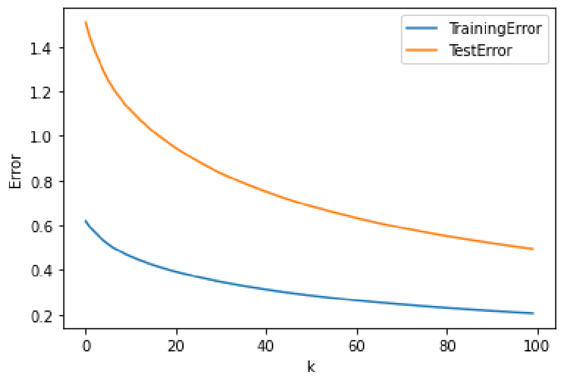
```python
eigvals,eigvecs = la.eig(sigma)
```

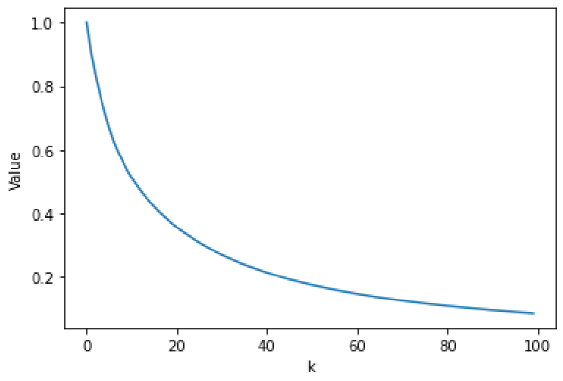
```python
print(f'''Lambda1 = {eigvals[0]}, Lambda2 = {eigvals[1]}, Lambda10 = {eigvals[9]},
    Lambda30 = {eigvals[29]}, Lambda50 = {eigvals[49]}''')
print(f'Sum of Eigenvalues = {sum(eigvals)}')
```

```
Lambda1 = 5.116787728342092, Lambda2 = 3.7413284788648316, Lambda10 = 1.2427293764173317,
    Lambda30 = 0.3642557202788918, Lambda50 = 0.16970842700672786
Sum of Eigenvalues = 52.725035495126946
```

b. Let $U_k \in \mathbb{R}^{d\times k}$ be the matrix of the first k eigenvectors of Σ. Then any example $x \in \mathbb{R}^d$ can be approximated by $\mu + U_k U_k^T(x-\mu)$.

c. Below are the plots showing the average mean squared error of the reconstructions for the train and test images, followed by the plot of $1 - \frac{\sum_1^k \lambda_i}{\sum_1^d \lambda_i}$ as a function of k.





Here is the code used to make these plots.

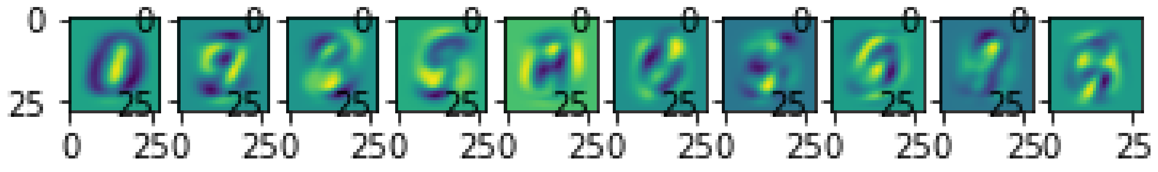
```
recon_error_train = np.zeros(100)
recon_error_test = np.zeros(100)
for k in range(100):
    Uk = eigvecs[:,0:k] #take first k eigvectors
    Uk2 = Uk@np.transpose(Uk)
    recon_train = mu_matrix + np.transpose(Uk2@np.transpose(X_train-mu_matrix))
    recon_test = mu_matrix[0:10000,:] + np.transpose(Uk2@np.transpose(X_test-mu_matrix[0:10000,:]))
    print(f'Running iteration {k}')
    recon_error_train[k] = error_test = np.sum(la.norm(recon_train - X_train, axis=0))
    recon_error_test[k] = np.sum(la.norm(recon_test - X_test, axis=0))
recon_error_train /= n
recon_error_test /= 10000
```

...

```
plt.plot(recon_error_train,label='TrainingError')
plt.plot(recon_error_test,label='TestError')
plt.legend()
plt.xlabel('k')
plt.ylabel('Error')
```

```
#A.3(c)
eig_weights = np.zeros(100)
for k in range(100):
    eig_weights[k] = 1-(np.sum(eigvals[0:k])/np.sum(eigvals))
plt.plot(eig_weights)
plt.xlabel('k')
plt.ylabel('Value')
```

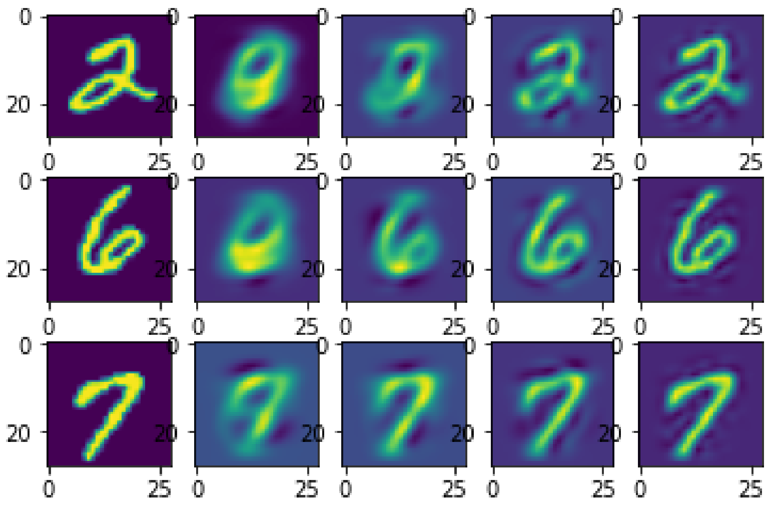
d. Below are the first 10 eigenvectors reshaped and plotted. We see that each of them capture a vague sense of some of the digits, but each eigenvector has elements of multiple digits. For instance, in the fourth eigenvector one can see a ”5” and perhaps also a ”6”.



Here is the code used to plot these images.

```
#A.3(d)
images = [eigvecs[:,j].reshape((28,28)) for j in range(10)]
im, arr = plt.subplots(1,10)
for j in range(10):
    arr[j].imshow(images[j])
```

e. Below are the reconstructed images. On the left are the true images, and on the right are the approximations ordered by increasing value of k. We see that with $k = 5$ it is difficult to determine the ”2” and ”6”, but the ”7” is already quite clear. By $k = 15$, only ”2” is ambiguous and by $k = 40$ they are all quite easy to determine.



Here is the code used to plot the reconstructions along with the true images.

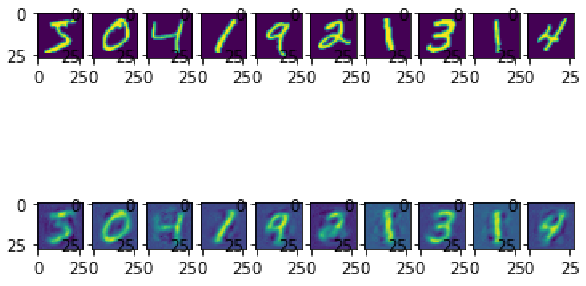
```
#find indices of images of 2, 6, and 7
index2=0
index6=0
index7=0
for j in range(100):
    if labels_train[j]==2:
        index2=j
    elif labels_train[j]==6:
        index6=j
    elif labels_train[j]==7:
        index7=j
    if index2 != 0 and index6 != 0 and index7 != 0:
        break
```

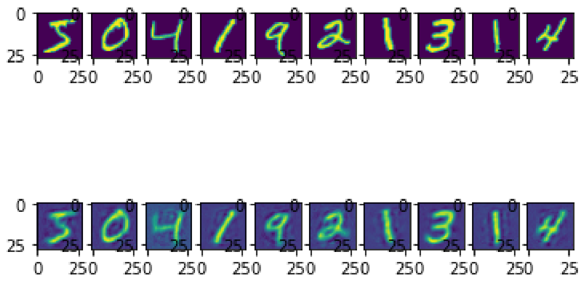
```
#A,3(e)
two_true = X_train[index2,:].reshape((28,28))
six_true = X_train[index6,:].reshape((28,28))
seven_true = X_train[index7,:].reshape((28,28))
k_values = [5,15,40,100]
im, arr = plt.subplots(3,5)
arr[0,0].imshow(two_true)
arr[1,0].imshow(six_true)
arr[2,0].imshow(seven_true)
for i in range(4):
    Uk = Uk = eigvecs[:,0:k_values[i]]
    two_approx = (mu + np.transpose((Uk@np.transpose(Uk))@(X_train[index2,:]-mu))).reshape((28,28))
    six_approx = (mu + np.transpose((Uk@np.transpose(Uk))@(X_train[index6,:]-mu))).reshape((28,28))
    seven_approx = (mu + np.transpose((Uk@np.transpose(Uk))@(X_train[index7,:]-mu))).reshape((28,28))
    arr[0,i+1].imshow(two_approx)
    arr[1,i+1].imshow(six_approx)
    arr[2,i+1].imshow(seven_approx)
```

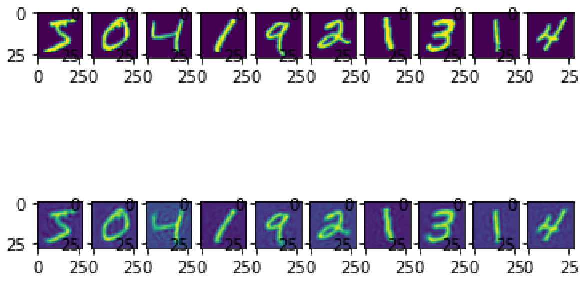
Unsupervised Learning with Autoencoders

A4.

a. We used `lr = .001`, `momentum=0.9`, and `nn.MSELoss()`, and trained the model for 500 epochs for all values of h. For $h = 32$, the final training loss was $2.20 \cdot 10^{-2}$, for $h = 64$ it was $1.23 \cdot 10^{-2}$, and for $h = 128$ it was $6.29 \cdot 10^{-3}$. Here are the reconstructed images for $h = 32$ (left), 64 (right), 128 (center bottom). We see that as h increases, the loss becomes smaller and the reconstructed images are much more clear.







Code:

```
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F
```

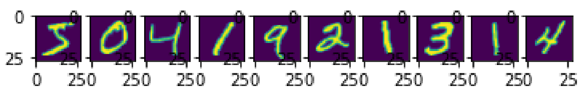
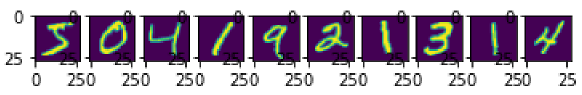
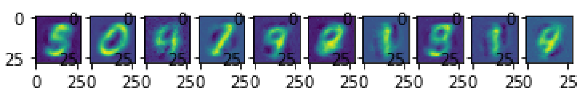
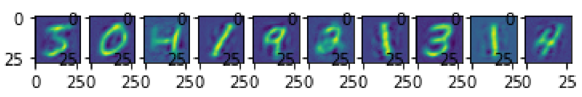
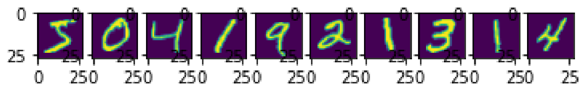
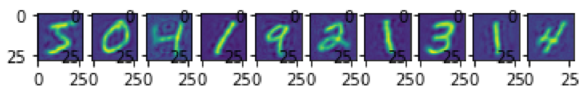
```
mnist = fetch_openml('mnist_784', cache=False)
X = mnist.data.astype('float32')
Y = mnist.target.astype('int64')
X /= 255.0

X_train = X[0:60000,:]
labels_train = Y[0:60000]
X_test = X[60000:,:]
labels_test = Y[60000:]
```

```
X_train = torch.from_numpy(X_train).float().cuda()
labels_train = torch.from_numpy(labels_train).long().cuda()
X_test = torch.from_numpy(X_test).float().cuda()
labels_test = torch.from_numpy(labels_test).long().cuda()
```

```
#A.4(a) model 1
n, n2, d= 60000, 10000, 784
h=32

model1 = nn.Sequential(nn.Linear(d,h), nn.Linear(h,d)).to('cuda')
optimizer = torch.optim.Adam(model1.parameters(),lr=.001)
loss = nn.MSELoss()

for epoch in range(500):
  loss_epoch=0
  prediction = model1(X_train)
  J = loss(prediction, X_train)
  model1.zero_grad()
  J.backward()
  optimizer.step()
  print(f'Experiment h={h} loss for epoch {epoch} = {J.item()}')
```

```
images = [X_train[j,:] for j in range(10)]
im, arr = plt.subplots(2,10)
for i in range(10):
  arr[0,i].imshow(images[i].view(28,28).cpu().detach())
  arr[1,i].imshow((model1(images[i])).view(28,28).cpu().detach())
```

b. We used the same hyperparameters as in (a), and trained for 500 epochs. The final training errors for $h = 32, 64$, and 128 respectively were $3.81 \cdot 10^{-2}$, $2.64 \cdot 10^{-2}$, and $9.33 \cdot 10^{-3}$. Below are the true images alongside the reconstructions. We see that for smaller model (b) does not perform quite as well as model (a) for all values of h.

Code:

```
#A.4(b) model 2
h, d = 128, 784

model2 = nn.Sequential(nn.Linear(d,h), nn.ReLU(), nn.Linear(h,d)).to('cuda')
optimizer = torch.optim.Adam(model2.parameters(),lr=.001)
loss = nn.MSELoss()

for epoch in range(500):
  loss_epoch=0
  prediction = model2(X_train)
  J = loss(prediction, X_train)
  model2.zero_grad()
  J.backward()
  optimizer.step()
  print(f'Experiment h={h} loss for epoch {epoch} = {J.item()}')
```

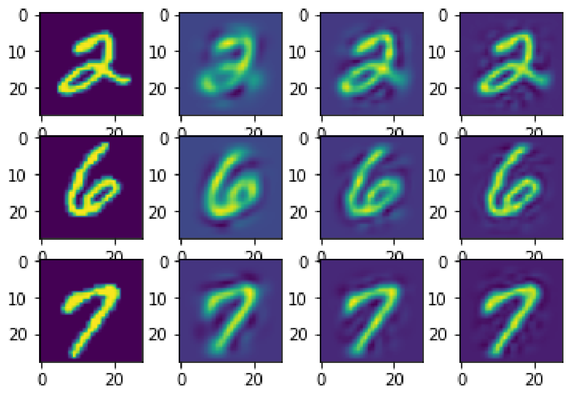
```
images = [X_train[j,:] for j in range(10)]
im, arr = plt.subplots(2,10)
for i in range(10):
  arr[0,i].imshow(images[i].view(28,28).cpu().detach())
  arr[1,i].imshow((model2(images[i])).view(28,28).cpu().detach())
```

c. We found that the test error for model (a) was $6.03 \cdot 10^{-3}$ and for model (b) it was $8.70 \cdot 10^{-3}$. Code:

```
#A.4(c)
predictions_model1 = model1(X_test)
predictions_model2 = model2(X_test)
test_error_model1 = loss(predictions_model1,X_test)
test_error_model2 = loss(predictions_model2, X_test)
print(f'Test Error for Model 1 = {test_error_model1}')
print(f'Test Error for Model 2 = {test_error_model2}')
```

```
Test Error for Model 1 = 0.006027239840477705
Test Error for Model 2 = 0.00870056077837944
```

d. We re-ran the PCA algorithm from A.3 with $k = 32, 64, 128$. The results are below. We see that when $k = 32$, the results using PCA are better than both trained networks models. When $k = 64$, model (a) and PCA perform about as well as each other, whereas the reconstructions for model (b) are less clear. When $k = 128$, the reconstructions all seem to have the same level of quality.



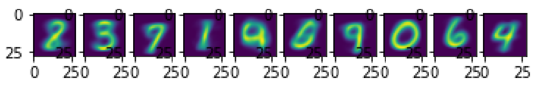
k-means clustering

A5.

a. Below is the code implementing Lloyd's algorithm.

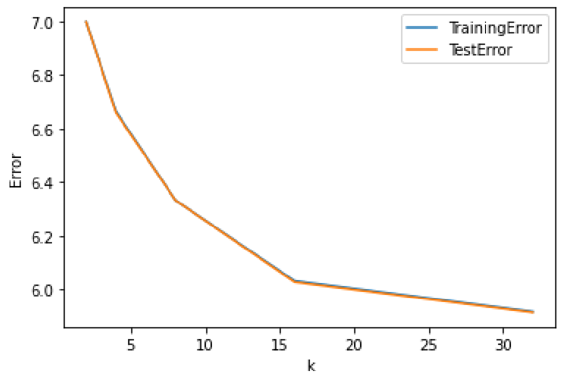
```
def lloyd(data,k):
    n, d = np.size(data,0), np.size(data,1)
    change = 1
    centroids = np.random.rand(k,d)  #stores centroids k centroid points as matrix
    close_centroid = 0  #will store which centroid is closest to given point
    for iter in range(10):
        num_points = np.zeros(k)  #how many points are closest to each centroid
        centroids_new = np.zeros((k,d))
        for j in range(n):
            x = data[j,:]  #stores data point
            min_dist = 100 #to determine which centroid x is closest to
            for i in range(k):
                distance = la.norm(x-centroids[i,:])
                if (distance < min_dist):
                    min_dist = distance
                    close_centroid = i
            centroids_new[close_centroid,:] += x
            num_points[close_centroid] += 1   #add 1 to counter for that centroid
        for j in range(k):
            centroids_new[j,:] /= num_points[j]  #average each centroid
        change = np.sum([la.norm(centroids[i,:] - centroids_new[i,:]) for i in range(k)]) / k
        #copy centroids_new into centroids
        centroids = centroids_new
        plt.figure(iter)
        im, arr = plt.subplots(1,10)
        for i in range(10):
            arr[i].imshow(np.reshape(centroids_new[i,:], (28,28)))
```

b. Below are the ten centers using Lloyd's algorithm for 10 iterations. We see that most of them look like a digit, with each digit being clearly visible except for "5".



```
n, d = 60000, 784
k = 10
lloyd(X_train,k)
```

c. Below is the figure and code.



```
train_errors = np.zeros(5)
test_errors = np.zeros(5)
kmeans_2 = lloyd(X_train,2)
kmeans_4 = lloyd(X_train,4)
kmeans_8 = lloyd(X_train,8)
kmeans_16 = lloyd(X_train,16)
kmeans_32 = lloyd(X_train,32)
```

```
for j in range(n):
    x = X_train[j,:]
    distance = np.min([la.norm(x-kmeans_2[i,:]) for i in range(2)])
    train_errors[0] += distance
    distance = np.min([la.norm(x-kmeans_4[i,:]) for i in range(4)])
    train_errors[1] += distance
    distance = np.min([la.norm(x-kmeans_8[i,:]) for i in range(8)])
    train_errors[2] += distance
    distance = np.min([la.norm(x-kmeans_16[i,:]) for i in range(16)])
    train_errors[3] += distance
    distance = np.min([la.norm(x-kmeans_32[i,:]) for i in range(32)])
    train_errors[4] += distance
train_errors /= n

for j in range(10000):
    x = X_test[j,:]
    distance = np.min([la.norm(x-kmeans_2[i,:]) for i in range(2)])
    test_errors[0] += distance
    distance = np.min([la.norm(x-kmeans_4[i,:]) for i in range(4)])
    test_errors[1] += distance
    distance = np.min([la.norm(x-kmeans_8[i,:]) for i in range(8)])
    test_errors[2] += distance
    distance = np.min([la.norm(x-kmeans_16[i,:]) for i in range(16)])
    test_errors[3] += distance
    distance = np.min([la.norm(x-kmeans_32[i,:]) for i in range(32)])
    test_errors[4] += distance
test_errors /= 10000
```

```
plt.plot([2,4,8,16,32],train_errors, label="TrainingError")
plt.plot([2,4,8,16,32],test_errors,label='TestError')
plt.xlabel('k')
plt.ylabel('Error')
plt.legend()
```

Think about real-world scenario

A6.

a. The issue that is most apparent to me in this problem is separating features that cause a disease from those that are a result or are simply correlated with the disease. For this reason, I would suggest using LASSO to find a good model for this problem. I would first find a good value for λ using a validation set and from there fit the model. This would allow us to learn a model with most parameters equal to zero except for a few features that are most correlated to the presence of the given disease. This model could then be given a new person's personal data and compute the likelihood that they have the disease.

b. First, I would take the image data and convert each into a vector, so that I can store all the images as a matrix, and then separate it into a training and validation set. Then, I would normalize each image with mean 0 and standard deviation 1. Then, I would go through the images and crop out all rectangular portions and label the ones corresponding to "key features" (or label as nothing if it is not a key feautre). I would then train a neural network on these cropped images using a combination of linear and nonlinear layers, as well as convolutional layers, that would output probabilities that a given image is of some feature. After training the model so that it achieves a desirable validation accuracy, it would be able to be used on the test data. For this, I would take an image of a face and go through the image and crop small rectangular portions of the image meant to capture a feature. For each cropping, I would pass it through the trained neural network and it would determine the probability that it is a given key feature, or if it is nothing. Thus, if a cropped image is classified as a key feature, we will know where in the image it is.

c. First, I would need to determine a good way to encode the features that are categorical. For this I would use one-hot encoding if the number of distinct values is not too large. If it is, then I would think of a way of possibly compressing the values that does not lose too much information. Then, I would use Principal Component Analysis to reduce the dimensionality of the data and ensure that I could train a model fast enough. From there I would train a neural network using linear and convolutional layers, that would take in the attributes of some file metadata and return a likelihood that it is malicious or not. However before that I would use a validation set to tune the hyperparameters of learning rate and batch size in order to train the model using Stochastic Gradient Descent. After achieving the desired validation accuracy, I would train the model on the training data using the tested hyperparameters and it would output whether it is likely malicious.

Matrix Completion and Recommendation System

B1.

a. The formula for $\widehat{R} \in \mathbb{R}^{1\times 1682}$ is given by: the ith row of $\widehat{R}$ is the average of all ratings of the ith movie, i.e sum of the ith row of R divided by the number of nonzero entries of the ith row of R. We calculated the error of this prediction and got an accuracy of 2.26. Code:

```
# B.1
```

```
import csv
import numpy as np
from numpy import linalg as la
import scipy
from scipy import linalg
import matplotlib.pyplot as plt
import math
```

```
data = []
with open('u.data') as csvfile:
    spamreader = csv.reader(csvfile, delimiter='\t')
    for row in spamreader:
        data.append([int(row[0])-1, int(row[1])-1, int(row[2])])
data = np.array(data)

num_observations = len(data) # num_observations = 100,000
num_users = max(data[:,0])+1 # num_users = 943, indexed 0,...,942
num_items = max(data[:,1])+1 # num_items = 1682 indexed 0,...,1681

np.random.seed(1)
num_train = int(0.8*num_observations)
perm = np.random.permutation(data.shape[0])
train = data[perm[0:num_train],:]
test = data[perm[num_train::],:]
```

```
def calc_error(R,R_hat):
    return np.nanmean((R-R_hat)**2)
```

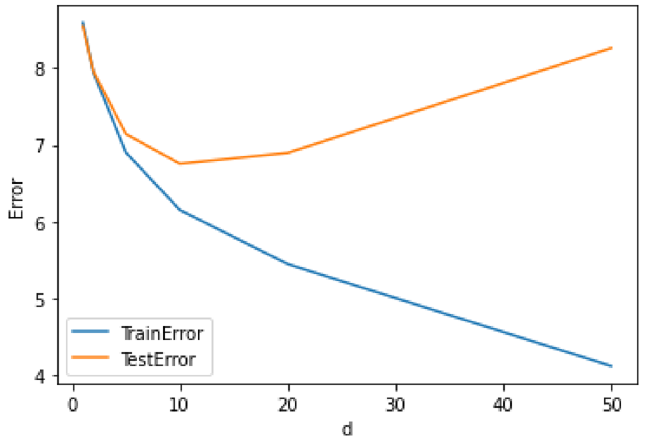
```
# create spare matrix of user movie ratings
R = np.zeros((num_items,num_users))
for i in range(len(train)):
    R[train[i,1],train[i,0]] = train[i,2]
for i in range(num_items):
    for j in range(num_users):
        if R[i,j] == 0:
            R[i,j] = np.nan

R_test = np.zeros((num_items,num_users))
for i in range(len(test)):
    R_test[test[i,1],test[i,0]] = test[i,2]
for i in range(num_items):
    for j in range(num_users):
        if R_test[i,j] == 0:
            R_test[i,j] = np.nan
```

```
#B.1(a)
num_ratings = np.zeros(num_items) #store how many ratings each movie has
avg_ratings = np.zeros(num_items) #store each movies average ratings
for i in range(len(train)):
    avg_ratings[train[i,0]] += train[i,2]
    num_ratings[train[i,0]] += 1
for i in range(len(num_ratings)):
    if num_ratings[i] == 0:
        num_ratings[i] = 1
avg_ratings /= num_ratings
avg_ratings_mtx = np.zeros((num_items,num_users))
for i in range(num_users):
    avg_ratings_mtx[:,i] = avg_ratings
```

```
print(calc_error(R_test,avg_ratings_mtx))
```

```
2.261540452277984
```

b. Below is the figure showing MSE for different values of d. We see that the test error has a minimum at $d = 10$. The code is after the figure.



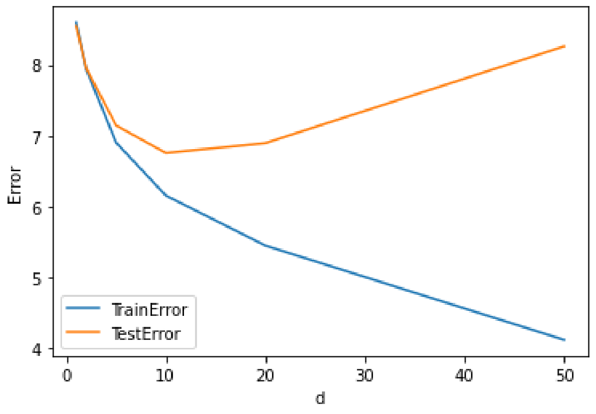
```
R_til = np.zeros((num_items,num_users))
for i in range(len(train)):
    R_til[train[i,1],train[i,0]] = train[i,2]
d_values = [1,2,5,10,20,50]
train_error=[]
test_error=[]
U,S,V = linalg.svd(R_til,full_matrices=False)
S = np.diag(S)
for d in d_values:
    R_hat = U[:,0:d] @ S[0:d,0:d] @ V[0:d,:]
    train_error.append(calc_error(R,R_hat))
    test_error.append(calc_error(R_test,R_hat))
```

```
plt.plot(d_values,train_error,label='TrainError')
plt.plot(d_values,test_error,label='TestError')
plt.xlabel('d')
plt.ylabel('Error')
plt.legend()
```

c. Since at each step we are either fixing $\{u_i\}$ or $\{v_i\}$, the loss function has the same gradient as ridge regression and therefore, in the case of fixing $\{v_i\}$, we have

$$U = (V^T V + \lambda I)^{-1} V^T \widetilde{R}^T.$$

The case when $\{u_i\}$ is fixed is analogous. We used hyperparameters $\sigma = \sqrt{5/d}$ (to ensure that the randomized predictions are somewhere between 0 and 5) and $\lambda = 0.1$. Below is the figure, and the code is after. Note that we get almost the same results as in part (b).



```
# B.1 (c)
```

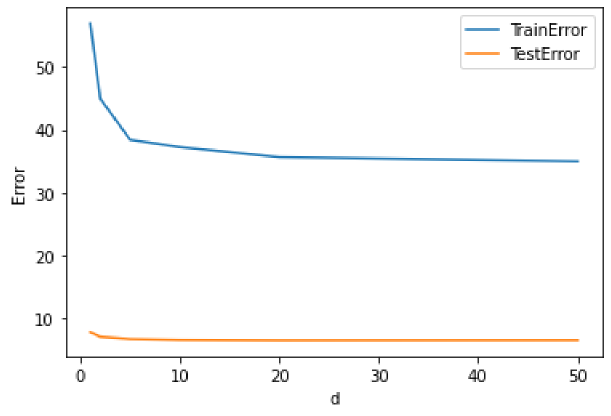
```
def loss_func(R,U,V,lam):
    return np.nansum((U@np.transpose(V) - R)**2) + lam*(la.norm(U)**2 + la.norm(V)**2)

def trainer(R,R_til,d,lam,sigma):
    U = sigma*np.random.rand(num_items,d)
    V = sigma*np.random.rand(num_users,d)
    I = np.eye(d)
    loss = loss_func(R,U,V,lam)
    diff = 21
    while diff > .1:
        new_U = np.transpose(la.inv(np.transpose(V)@V + lam*I) @ np.transpose(V)@np.transpose(R_til))
        new_V = np.transpose(la.inv(np.transpose(new_U)@new_U + lam*I) @ np.transpose(new_U)@R_til)
        diff = max([la.norm(U-new_U), la.norm(V-new_V)])
        V = new_V
        U = new_U
    loss = loss_func(R,U,V,lam)
    return U,V,loss
```

```
d_values = [1,2,5,10,20,50]
train_error=[]
test_error=[]
for d in d_values:
    U,V,loss = trainer(R,R_til,d,0.1,math.sqrt(5/d))
    train_error.append(calc_error(R,U@np.transpose(V)))
    test_error.append(calc_error(R_test,U@np.transpose(V)))
    print(f'Finished Running Experiment: {d}')
```

...

```
plt.plot(d_values,train_error,label='TrainError')
plt.plot(d_values,test_error,label='TestError')
plt.xlabel('d')
plt.ylabel('Error')
plt.legend()
```

d. We used the same hyperparameters along with a batch size of 20,000. I believe there is a mistake in the error calculation function `calcerrortor` that did not transfer from the original function because the results are clearly incorrect. Below is the figure and code.



```
#B1(d)
import torch
train_torch = torch.tensor(train)
test_torch = torch.tensor(test)
R_torch = torch.zeros(R.shape)
R_test_torch = torch.tensor(R_test)
R_til_torch = torch.tensor(R_til)
def loss_func_tor(R_torch,U,V,lam):
    return torch.nansum((torch.mm(U,torch.transpose(V,0,1)) - R_torch)**2) + lam*(torch.linalg.norm(U)**2 + torcl
def calc_error_tor(R,R_hat,batch_size):
    return torch.nansum(torch.square(R-R_hat)) / batch_size
def trainer_SGD(R_torch,R_til_torch,d,lam,sigma,batch_size):
    U = sigma*torch.rand((num_items,d),requires_grad=True)
    V = sigma*torch.rand((num_users,d),requires_grad=True)
    for i in range(100):
        batch = torch.randint(0,train_torch.shape[0],(batch_size,))
        for i in range(batch_size):
            R_torch[train[batch[i],1],train[batch[i],0]] = R[train[batch[i],1],train[batch[i],0]]
        loss = loss_func_tor(R_torch,U,V,lam)
        U.retain_grad()
        V.retain_grad()
        loss.backward(retain_graph=True)
        with torch.no_grad():
            U -= (.001/(i+1))*U.grad
            V -= (.001/(i+1))*V.grad
            U.grad.zero_()
            V.grad.zero_()
    return U,V,loss.item()
```

```
d_values = [1,2,5,10,20,50]
train_error=[]
test_error=[]
for d in d_values:
    U,V,loss = trainer_SGD(R_torch,R_til_torch,d,0.1,math.sqrt(5/d),20000)
    train_error.append(calc_error_tor(R_torch,torch.mm(U,torch.transpose(V,0,1)),80000))
    test_error.append(calc_error_tor(R_test_torch,torch.mm(U,torch.transpose(V,0,1)),20000))
    print(f'Finished Running Experiment: {d}')
```

...

```
plt.plot(d_values,train_error,label='TrainError')
plt.plot(d_values,test_error,label='TestError')
plt.xlabel('d')
plt.ylabel('Error')
plt.legend()
```

e. I believe what we should have seen is that using stochastic gradient descent allows us to control how much we overfit the data, whereas using an closed-form solution will always overfit the data. Thus in part (d) we should have seen a much better test error than in part (c).