CSE546 - Homework # 4 - Solutions

Cheng-Yen Yang

June 9, 2021

A.1

A.1.a

True. Given the situation in the statement, the projection onto a k dimensional subspace using PCA will have zero reconstruction error as there are only k non-zero eigen-values in the covariance matrix.

A.1.b

False. The **columns** of $\mathbf{V}$ or the **rows** of $\mathbf{V}^T$ are equal to the eigen-vectors of $\mathbf{X}^T\mathbf{X}$.

A.1.c

False. If we ought to minimize the k-means objective function by choosing different k, we will find that when $k = n$, the objective function will gives 0 value which is totally meaningless as we are having each cluster with only one data point.

A.1.d

False. The singular value decomposition (SVD) of a given matrix $\mathbf{A}$ is not always unique. Let $\mathbf{USV}^T$ be the SVD of matrix $\mathbf{A}$, only the singular values are unique. For distinct positive singular values $S_{jj} > 0$, the j-th corresponding column of $\mathbf{U}$ and $\mathbf{V}$ are not unique because a sign change of both columns also yield the SVD: $\mathbf{A} = \mathbf{USV}^T$.

A.1.e

True. The rank of a square matrix equals the number of its nonzero eigenvalues.

A.1.f

False. PCA will capture more (or all) of the variance of the data in its principal components in comparison to the encoded representation from auto-encoders with nonlinear activation.

A.2.

A.2.a

(a)

From previous lecture notes and homeworks, the solution $\widehat{w}$ of the standard regression problem is:

$$\widehat{w} = (X^\top X)^{-1} X^\top y \tag{1}$$

with the singular value decomposition $X = U\Sigma V^\top$ we can simplify the solution $\widehat{w}$:

$$\begin{aligned}
\widehat{w} &= (X^\top X)^{-1} X^\top y && (2)\\
&= ((U\Sigma V^\top)^\top (U\Sigma V^\top))^{-1} (U\Sigma V^\top)^\top y && (3)\\
&= (V\Sigma U^\top U\Sigma V^\top)^{-1} (V\Sigma U^\top) y && (4)\\
&= V\Sigma^{-2} V^\top V\Sigma U^\top y && (5)\\
&= V\Sigma^{-1} U^\top y. && (6)
\end{aligned}$$

On the other hand, the solution $\widehat{w}_{\mathrm{R}}$ to the ridge regression problem is:

$$\widehat{w}_{\mathrm{R}} = (X^\top X + \lambda)^{-1} X^\top y \tag{7}$$

with the singular value decomposition $X = U\Sigma V^\top$ we can simplify the solution $\widehat{w}_{\mathrm{R}}$:

$$\begin{aligned}
\widehat{w}_{\mathrm{R}} &= (X^\top X + \lambda)^{-1} X^\top y && (8)\\
&= ((U\Sigma V^\top)^\top (U\Sigma V^\top) + \lambda I)^{-1} (U\Sigma V^\top)^\top y && (9)\\
&= (V\Sigma^2 V^\top + \lambda V V^\top)^{-1} (V\Sigma U^\top) y && (10)\\
&= V(\Sigma^2 + \lambda)^{-1} V^\top V\Sigma U^\top y && (11)\\
&= V(\Sigma^2 + \lambda)^{-1} \Sigma U^\top y. && (12)
\end{aligned}$$

We can see that the difference between two solution is by a fraction of $\alpha = \frac{\Sigma^2}{\Sigma^2 + \lambda}$ where α is a value between 0 and 1 when $\lambda > 0$ stands. Therefore the solution of ridge regression problem "shrinks".

(b)

Let $X \in \mathbb{R}^{n \times n}$ be a matrix with singular values all equal to one. We have:

$$\begin{aligned}
XX^\top &= U\Sigma V^\top (U\Sigma V^\top)^\top && (13)\\
&= U\Sigma^2 U^\top && (14)\\
&= UU^\top && (15)\\
&= I && (16)
\end{aligned}$$

other way around we have:

$$\begin{aligned}
X^\top X &= (U\Sigma V^\top)^\top U\Sigma V^\top && (17)\\
&= V\Sigma^2 V^\top && (18)\\
&= VV^\top && (19)\\
&= I. && (20)
\end{aligned}$$

A.3

A.3.a

The eigen-values are $\lambda_1 = 5.11679$, $\lambda_2 = 3.74133$, $\lambda_{10} = 1.24273$, $\lambda_{30} = 0.36426$, and $\lambda_{50} = 0.16971$. And the sum of eigen-values $\sum_{i=1}^{d} \lambda_i$ is 52.72504.

A.3.b

Any example $x \in \mathbb{R}^d$ can be approximated as $\widehat{x}$ using just μ and the first k eigen-value/eigen-vectors pairs:

$$\widehat{x} \approx (x - \mu^T) V_k V_k^T + \mu, \tag{21}$$

where $V_k \in \mathbb{R}^{d \times k}$ is the matrix which the columns are formed by the first k eigen-vectors.

A.3.c

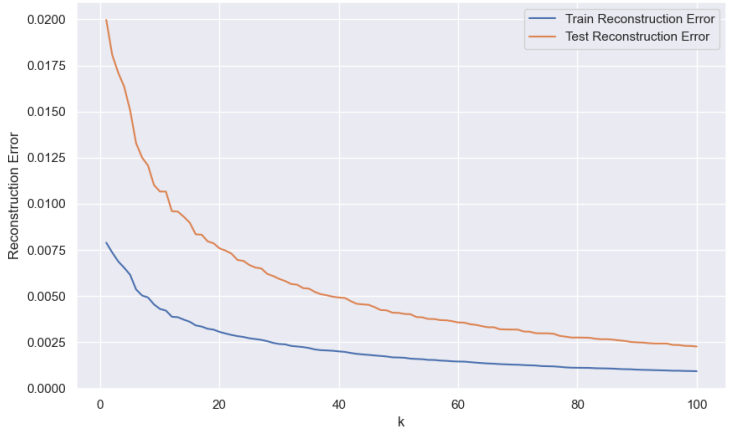

Figure 1: Plot of the training and testing reconstruction error (mean-squared error) from $k = 1$ to 100.

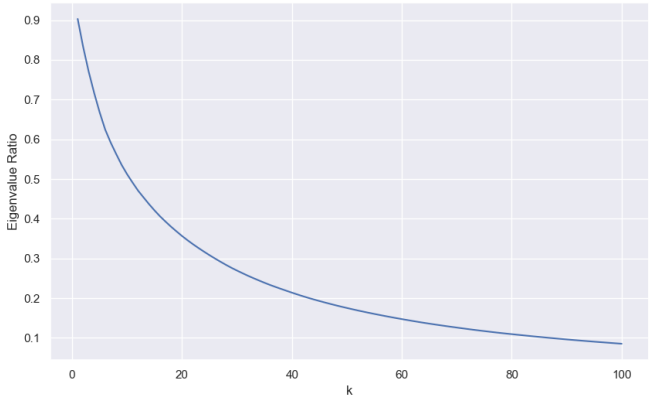

Figure 2: Plot of the ratio $1 - \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{d} \lambda_i}$ from $k = 1$ to 100.

A.3.d

From the visualization, I think the first few eigen-vectors are ought to capture where these different digits overlap (or not) on the 2D image space. For example, the largest corresponding eigen-vector was visualized into a zero-like image as it is trying to reconstruct the inner (or outer) portion of the digits. And the latter corresponding eigen-vectors look like some of the other digits like 9, 3 or 6 all following similar idea. Lastly, after some shared characteristic of the digit classes, the other less important eigen-vectors will probably be account for other minor difference like the styles or the sizes of the brush since this is a hand-written digit dataset after all.

Figure 3: Visualization of the first 10 eigen-vectors ($\mathbb{R}^{784}$) as image ($\mathbb{R}^{28\times28}$).

A.3.e

Using the first 5 or 15 eigen-vectors were not enough to recontrdcut the digits in the MNIST case, as you can see, the 2 in the second column looks like 0 while the 7 in the second column looks like 9. Then, using the first 40 eigen-vectors seems to be capable of reconstructing the original digit to a readable level. Furthermore, by using the first 100 eigen-vectors, we can observed that the blurry edges of the digits start to became clear.

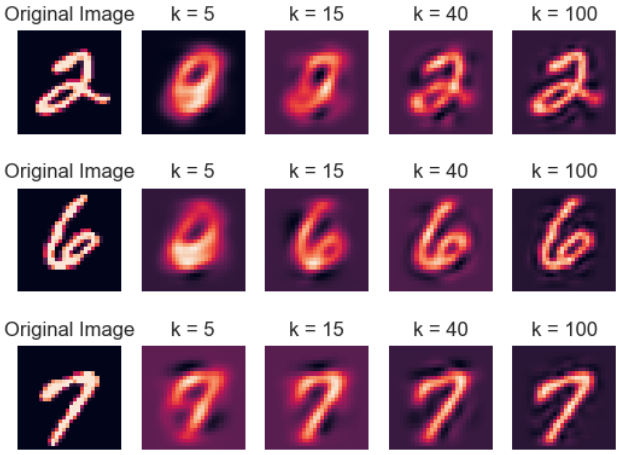

Figure 4: Visualization of a set of reconstructed digits from the training set for different values of $k = 5, 15, 40, 100$ with the original image.

A.3.code

```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
```

```
from mnist import MNIST

sns.set()

np.random.seed(1968990 + 20210531)


def load_dataset():
    mnist_data = MNIST('./mnist/')
    X_train, Y_train = map(np.array, mnist_data.load_training())
    X_test, Y_test = map(np.array, mnist_data.load_testing())
    X_train = X_train / 255.0
    X_test = X_test / 255.0
    return X_train, Y_train, X_test, Y_test


def main():
    X_train, Y_train, X_test, Y_test = load_dataset()
    n, d = X_train.shape
    m, d = X_test.shape
    # print(X_train.shape)
    # print(X_test.shape)
    I = np.ones((n, 1))

    mu = np.dot(X_train.T, I) / n
    sigma = np.dot((X_train - np.dot(I, mu.T)).T, (X_train - np.dot(I, mu.T))) /
        n
    print(mu.shape, sigma.shape)

    eigen_values, eigen_vectors = np.linalg.eigh(sigma)
    eigen_idx = np.argsort(eigen_values)[::-1]
    eigen_values_sorted, eigen_vectors_sorted = eigen_values[eigen_idx],
        eigen_vectors[:, eigen_idx]

    # A.3.a
    print(eigen_values_sorted[0], eigen_values_sorted[1], eigen_values_sorted
        [9], eigen_values_sorted[29], eigen_values_sorted[49])
    print('Summation of eigenvalues:', sum(eigen_values_sorted))

    # A.3.c
    current_eigen_values_sum = 0
    ks = []
    ratios = []
    train_losses = []
    test_losses = []

    def mse(Xhat, X):
        return np.linalg.norm(Xhat - X, ord=2) / len(X)

    for k in range(1, 101):
        X_train_reconstruct = np.dot(X_train - mu.T, np.dot(eigen_vectors_sorted
            [:, :k], eigen_vectors_sorted[:, :k].T)) + mu.reshape((784,))
        X_test_reconstruct = np.dot(X_test - mu.T, np.dot(eigen_vectors_sorted
            [:, :k], eigen_vectors_sorted[:, :k].T)) + mu.reshape((784,))
```

```
        train_losses.append(mse(X_train_reconstruct, X_train))
        test_losses.append(mse(X_test_reconstruct, X_test))
        ks.append(k)
        current_eigen_values_sum += eigen_values_sorted[k-1]
        ratios.append(1 - (current_eigen_values_sum/sum(eigen_values_sorted)))
        print(k, train_losses[-1], test_losses[-1])
    plt.figure(figsize=(10, 6))
    plt.plot(ks, train_losses, label='Train Reconstruction Error')
    plt.plot(ks, test_losses, label='Test Reconstruction Error')
    plt.xlabel('k')
    plt.ylabel('Reconstruction Error')
    plt.legend()
    plt.savefig('A3c_err.png')

    plt.figure(figsize=(10, 6))
    plt.plot(ks, ratios)
    plt.xlabel('k')
    plt.ylabel('Eigenvalue Ratio')
    plt.savefig('A3c_ratio.png')


    # A.3.d
    plt.figure(figsize=(10, 2))
    fig, axes = plt.subplots(1, 10)
    for i, ax in enumerate(axes.ravel()):
        ax.imshow(eigen_vectors_sorted[:, i].reshape((28, 28)))
        ax.set_title('k = {}'.format(i + 1))
        ax.axis('off')
    plt.savefig('A3d.png')

    # A.3.e
    plt.figure(figsize=(10, 6))
    fig, axes = plt.subplots(3, 5)
    # Y_train[5] => 2, Y_train[13] => 6, Y_train[15]
    plot_labels = [(2, 5), (6, 13), (7, 15)]
    plot_ks = [-1, 5, 15, 40, 100]
    for nrow, (label, index) in enumerate(plot_labels):
        for ncol, k in enumerate(plot_ks):
            if k == -1:
                axes[nrow, ncol].imshow(X_train[index].reshape(28, 28))
                axes[nrow, ncol].set_title('Original Image')
                axes[nrow, ncol].axis('off')
            else:
                reconstruct = (np.dot(X_train - mu.T, np.dot(
                    eigen_vectors_sorted[:, :k], eigen_vectors_sorted[:, :k].T))
                    + mu.reshape((784,)))[index]
                axes[nrow, ncol].imshow(reconstruct.reshape(28, 28))
                axes[nrow, ncol].set_title('k = {}'.format(k))
                axes[nrow, ncol].axis('off')
    plt.savefig('A3e.png')


if __name__ == '__main__':
    main()
```

A.4

A.4.a

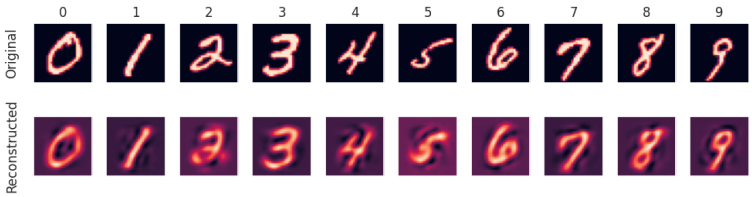

Figure 5: Reconstructed digits using auto-encoder with $h = 32$ (Training loss after 15 epochs is 0.0017541).

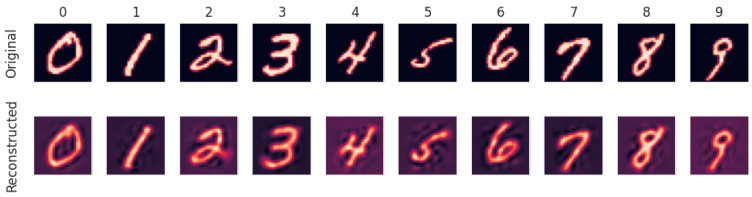

Figure 6: Reconstructed digits using auto-encoder with $h = 64$ (Training loss after 15 epochs is 0.009677).

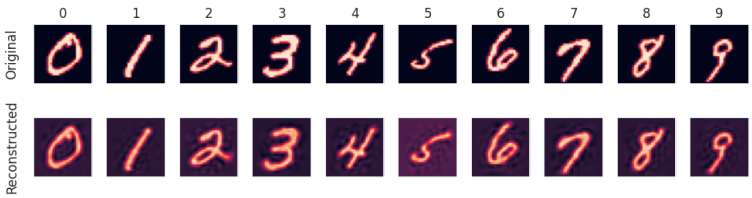

Figure 7: Reconstructed digits using auto-encoder with $h = 128$ (Training loss after 15 epochs is 0.004753).

A.4.b

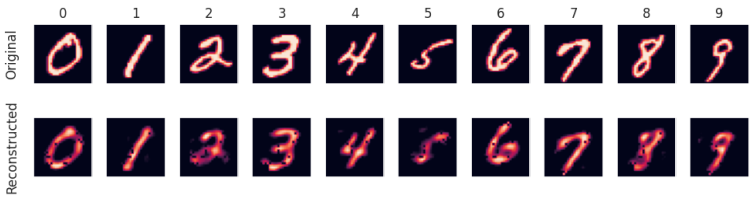

Figure 8: Reconstructed digits using auto-encoder with $h = 32$ (Training loss after 15 epochs is 0.020326).

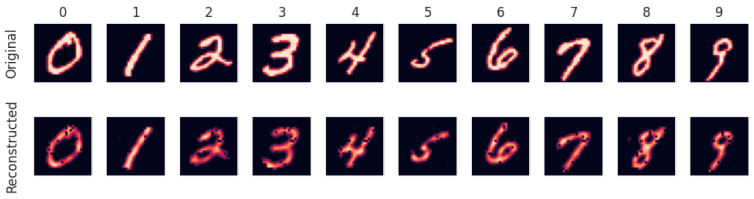

Figure 9: Reconstructed digits using auto-encoder with $h = 64$ (Training loss after 15 epochs is 0.013665).

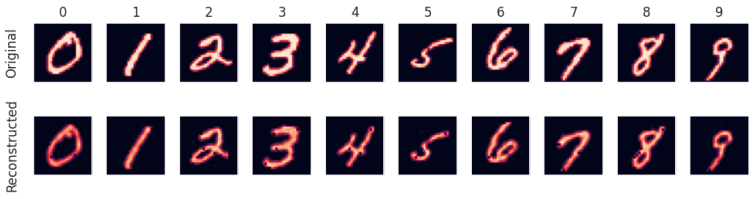

Figure 10: Reconstructed digits using auto-encoder with $h = 128$ (Training loss after 15 epochs is 0.011074).

A.4.c

The test loss of a single-layer network without non-linearity after 15 epochs is 0.004756 while the test loss of a single-layer network with non-linearity after 15 epochs is 0.010983.

A.4.d

Theoretically, the PCA and Auto-encoders without ReLU are essentially linear transformations but Auto-encoders with ReLU are capable of capturing complex non-linear functions. Statistically, higher the hidden dimension we set for the Auto-encoders or higher the k components chosen to reconstruct, lower the loss we will obtain from the training and test set. However, I will like to point out one finding, even though the non-linear Auto-encoder had higher test loss, the reconstructed digits seem much better in comparison to the linear Auto-encoder ones.

A.4.code

```
import random
import time

import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision.datasets as datasets
from mnist import MNIST
from torchvision import transforms
from tqdm import tqdm

sns.set()
```

```
plt.rcParams["axes.grid"] = False

torch.manual_seed(1968990 + 20210604)
np.random.seed(1968990 + 20210604)
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')


class AutoEncoder(nn.Module):
    def __init__(self, input_size=784, hidden_size=64):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.encoder = nn.Sequential(
            nn.Linear(self.input_size, self.hidden_size)
        )
        self.decoder = nn.Sequential(
            nn.Linear(self.hidden_size, self.input_size)
        )

    def forward(self, x):
        z = self.encoder(x)
        r = self.decoder(z)
        return r


class AutoEncoder_relu(nn.Module):
    def __init__(self, input_size=784, hidden_size=64):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.encoder = nn.Sequential(
            nn.Linear(self.input_size, self.hidden_size),
            nn.ReLU()
        )
        self.decoder = nn.Sequential(
            nn.Linear(self.hidden_size, self.input_size),
            nn.ReLU()
        )

    def forward(self, x):
        z = self.encoder(x)
        r = self.decoder(z)
        return r


def load_dataset():
    mnist_trainset = datasets.MNIST(root='./data/', train=True, download=True,
        transform=transforms.ToTensor())
    mnist_testset = datasets.MNIST(root='./data/', train=False, download=True,
        transform=transforms.ToTensor())
    train_loader = torch.utils.data.DataLoader(mnist_trainset, batch_size=16,
        shuffle=True)
    test_loader = torch.utils.data.DataLoader(mnist_testset, batch_size=16,
        shuffle=False)
```

```
    mnist_visset = datasets.MNIST(root='./data/', train=False, download=True,
        transform=transforms.ToTensor())
    current_digit = 0
    mnist_visset.data = []
    mnist_visset.targets = []
    for data, label in zip(mnist_trainset.data, mnist_trainset.targets):
        if label == current_digit:
            mnist_visset.data.append(data)
            mnist_visset.targets.append(label)
            current_digit += 1
        if label >= 10:
            break

    vis_loader = torch.utils.data.DataLoader(mnist_visset, batch_size=10,
        shuffle=False)

    return train_loader, test_loader, vis_loader


def train(epochs, data_loader, model, optimizer, criterion):
    for epoch in range(epochs):
        loss = 0
        for i, (data, _) in enumerate(data_loader):
            data = data.view(-1, 28*28).to(DEVICE)
            optimizer.zero_grad()

            outputs = model(data)

            train_loss = criterion(outputs, data)
            train_loss.backward()

            optimizer.step()

            loss += train_loss.item()

        loss /= len(data_loader)
        if epoch + 1 == epochs:
            print("Epoch : {}/{}, Train loss = {:.6f}".format(epoch + 1, epochs,
                loss))


def test(data_loader, model, criterion=nn.MSELoss()):
    loss = 0
    with torch.no_grad():
        for i, (data, _) in enumerate(data_loader):

            data = data.view(-1, 28*28).to(DEVICE)
            outputs = model(data)
            test_loss = criterion(outputs, data)
            loss += test_loss.item()

        loss /= len(data_loader)
        print("Test loss = {:.6f}".format(Aloss))
```

```
def plot_original_and_reconstruction(data_loader, model, save_path):

    for data, labels in data_loader:
        data = data.view(-1, 28*28).to(DEVICE)
        with torch.no_grad():
            original = data
            reconstruct = model(data)

    fig, axes = plt.subplots(nrows=2, ncols=10, sharex=True, sharey=True,
        figsize=(10, 3))
    for i, row in enumerate(axes):
        for j, cell in enumerate(row):
            if i == 0:
                cell.imshow(original.cpu().data[j, :].reshape((28, 28)))
                cell.set_title(j)
                if j == 0:
                    cell.set_ylabel('Original')
                cell.set(xticklabels=[])
                cell.set(yticklabels=[])
            else:
                cell.imshow(reconstruct.cpu().data[j, :].reshape((28, 28)))
                if j == 0:
                    cell.set_ylabel('Reconstructed')
    plt.tight_layout()
    plt.savefig(save_path)


def main_A4a(ks=[32, 64, 128]):

    train_loader, test_loader, vis_loader = load_dataset()

    for k in ks:
        model = AutoEncoder(hidden_size=k).to(DEVICE)
        optimizer = optim.Adam(model.parameters(), lr=1e-3)
        criterion = nn.MSELoss()

        train(15, train_loader, model, optimizer, criterion)

        plot_original_and_reconstruction(vis_loader, model, 'A4a_{}.png'.format(
            k))

        test(test_loader, model, criterion)

    return model


def main_A4b(ks=[32, 64, 128]):

    train_loader, test_loader, vis_loader = load_dataset()

    for k in ks:
        model = AutoEncoder_relu(hidden_size=k).to(DEVICE)
        optimizer = optim.Adam(model.parameters(), lr=1e-3)
        criterion = nn.MSELoss()
```

```
        train(15, train_loader, model, optimizer, criterion)

        plot_original_and_reconstruction(vis_loader, model, 'A4b_{}.png'.format(
            k))

        test(test_loader, model, criterion)

    return model


if __name__ == '__main__':
    model_a = main_A4a()
    model_b = main_A4b()
```

A.5

A.5.a

```
class LloydsAlgorithm:
    def __init__(self, k=2, max_iter=100):
        self.k = k
        self.max_iter = max_iter

        self.x_data, _, self.x_test, _ = load_dataset()
        self.n, self.d = self.x_data.shape
        self.current_iter = 0
        # self.distances = [float('inf')]
        # initial clustering
        self.clusters = { tuple(centroid): [] for centroid in self.x_data[np.
            random.choice(np.arange(self.n), self.k)]}
        self.test_clusters = {}
        self.clustering()
        self.distances = [k_means_obj(self.clusters)]
        self.centering()

    def clustering(self):
        centroids = [np.asarray(tuple_centroid) for tuple_centroid in self.
            clusters.keys()]
        for centroid in self.clusters.keys():
            self.clusters[centroid] = []
        for point in self.x_data:
            closest_centroid = centroids[np.argmin([np.linalg.norm(point -
                centroid) for centroid in centroids])]
            if tuple(closest_centroid) not in self.clusters:
                raise RuntimeError
            self.clusters[tuple(closest_centroid)].append(point)

    def centering(self):
        new_clusters = {}
        for centroid, points in self.clusters.items():
            new_centroid = np.mean(points, axis=0)
            new_clusters[tuple(new_centroid)] = points
        self.clusters = new_clusters

    def test_clustering(self):
        centroids = [np.asarray(tuple_centroid) for tuple_centroid in self.
            clusters.keys()]
        for centroid in self.clusters.keys():
            self.test_clusters[centroid] = []
        for point in self.x_test:
            closest_centroid = centroids[np.argmin([np.linalg.norm(point -
                centroid) for centroid in centroids])]
            if tuple(closest_centroid) not in self.test_clusters:
                raise RuntimeError
            self.test_clusters[tuple(closest_centroid)].append(point)

    def iter(self, threshold=1e-4):
        while self.current_iter <= self.max_iter:
```

```
        self.current_iter += 1
        start_time = time.time()
        old_dist = self.distances[-1]
        old_clusters = self.clusters

        self.clustering()
        self.distances.append(k_means_obj(self.clusters))
        self.centering()

        new_dist = self.distances[-1]
        # if (old_dist - new_dist) < threshold or (old_dist - new_dist) > (
            self.distances[-2] - old_dist):
        if old_dist < new_dist:
            self.clusters = old_clusters
            self.current_iter -= 1
            print('(k={}) Iter {}: {} (in {}s)'.format(self.k, self.
                current_iter, old_dist - new_dist, time.time() - start_time))
            break
        if (old_dist - new_dist) < threshold:
            print('(k={}) Iter {}: {} (in {}s)'.format(self.k, self.
                current_iter, old_dist - new_dist, time.time() - start_time))
            break
        else:
            print('(k={}) Iter {}: {} (in {}s)'.format(self.k, self.
                current_iter, old_dist - new_dist, time.time() - start_time))
```

A.5.b

Run the algorithm on the training dataset of MNIST with k = 10, plotting the objective function (1) as a function of the iteration number. Visualize (and include in your report) the cluster centers as a 28 × 28 image.

Figure 11: Visualization the cluster centers on the training dataset of MNIST with $k = 10$.

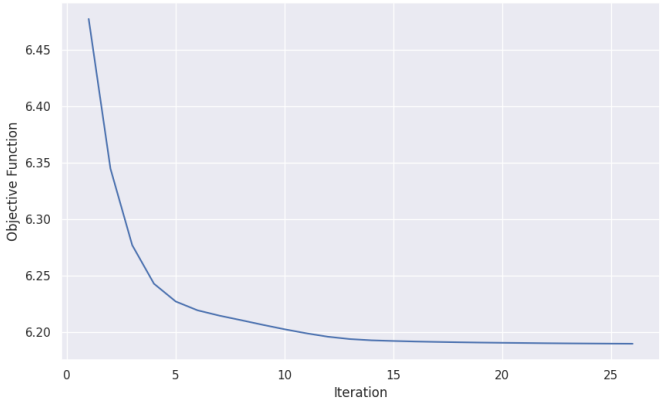

Figure 12: Plot of objective function versus iteration number on the training dataset of MNIST with $k = 10$.

A.5.c

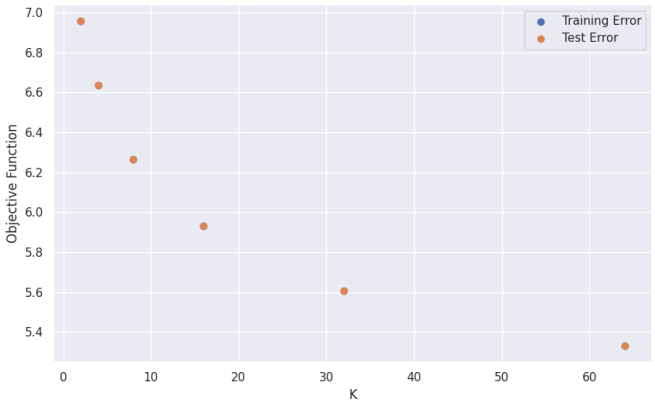

Figure 13: Plot of errors the training and testing dataset of MNIST with different k.

A.5.code

```
import time

import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

from mnist import MNIST

sns.set()

np.random.seed(1968990 + 20210601)


def load_dataset():
    mnist_data = MNIST('./mnist/')
    X_train, Y_train = map(np.array, mnist_data.load_training())
    X_test, Y_test = map(np.array, mnist_data.load_testing())
    X_train = X_train / 255.0
    X_test = X_test / 255.0
    return X_train, Y_train, X_test, Y_test


def k_means_obj(clusters):
    dist = 0
    cnt = 0
    for centroid, points in clusters.items():
        for point in points:
            dist += np.linalg.norm(centroid - point)
            cnt += 1
    return dist / cnt
```

```
class LloydsAlgorithm:
    def __init__(self, k=2, max_iter=100):
        self.k = k
        self.max_iter = max_iter

        self.x_data, _, self.x_test, _ = load_dataset()
        self.n, self.d = self.x_data.shape
        self.current_iter = 0
        # self.distances = [float('inf')]
        # initial clustering
        self.clusters = { tuple(centroid): [] for centroid in self.x_data[np.
            random.choice(np.arange(self.n), self.k)]}
        self.test_clusters = {}
        self.clustering()
        self.distances = [k_means_obj(self.clusters)]
        self.centering()

    def clustering(self):
        centroids = [np.asarray(tuple_centroid) for tuple_centroid in self.
            clusters.keys()]
        for centroid in self.clusters.keys():
            self.clusters[centroid] = []
        for point in self.x_data:
            closest_centroid = centroids[np.argmin([np.linalg.norm(point -
                centroid) for centroid in centroids])]
            if tuple(closest_centroid) not in self.clusters:
                raise RuntimeError
            self.clusters[tuple(closest_centroid)].append(point)

    def centering(self):
        new_clusters = {}
        for centroid, points in self.clusters.items():
            new_centroid = np.mean(points, axis=0)
            new_clusters[tuple(new_centroid)] = points
        self.clusters = new_clusters

    def test_clustering(self):
        centroids = [np.asarray(tuple_centroid) for tuple_centroid in self.
            clusters.keys()]
        for centroid in self.clusters.keys():
            self.test_clusters[centroid] = []
        for point in self.x_test:
            closest_centroid = centroids[np.argmin([np.linalg.norm(point -
                centroid) for centroid in centroids])]
            if tuple(closest_centroid) not in self.test_clusters:
                raise RuntimeError
            self.test_clusters[tuple(closest_centroid)].append(point)

    def iter(self, threshold=1e-4):
        while self.current_iter <= self.max_iter:
            self.current_iter += 1
            start_time = time.time()
            old_dist = self.distances[-1]
            old_clusters = self.clusters
```

```
            self.clustering()
            self.distances.append(k_means_obj(self.clusters))
            self.centering()

            new_dist = self.distances[-1]
            # if (old_dist - new_dist) < threshold or (old_dist - new_dist) > (
                self.distances[-2] - old_dist):
            if old_dist < new_dist:
                self.clusters = old_clusters
                self.current_iter -= 1
                print('(k={}) Iter {}: {} (in {}s)'.format(self.k, self.
                    current_iter, old_dist - new_dist, time.time() - start_time))
                break
            if (old_dist - new_dist) < threshold:
                print('(k={}) Iter {}: {} (in {}s)'.format(self.k, self.
                    current_iter, old_dist - new_dist, time.time() - start_time))
                break
            else:
                print('(k={}) Iter {}: {} (in {}s)'.format(self.k, self.
                    current_iter, old_dist - new_dist, time.time() - start_time))


def main():
    kmeans = LloydsAlgorithm(k=10, max_iter=10A0)
    kmeans.iter()

    plt.figure(figsize=(10, 6))
    plt.plot(np.arange(1, kmeans.current_iter + 1), kmeans.distances[1:])
    plt.xlabel('Iteration')
    plt.ylabel('Objective Function')
    plt.savefig('A5b_obj.png')

    plt.figure(figsize=(10, 2))
    fig, axes = plt.subplots(1, 10)
    for i, (ax, centroid) in enumerate(zip(axes.ravel(), kmeans.clusters.keys())
        ):
        ax.imshow(np.asarray(centroid).reshape((28, 28)))
        ax.axis('off')
    plt.savefig('A5b_centroid.png')
    plt.tight_layout()


def A5c():
    ks = [2, 4, 8, 16, 32, 64]
    train_objs = []
    test_objs = []
    for k in ks:
        kmeans = LloydsAlgorithm(k=k, max_iter=100)
        kmeans.iter()
        kmeans.clustering()
        kmeans.test_clustering()

        train_obj = k_means_obj(kmeans.clusters)
        test_obj = k_means_obj(kmeans.test_clusters)
```

```
        print('Training Objective: {} / Test Objective: {}'.format(train_obj,
            test_obj))

        train_objs.append(train_obj)
        test_objs.append(test_obj)

    plt.figure(figsize=(10, 6))
    plt.scatter(ks, train_objs, label='Training Error')
    plt.scatter(ks, train_objs, label='Test Error')
    plt.legend()
    plt.xlabel('K')
    plt.ylabel('Objective Function')
    plt.savefig('A5c.png')


if __name__ == '__main__':
    main()
    A5c()
```

A.6

A.6.a

(Disease Susceptibility Predictor)

- Dataset Pipeline: The categorical features including race, ethnicity, gender and many others can be one-hot encoded as other numerical features including age, income, height/weight and many others can be use directly or normalized in a proper manner. The labels is whether a person has the disease or not so it will be formulate into a binary classification problem.
- Machine Learning Pipeline: As our goal is to determine how susceptible someone is to this specific disease when they enter in personal information, and those data are expected to be incomplete. Therefore the machine learning framework I chose here will be simple Neural Networks as it will probably handling the missing input features more robust in comparison to decision tree. The input we use will be the one-hot encoded and normalized features with a certain mechanism to filled up the missing values. The output will be a scalar representing the probability of this certain person to the specific disease.

A.6.b

(Social Media App Facial Recognition Technology)

- Dataset Pipeline: In order to map someone's face for the application of filters, we need the provided face dataset to be labeled with some 2d-keypoints as the can be important face landmark we are interested in using in the filters. Manually labeling those 2d-keypoints of the collected face images could cost lots of effort as it may requires around 40 2d-keypoints for each face. However, we could use some pre-trained model like OpenPose developed by CMU to inference the 2d-keypoints and make slight modification base on the generated 2d-keypoints to save time.
- Machine Learning Pipeline: As our goal is to quickly identify the key features of a person's face, we can continue to use the OpenPose framework to train our own model base on the 2d-keypoints we defined. The input will be any image with a human face and the output will be a vector containing the detected 2d-keypoints locations and confidence scores. Then the filters can use these 2d-keypoints locations to render the desired effect.

A.6.c

(Malware Detection)

- Dataset Pipeline: The categorical features including file extension can be one-hot encoded as other numerical features including file size, created time and modified time and many others can be use directly or normalized in a proper manner. Furthermore,some data like file publisher or file name may have too much variation so it is not practical to use one-hot encoding on them, for these data we will try to use an pre-trained word embedding to generate word vectors for them. The labels is fairly simple in this case, it is a binary label whether the file is a malware or not.
- Machine Learning Pipeline: As our goal is to detect whether incoming files are malicious for users, we can again use a simple Neural Networks to do the job. However, considering the fact that the same malware will probably being seen by different users, it is in best interest for us to construct a database of confirmed malware and to do the matching before shoving the metatdata into the model. The input we use will be the one-hot encoded categorical features, normalized numerical features and some word-embedding features concatenate together. The output will be a scalar representing the probability of this certain file being a malware or not.

B.1

Collaborator: Zhongyu Jiang

B.1.a

The error $\mathcal{E}_{\text{test}}(\widehat{R}) = 1.063564$ as the estimate $\widehat{R}$ is simply constructed by the average rating of the users that rated the i-th movie which $\widehat{R}_{i,j} = \mu_i \ \forall j$.

B.1.b

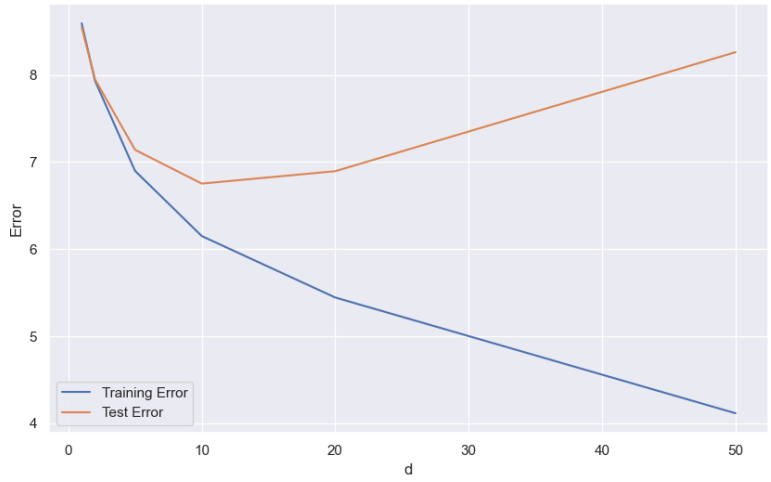

Figure 14: Plot of MSE on training and test set vs. d using best rank-d approximation.

B.1.c

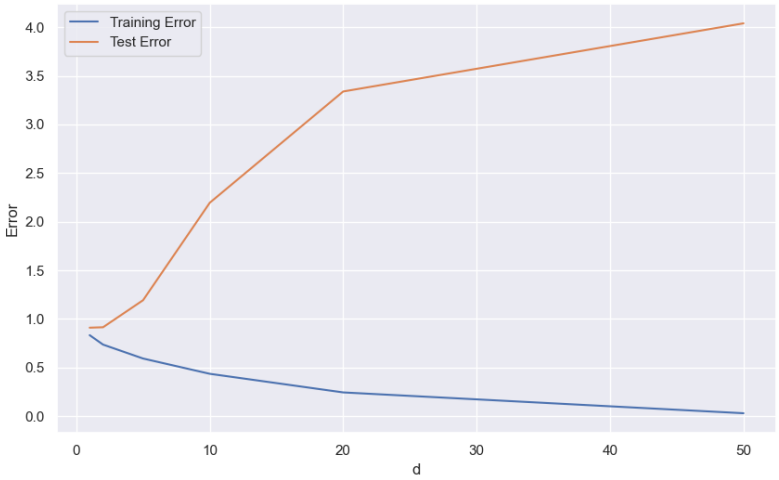

Figure 15: Plot of MSE on training and test set vs. d using alternating minimization.

B.1.d

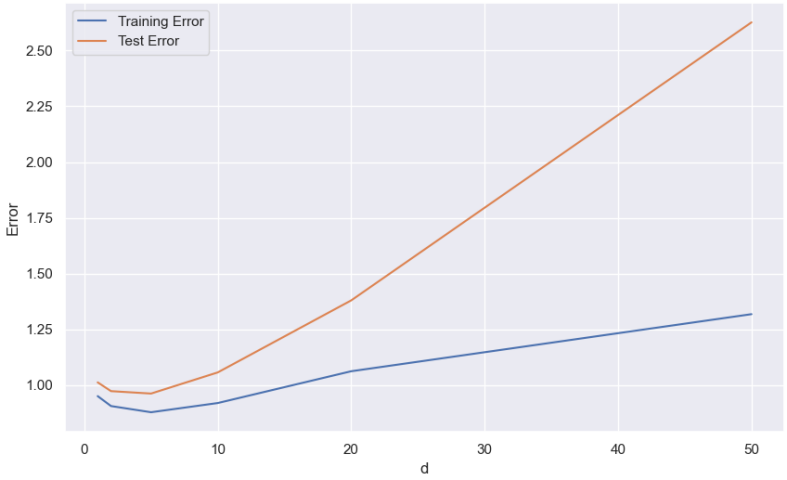

Figure 16: Plot of MSE on training and test set vs. d using batched SGD (batch size=100).

B.1.e

We implement *Alternating minimization* in B.1.c and *Batched SGD* in B.1.d using the identical loss function:

$$L\Big(\{u_i\}_{i=1}^m, \{v_j\}_{j=1}^n\Big) := \sum_{(i,j,R_{i,j})\in \text{train}} (\langle u_i, v_j\rangle - R_{i,j})^2 + \lambda \sum_{i=1}^m \|u_i\|_2^2 + \lambda \sum_{j=1}^n \|v_j\|_2^2. \tag{22}$$

However, we can observe a very large difference in term of how the models eventually fit. For example, the batched SGD will take a great amount of effort as we need to carefully tune the hyper-parameters but better results can also be expected in comparison to alternating minimization. The alternating minimization take the entire training data and calculate the closed form solution at once and thus converge in a much faster manner in comparison to batched SGD.

B.1.f (extra)

The Funk MF algorithm in B.1.c with the parameters ($d = 2$, $\sigma = 1.5$, $\lambda = 0.01$) achieved a 0.898487 test error after 20 iterations of updates.

B.1.code

```
#%%

import scipy
from scipy.sparse.linalg import svds
import matplotlib.pyplot as plt
import csv
import numpy as np
import collections
import seaborn as sns
```

```
sns.set()

#%%

data = []
with open('ml-100k/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 error(r_hat, r):
    return np.mean((r_hat[r[:, 0], r[:, 1]] - r[:, 2]) ** 2)

#%%

movies_scores = collections.defaultdict(list)
mu = np.array([0.0 for _ in range(num_items)])

for u, m, score in train:
    movies_scores[m].append(score)

for m, scores in movies_scores.items():
    mu[m] = np.mean(scores)

r_hat_mu = np.zeros((num_users, num_items))
r_hat_mu[train[:, 0], train[:, 1]] = train[:, 2]

for m, score in enumerate(mu):
    for j in range(num_users):
        if r_hat_mu[j, m] == 0:
            r_hat_mu[j, m] = mu[m]

err = error(r_hat_mu, test)

print(err)

#%%

# B.1.b
ds = [1, 2, 5, 10, 20, 50]

r_hat = np.zeros((num_users, num_items))
```

```
r_hat[train[:, 0], train[:, 1]] = train[:, 2]

train_errors = []
test_errors = []

for d in ds:
    U, S, V = scipy.sparse.linalg.svds(r_hat, d)
    r_hat_rank_d = np.matmul(np.matmul(U, np.diag(S)), V)

    train_errors.append(error(r_hat_rank_d, train))
    test_errors.append(error(r_hat_rank_d, test))

plt.figure(figsize=(10, 6))
plt.plot(ds, train_errors, label='Training Error')
plt.plot(ds, test_errors, label='Test Error')
plt.legend()
plt.xlabel('d')
plt.ylabel('Error')
plt.savefig('B1b.png')
plt.show()

#%%

# B.1.c.new
# Alernating Minimization refrerence from https://math.stackexchange.com/
    questions/1072451/analytic-solution-for-matrix-factorization-using-
    alternating-least-squares
def alternating_minimization_U(d, r_hat, fixed_vector, lambda_reg=0.01):
    A = fixed_vector.T.dot(fixed_vector) + np.eye(d) * lambda_reg
    return r_hat.dot(fixed_vector).dot(np.linalg.inv(A))

def alternating_minimization_V(d, r_hat, fixed_vector, lambda_reg=0.01):
    A = fixed_vector.T.dot(fixed_vector) + np.eye(d) * lambda_reg
    return np.linalg.inv(A).dot(fixed_vector.T.dot(r_hat))

iters = 20
ds = [1, 2, 5, 10, 20, 50]
sigmas = [1.5]
lambda_regs = [0.01]
train_errors = []
test_errors = []
r_hat = np.zeros((num_users, num_items))
r_hat[train[:, 0], train[:, 1]] = train[:, 2]


for d in ds:
    for sigma in sigmas:
        for lambda_reg in lambda_regs:
            U = sigma * np.random.random((num_users, d))
            V = sigma * np.random.random((num_items, d))

            for _ in range(iters):
                U_ = np.zeros((num_users, d))
                V_ = np.zeros((num_items, d))
```

```
                for i in range(num_users):
                    U_[i] = alternating_minimization_U(
                        d,
                        train[train[:, 0] == i, 2],
                        V[train[train[:, 0] == i, 1]]
                    )

                for j in range(num_items):
                    V_[j] = alternating_minimization_V(
                        d, train[train[:, 1] == j, 2],
                        U_[train[train[:, 1] == j, 0]]
                    )

                r_hat_pred = U_.dot(V_.T)
                train_error = error(r_hat_pred, train)
                test_error = error(r_hat_pred, test)
                # print('sigma={} lambda={} train={} test={}'.format(sigma,
                    lambda_reg, train_error, test_error))

                U = U_
                V = V_

            print('d={} sigma={} lambda={} train={} test={}'.format(d, sigma,
                lambda_reg, train_error, test_error))

            train_errors.append(error(r_hat_pred, train))
            test_errors.append(error(r_hat_pred, test))

# print(train_errors, test_errors)

plt.figure(figsize=(10, 6))
plt.plot(ds, train_errors, label='Training Error')
plt.plot(ds, test_errors, label='Test Error')
plt.legend()
plt.xlabel('d')
plt.ylabel('Error')
plt.savefig('B1c.png')
plt.show()

#%%

# B.1.d.new

iters = 320
ds = [1, 2, 5, 10, 20, 50]
sigmas = [1.5]
lambda_regs = [0.01]
train_errors = []
test_errors = []
r_hat = np.zeros((num_users, num_items))
r_hat[train[:, 0], train[:, 1]] = train[:, 2]

batch_size = 1000
learning_rate = 5e-3
```

```
for d in ds:
    for sigma in sigmas:
        for lambda_reg in lambda_regs:
            U = sigma * np.random.random((num_users, d))
            V = sigma * np.random.random((num_items, d))
            cnt = 0

            for _ in range(iters+1):
                U_ = np.zeros((num_users, d))
                V_ = np.zeros((num_items, d))

                # epoch
                if cnt > len(train):
                    cnt = 0
                    np.random.shuffle(train)
                    r_hat_pred = U.dot(V.T)
                    train_error = error(r_hat_pred, train)
                    test_error = error(r_hat_pred, test)
                    print('sigma={} lambda={} train={} test={}'.format(sigma,
                        lambda_reg, train_error, test_error))
                # batch
                train_batch = train[cnt:cnt + batch_size]
                cnt += batch_size

                for i in range(num_users):
                    if np.sum(train_batch[train_batch[:, 0] == i, 2]) != 0:
                        fixed_vector = V[train_batch[train_batch[:, 0] == i, 1]]
                        U_[i] = U[i].dot(fixed_vector.T.dot(fixed_vector) + np.
                            eye(d) * lambda_reg) - train_batch[train_batch[:, 0]
                            == i, 2].dot(fixed_vector)

                for j in range(num_items):
                    if np.sum(train_batch[train_batch[:, 1] == j, 2]) != 0:
                        fixed_vector = U[train_batch[train_batch[:, 1] == j, 0]]
                        V_[j] = V[j].dot(fixed_vector.T.dot(fixed_vector) + np.
                            eye(d) * lambda_reg) - train_batch[train_batch[:, 1]
                            == j, 2].dot(fixed_vector)

                U -= learning_rate * U_
                V -= learning_rate * V_
                # print('sigma={} lambda={} train={} test={}'.format(sigma,
                    lambda_reg, train_error, test_error))

            r_hat_pred = U.dot(V.T)
            train_error = error(r_hat_pred, train)
            test_error = error(r_hat_pred, test)

            print('d={} sigma={} lambda={} train={} test={}'.format(d, sigma,
                lambda_reg, train_error, test_error))

            train_errors.append(error(r_hat_pred, train))
            test_errors.append(error(r_hat_pred, test))


plt.figure(figsize=(10, 6))
```

```
plt.plot(ds, train_errors, label='Training Error')
plt.plot(ds, test_errors, label='Test Error')
plt.legend()
plt.xlabel('d')
plt.ylabel('Error')
plt.savefig('B1d.png')
plt.show()
```