Cem Daloğlu
21602161 / 29.02.2021

EEE 361 Homework-1 Report

Part 1.1

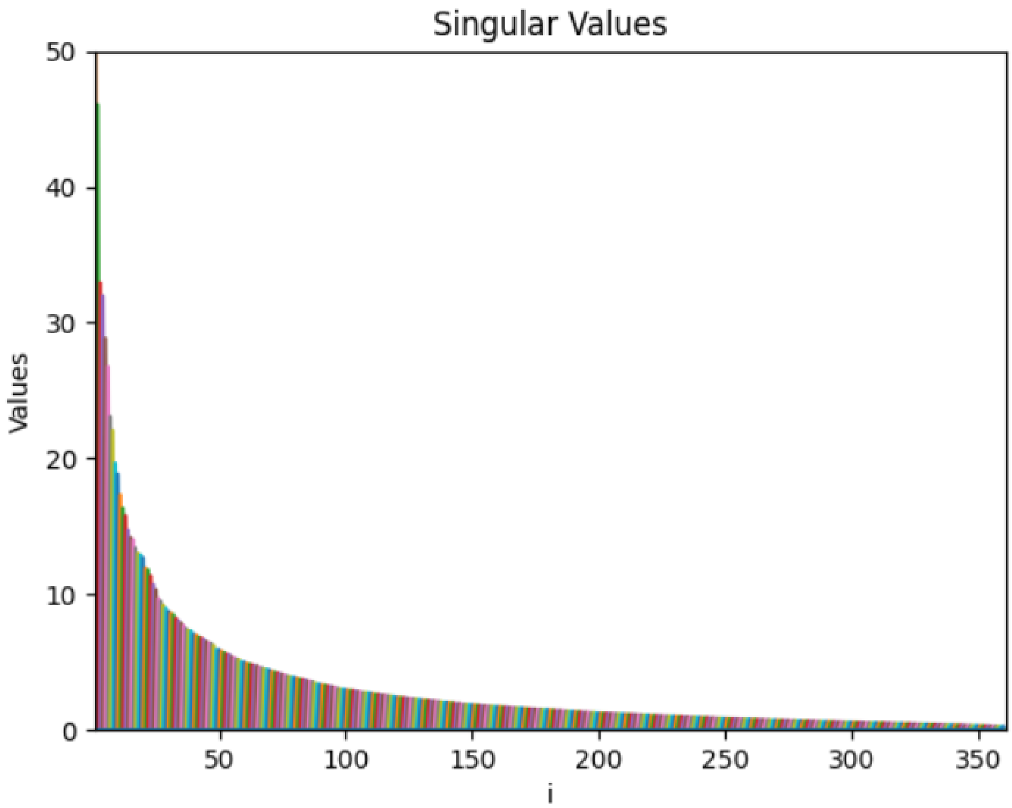

Figure 1: Singular values

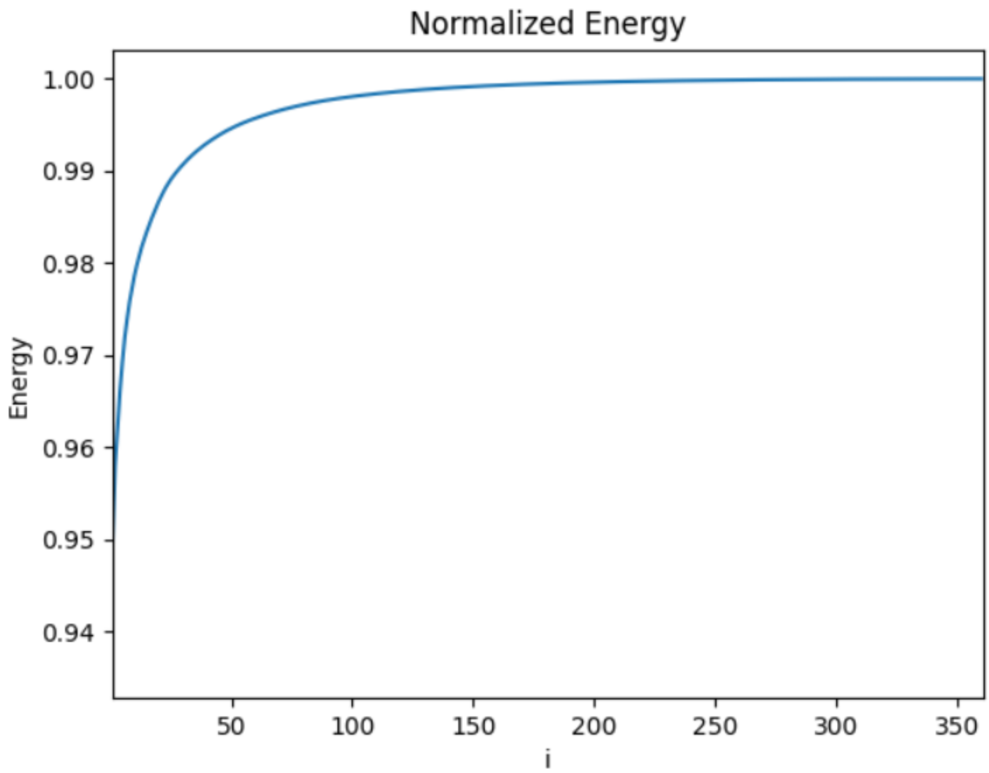

Figure 2: Normalized accumulated energy

Defined indices gave the corresponding energy rates as the following;

$$I_{90} = 0, I_{95} = 2, I_{99} = 28.$$

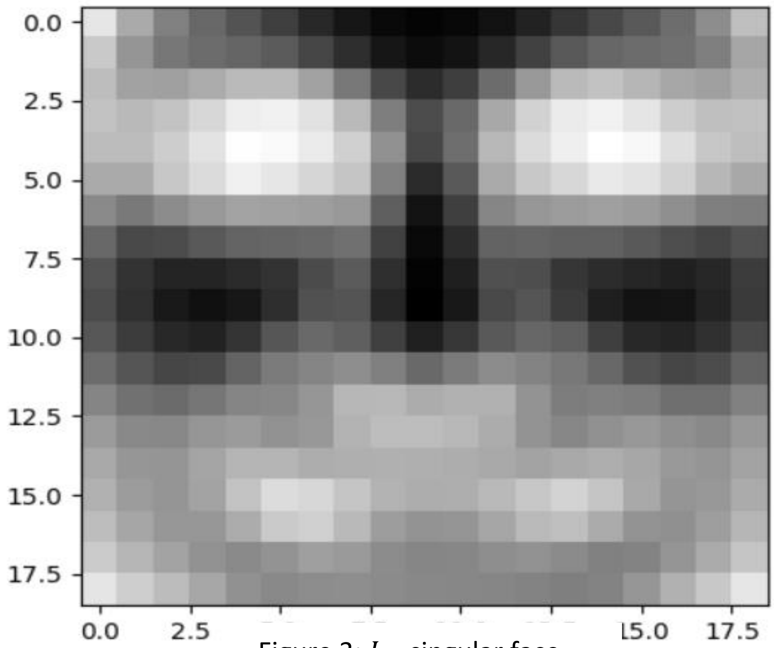

Figure 3: I_{90} singular face

Since $I_{90} = 0$, only one singular face is displayed. Singular face does not have any localized but have distributed features, it only has eye, mouth and face shape features. No local feature is obtained from this singular face. There are non-negative values in the plotted singular face.

Part 1.2

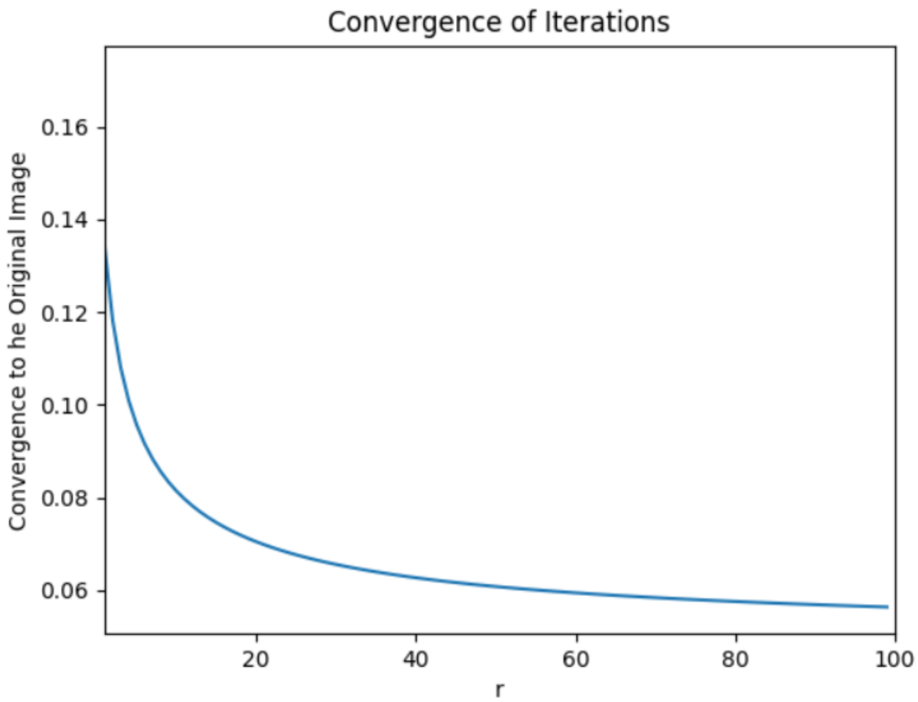

Figure 4: Convergence of Iterations

Obtained convergence of iteration plot in Figure 4 is similar to the Figure 3 in the first reference for HALS update label. Since I normalized the input matrix X, obtained plot begins from a lower point than the one in the reference. Normalizing input matrix lessens the processing time and does not change anything in the input matrix.

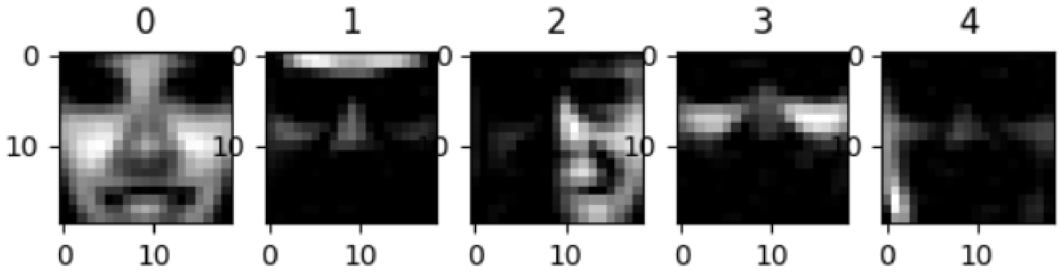

Figure 5: First 5 eigenfaces

From Figure 5, the first eigenface does not have localized features, it has general nose, eyes, mouth and face shape features. However, second eigenface contains localized features such as front head of the person, third eigenface contains half of the person face so it is localized as well. Last two eigenface contains localized features as cheeks of the person. Therefore, NMF gave localized features not distributed features. Eigenfaces have only non-negative values as described in the first reference.

Part 2

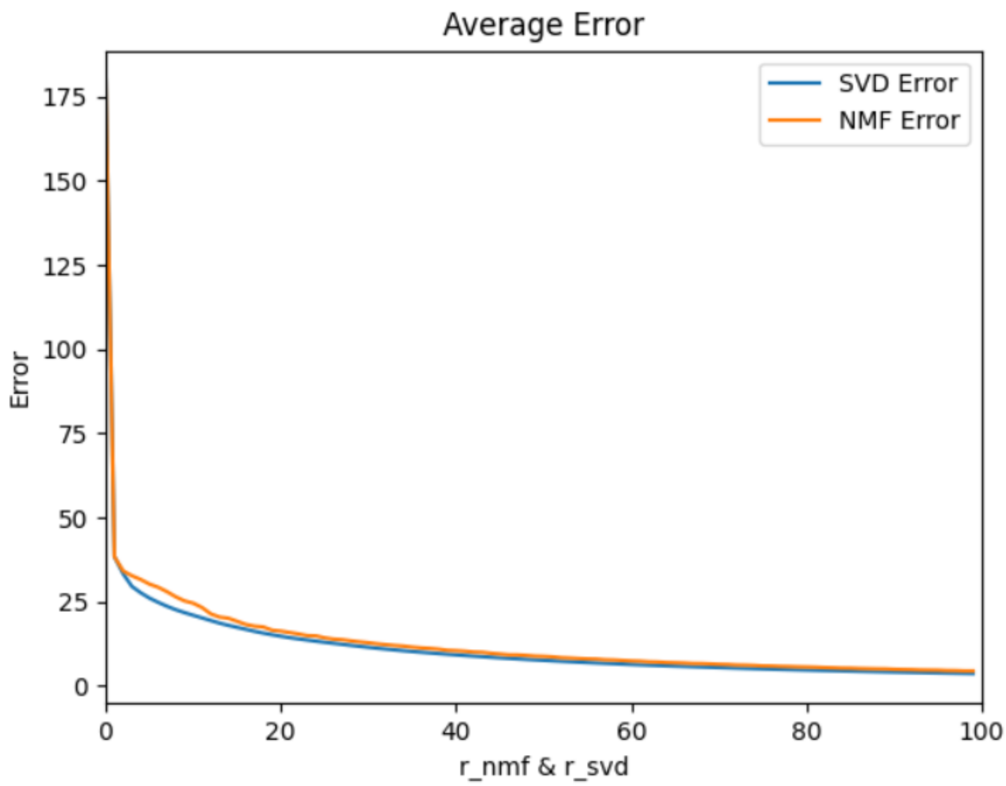

Figure 6: Average of the errors over the test images versus different r_{svd}, r_{nmf} values for noise rate $\eta = 1$.

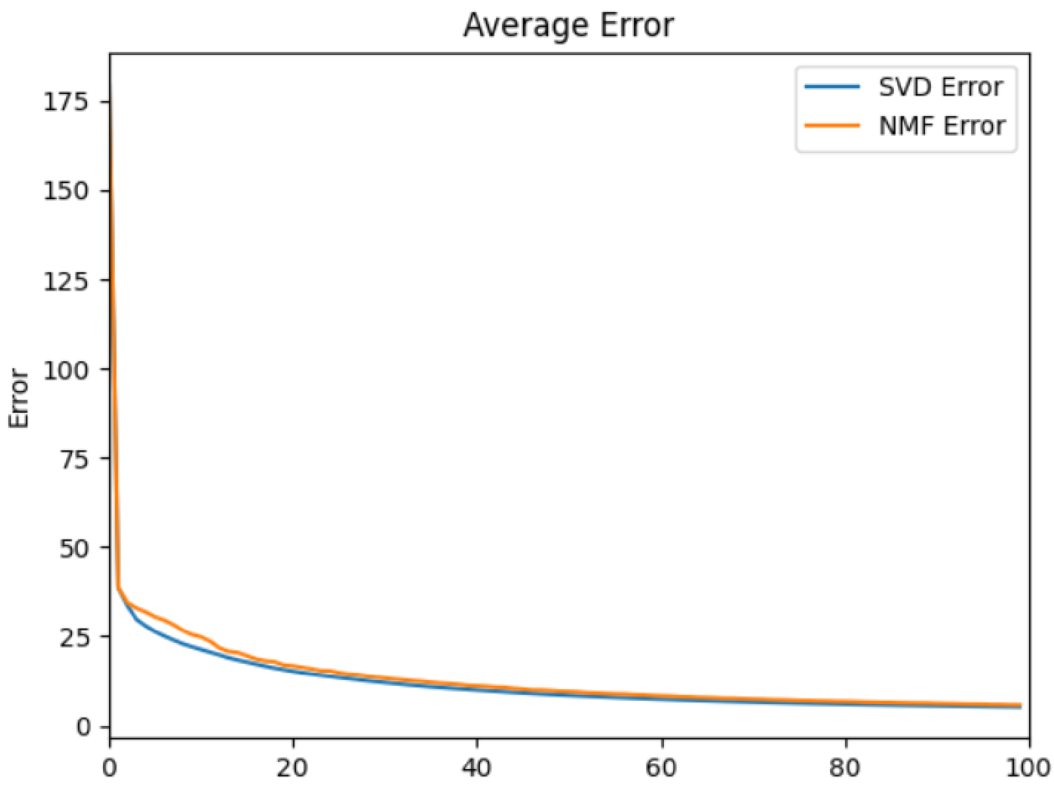

Figure 7: Average of the errors over the test images versus different r_{svd} , r_{nmf} values for noise rate $\eta = 10$.

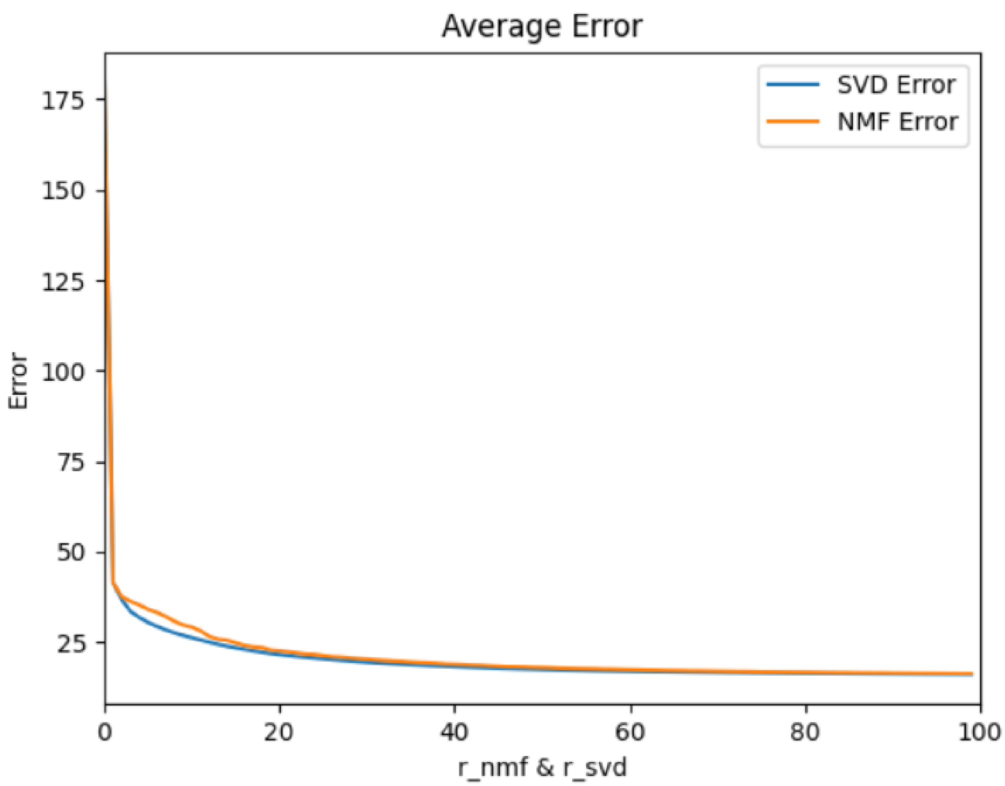

Figure 8: Average of the errors over the test images versus different r_{svd}, r_{nmf} values for noise rate $\eta = 25$.

By looking at Figure 6, 7 and 8, it could be said that SVD is a better approach than NMF since blue curves are more likely to converge to 0. Average error is likely to converge to zero as r_{nmf} and r_{svd} increases so there is a trend to 0. To lessen the processing time Y_k, $Y_k$is normalized and iterations are bounded to 100. Average error would decrease more as the iteration number increases for all noise rates. According to the Figures 6, 7 and 8, noise rate influenced the performance of the two methods. When $\eta = 1$ and $\eta = 10$ average error is approximately 10-12 but when $\eta = 25$ error is approximately 15-20. However, for different noise rates SVD always gave better results than NMF.

Part 3

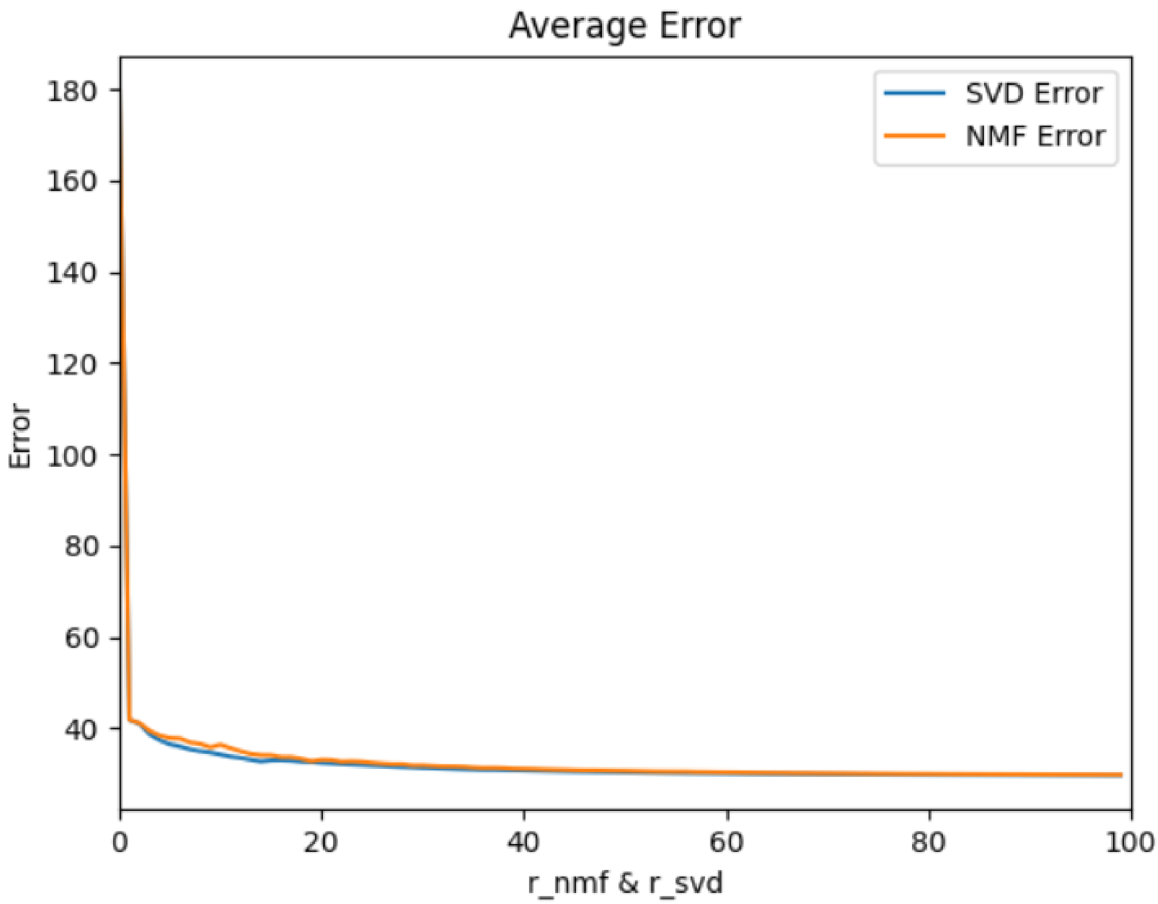

Figure 9: Average of the errors over the test images versus different r_{svd}, r_{nmf} values

By looking at Figure 9, it could be said that SVD is slightly better approach than NMF since blue curves are more likely to converge to 0. In this task, SVD and NMF approaches gave more similar results than in the second part's task. The average error difference was more obvious in Figures 6, 7 and 8 but in Figure 9 the difference is not that obvious.

Average error is likely to converge to zero as r_{nmf} and r_{svd} increases so there is a trend to 0. Average error would decrease more as the iteration number increases. However, for different noise rates SVD always gave better results than NMF.

Appendix

Homework is implemented in Python.

```
import matplotlib.pyplot as plt
import cv2 as cv
from skimage.measure import compare_ssim
import matplotlib.pyplot as plt
import numpy as np
import csv

# A function to read .pgm files. This function is taken from the following
link
# link: https://stackoverflow.com/questions/35723865/read-a-pgm-file-in-
python/60657836#60657836
def read_pgm(pgmf):
    pgmf.readline() == 'P5\n'
    (width, height) = [int(i) for i in pgmf.readline().split()]
    depth = int(pgmf.readline())
    assert depth <= 255

    raster = []
    for y in range(height):
        row = []
        for y in range(width):
            row.append(ord(pgmf.read(1)))
        raster.append(row)
    return raster


# Part 1.1.a
train_arr = []
for x in range(1, 2430):
    train_directory = "Dataset/train/face0" + "{:04n}".format(x) + ".pgm"
    f = open(train_directory, 'rb')
    train_arr.append(read_pgm(f))
train_arr = np.array(train_arr)
train_arr = train_arr.reshape([-1, 361])
X_first = np.array(np.transpose(train_arr))
X_first = X_first / 255
u, s, vh = np.linalg.svd(X_first)
# Part 1.1.a Son

# Part 1.1.b
def part1_1_b(s):
    s = s
    s_diag = np.zeros((361, 361), dtype=float)
    np.fill_diagonal(s_diag, s[:361])

# singular values are written as csv to monitor it, uncomment to see
normalized energy array
    # with open("singular_value_matrix.csv", 'w', newline='') as a_file:
    #     csv.writer(a_file, delimiter=',').writerows(s_diag)
    # a_file.close()

    energy = np.zeros(361, dtype=float)
    prev_energy = 0
    for x in range(361):
        energy[x] = s[x] ** 2 + prev_energy
        prev_energy = energy[x]
```

```
    max_e = np.max(energy)
    normalized_energy = energy / max_e
    normalized_energy = np.array(normalized_energy)

# Singular values and normalized energy plot code, uncomment to print plots
    # plt.plot(s_diag)
    # plt.xlim(1, 361)
    # plt.ylim(0, 50)
    # plt.title("Singular Values")
    # plt.xlabel("i")
    # plt.ylabel("Values")
    # plt.show()
    #
    # plt.plot(normalized_energy)
    # plt.xlim(1, 361)
    # plt.title("Normalized Energy")
    # plt.xlabel("i")
    # plt.ylabel("Energy")
    # plt.show()

# normalized energy is written as csv to monitor it, uncomment to see
normalized energy array
    # with open("normalized_energy.csv", 'w', newline='') as a_file:
    #     csv.writer(a_file, delimiter=',').writerow(normalized_energy)
    # a_file.close()
    # return normalized_energy
# Part 1.1.b Son

# Part 1.1.c
def part1_1_c(normalized_energy):
    normalized_energy = normalized_energy
    check_99 = True
    check_95 = True
    check_90 = True
    for x in range(normalized_energy.size):
        if normalized_energy[x] >= 0.99 and check_99:
            indice_99 = x
            check_99 = False
        if normalized_energy[x] >= 0.95 and check_95:
            indice_95 = x
            check_95 = False
        if normalized_energy[x] >= 0.90 and check_90:
            indice_90 = x
            check_90 = False
    return indice_99, indice_95, indice_90
# Part 1.1.c Son

# Part 1.1.d
normalized_energy = part1_1_b(s=s)
indice_99, indice_95, indice_90 = part1_1_c(normalized_energy)

fig, ax = plt.subplots(1, indice_90)
x = 0
while x in range(indice_90):
    img_x = np.reshape(u[:, x], (19, 19))

    ax[x].imshow(img_x, cmap='gray')
    ax[x].set_title(x.__str__())
    x += 1
plt.show()
# Part 1.1.d Son
```

Cem Daloğlu
21602161 / 29.02.2021

```python
# Part 1.2.a
def initialization(rank_value, unitary_matrix_u, singular_values,
unitary_matrix_vh):
    W_begin = np.zeros([361, rank_value])
    H_begin = np.zeros([rank_value, unitary_matrix_vh[0].size])
    for k_r in range(rank_value):
        u_plus_arr, u_minus_arr, vh_plus_arr, vh_minus_arr = [], [], [], []
        u_plus_arr.append(np.maximum(0, unitary_matrix_u[:, k_r]))
        u_minus_arr.append(np.maximum(0, -1 * unitary_matrix_u[:, k_r]))
        vh_plus_arr.append(np.maximum(0, unitary_matrix_vh[:, k_r]))
        vh_minus_arr.append(np.maximum(0, -1 * unitary_matrix_vh[:, k_r]))

        u_minus_arr = np.array(u_minus_arr)
        u_plus_arr = np.array(u_plus_arr)
        vh_plus_arr = np.array(vh_plus_arr)
        vh_minus_arr = np.array(vh_minus_arr)

        M_plus_arr = np.outer(u_plus_arr, vh_plus_arr.T)
        M_minus_arr = np.outer(u_minus_arr, vh_minus_arr.T)

        if np.linalg.norm(M_plus_arr) >= np.linalg.norm(M_minus_arr):
            w_k = u_plus_arr / np.linalg.norm(u_plus_arr)
            h_k_trans = singular_values[k_r] * np.linalg.norm(u_plus_arr) *
vh_plus_arr
        else:
            w_k = u_minus_arr / np.linalg.norm(u_minus_arr)
            h_k_trans = singular_values[k_r] * np.linalg.norm(u_minus_arr)
* vh_minus_arr
        W_begin[:, k_r] = w_k
        H_begin[k_r, :] = h_k_trans
    return W_begin, H_begin


def hals_update(x_hals, w_hals, h_hals, rank):
    for l in range(rank):
        temp_sum = np.zeros(np.expand_dims(w_hals[:, l], 1).shape)
        for k in range(rank):
            if k != l:
                temp_sum += np.outer(w_hals[:, k], (h_hals[k, :] @
h_hals[l, :].T))
        temp_sum = np.array(temp_sum)
        pay_w_update = (x_hals @ h_hals[l, :].T - temp_sum.T)
        payda_w_update = np.linalg.norm(h_hals[l, :]) ** 2
        w_nonegative = np.maximum(np.ones(pay_w_update.shape) * 1e-10,
(pay_w_update / payda_w_update))
        w_hals[:, l] = w_nonegative[0].T
    return w_hals


def part1_2(X_first, rank, u, s, vh):
    rank = rank
    W, H = initialization(rank, u, s, vh.T)
    # conv_x = []
    for x in range(rank):
        W = hals_update(X_first, W, H, rank)
        H = hals_update(X_first.T, H.T, W.T, rank).T

    # Convergence of iterations plot code, uncomment to use it.
    #     X_reconstruct = W @ H
```

```
    #     conv_x.append(np.linalg.norm((X_first - X_reconstruct), 'fro'))
    # conv_x = np.array(conv_x / np.linalg.norm(X_first))
    # plt.plot(conv_x)
    # plt.xlim(1, 100)
    # plt.title("Convergence of Iterations")
    # plt.xlabel("r")
    # plt.ylabel("Convergence to he Original Image")
    # plt.show()

    # Reconstructed image is constructed below, uncomment to see.
    # X_reconstruct = W @ H
    # img_x = np.reshape(X_first[:, 0], (19, 19))
    # img_reconstruct = np.reshape(X_reconstruct[:, 0], (19, 19))
    #
    # fig, ax = plt.subplots(1, 2)
    # ax[0].imshow(img_x, cmap='gray')
    # ax[0].set_title("Original Image")
    # ax[1].imshow(img_reconstruct, cmap='gray')
    # ax[1].set_title("Reconstructed Image")
    # plt.show()

    # Eigenface print face code, uncomment to use.
    # fig, ax = plt.subplots(1, 5)
    # x = 0
    # while x in range(5):
    #     img_x = np.reshape(W[:, x], (19, 19))
    #
    #     ax[x].imshow(img_x, cmap='gray')
    #     ax[x].set_title(x.__str__())
    #     x += 1
    # plt.show()
    return W, H, rank
# Part 1.2.a Son

# Part 2
test_arr = []
for x in range(1, 472):
    test_directory = "Dataset/test/cmu_0" + "{:03n}".format(x) + ".pgm"
    f = open(test_directory, 'rb')
    test_arr.append(read_pgm(f))
test_arr = np.array(test_arr)
test_arr = test_arr.reshape([-1, 361])
y_first = np.array(np.transpose(test_arr))

u, s, vh = np.linalg.svd(y_first)

y_noisy_k = []
error_rate = 1
y_noisy_k.append(np.minimum(255, y_first + error_rate * np.random.rand()))
y_noisy_k = np.array(y_noisy_k)
y_noisy_k = y_noisy_k.reshape([361, 471])
y_noisy_k = y_noisy_k / 255

normalized_energy = part1_1_b(s=s)
indice_99, indice_95, indice_90 =
part1_1_c(normalized_energy=normalized_energy)


def svd_nmf_error(y_noisy, y_first, iter):
    y_first = y_first.reshape([361, 471])
    y_noisy = y_noisy
```

Cem Daloğlu
21602161 / 29.02.2021

```
    y_svd = np.zeros([361, 471])
    error_svd = []
    for r_svd in range(0, iter):
        y_svd = u[:, :r_svd] @ u[:, :r_svd].T @ y_noisy
        y_svd_error = np.zeros([361, 471])
        for k in range(471):
            y_svd_error[:, k] = np.linalg.norm(y_first - y_svd)
        error_svd.append(1 / 472 * np.linalg.norm(y_svd_error, 'fro'))
    plt.plot(error_svd)

    error_nmf = []
    for r_nmf in range(0, iter):
        W, H, rank = part1_2(X_first=y_first, rank=r_nmf, u=u, s=s, vh=vh)
        y_nmf = np.zeros([361, 471])
        beta = (np.linalg.inv(W.T @ W) @ W.T)

        y_nmf = W[:, :r_nmf] @ beta[:r_nmf, :] @ y_noisy

        y_nmf_error = np.zeros([361, 471])
        for k in range(471):
            y_difference = y_first - y_nmf
            y_nmf_error[:, k] = np.linalg.norm(y_difference)

        error_nmf.append(1 / 472 * np.linalg.norm(y_nmf_error, 'fro'))
    plt.plot(error_nmf)
    plt.xlim(0, iter)
    plt.title("Average Error")
    plt.xlabel("r_nmf & r_svd")
    plt.ylabel("Error")
    plt.legend(["SVD Error", "NMF Error"])
    plt.show()
# part 2 Son

# part 3
y_noisy_s = np.zeros([19, 19, 471])
y_first = y_first.reshape([19, 19, -1])
for k in range(471):
    for i in range(19):
        for j in range(19):
            if j <= 10:
                error_rate = 1
            else:
                error_rate = 1 - 0.05 * (j - 10)
            y_noisy_s[i, j, k] = (y_first[i, j, k] * error_rate)
y_noisy_s = np.array(y_noisy_s)
y_noisy_s = y_noisy_s.reshape([361, 471])
y_noisy_s = y_noisy_s / 255
y_first = y_first / 255
# part 3 son

svd_nmf_error(y_noisy=y_noisy_s, y_first=y_first, iter=100)
```