Project 3

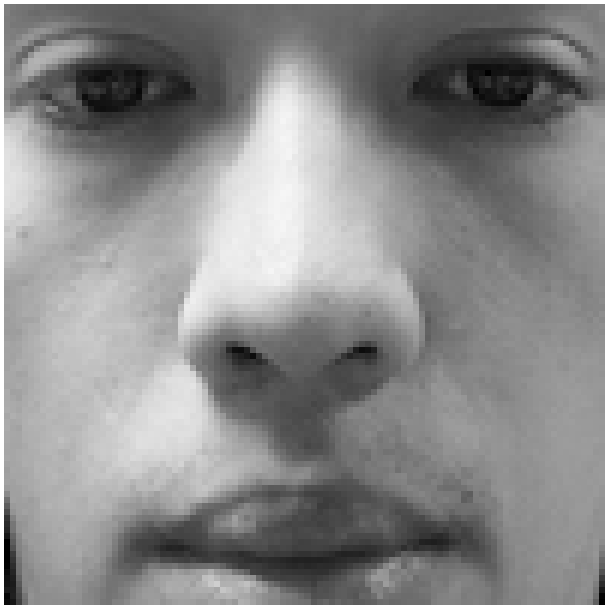
CSE 402

David Rackerby

1. Select one of your own frontal face photo (“selfie"). Crop this photo so that only the face is seen. Next, rescale the cropped face photo to a size of 100 × 100. Finally, convert the cropped and scaled photo from color to grayscale. Use this 100 × 100 grayscale image to do the following.

(a) [5 points] Apply the following operations to the grayscale image: (a) increase brightness; (b) improve contrast; (c) smoothen the image using a Gaussian Filter. You can use any set of parameter values for each of these operations. Display the original grayscale image (say, F0) along with the 3 modified images (say, FB, FC, FG) in your report.

F_0:



F_B (note that F_B and F_C were created in Apple's Preview app:

F_C:

F_G: ```m % simpleGaussian.m img = imread('proj3_1_selfie_grayscale.jpg'); img_blurred = imgaussfilt(img, 1); imwrite(img_blurred, 'proj3_1_selfie_grayscale_gaussian.jpg'); ```

(b) [10 points] For each of the 4 images, compute the corresponding LBP image with P=8, R=1. You can set the border pixels of each LBP image to 0. Display the 4 LBP images (say, L0, LB, LC, LG) in your report.

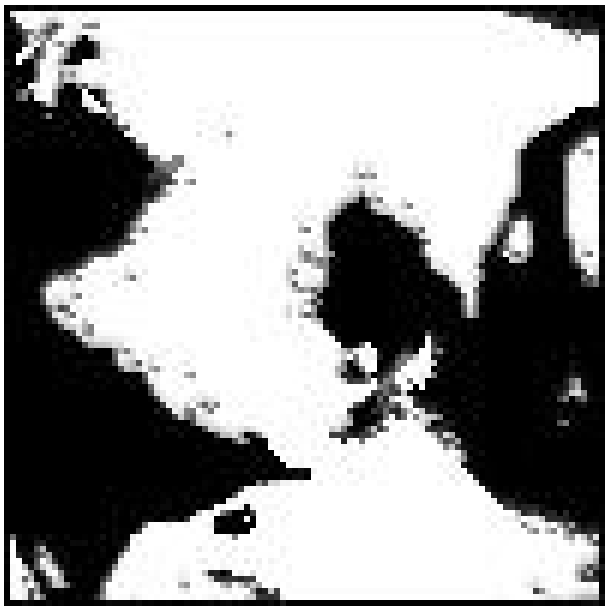
In [2]:

```
import numpy as np
import PIL
import matplotlib.pylab as plt

def compute_lbp_image(p: int, r: int, file: str) -> np.ndarray:
    img = PIL.Image.open(file)
    img_arr = np.array(PIL.ImageOps.grayscale(img))
    out = np.zeros((len(img_arr), len(img_arr[0])), dtype=np.int8)
    for i in range(r, len(img_arr) - r):
        for j in range(r, len(img_arr[i]) - r):
            out[i][j] = lbp(i, j, img_arr)
    return out

def lbp(i: int, j: int, arr: np.ndarray) -> int:
    neighbor = [
        lambda x, y: arr[y-1][x-1],
        lambda x, y: arr[y-1][x],
        lambda x, y: arr[y-1][x+1],
        lambda x, y: arr[y][x+1],
        lambda x, y: arr[y+1][x+1],
        lambda x, y: arr[y+1][x],
        lambda x, y: arr[y+1][x-1],
        lambda x, y: arr[y][x-1],
    ]

    return sum(int(neighbor[k](i, j) >= arr[i][j]) * (1 << k) for k in range

original_lbp = compute_lbp_image(8, 1, "proj3_1_selfies/proj3_1_selfie_grays
brighter_lbp = compute_lbp_image(8, 1, "proj3_1_selfies/proj3_1_selfie_grays
higher_contrast_lbp = compute_lbp_image(8, 1, "proj3_1_selfies/proj3_1_selfi
gaussian_lbp = compute_lbp_image(8, 1, "proj3_1_selfies/proj3_1_selfie_grays
```

```
PIL.Image.fromarray(original_lbp, mode='L').save("proj3_1_lbp_images/origina
PIL.Image.fromarray(brighter_lbp, mode='L').save("proj3_1_lbp_images/brighte
PIL.Image.fromarray(higher_contrast_lbp, mode='L').save("proj3_1_lbp_images/
PIL.Image.fromarray(gaussian_lbp, mode='L').save("proj3_1_lbp_images/gaussia
```

L_0:

L_B:



L_C:

L_G:

(c) [5 points] Write a program that compares L0 with each of LB, LC and LG (so there will be 3 comparison scores). This must be done as follows: compare each pixel in one image with the corresponding pixel in the other image, take the absolute difference between the two, sum all the absolute differences, and then divide this sum by the total number of pixels. Report the 3 scores.

In [3]:
```
def compare_images(im1: np.ndarray, im2: np.ndarray) -> float:
    left = im1.astype('int64')
    right = im2.astype('int64')
    return sum(abs(left[i][j] - right[i][j]) for i in range(len(im1)) for j

print(f"L_0 with L_B: {compare_images(original_lbp, brighter_lbp)}")
print(f"L_0 with L_C: {compare_images(original_lbp, higher_contrast_lbp)}")
print(f"L_0 with L_G: {compare_images(original_lbp, gaussian_lbp)}")
```

```
L_0 with L_B: 6.1331
L_0 with L_C: 5.4606
L_0 with L_G: 5.1367
```

(d) [5 points] Also report the scores obtained when comparing F0 with each of FB, FC and FG. Use the same procedure as above to compute the comparison scores.

In [4]:
```
original = PIL.Image.open("proj3_1_selfies/proj3_1_selfie_grayscale.jpg")
original = np.array(PIL.ImageOps.grayscale(original))

brighter = PIL.Image.open("proj3_1_selfies/proj3_1_selfie_grayscale_brighter
brighter = np.array(PIL.ImageOps.grayscale(brighter))

higher_contrast = PIL.Image.open("proj3_1_selfies/proj3_1_selfie_grayscale_h
higher_contrast = np.array(PIL.ImageOps.grayscale(higher_contrast))

gaussian = PIL.Image.open("proj3_1_selfies/proj3_1_selfie_grayscale_gaussian
gaussian = np.array(PIL.ImageOps.grayscale(gaussian))

print(f"F_0 with F_B: {compare_images(original, brighter)}")
print(f"F_0 with F_C: {compare_images(original, higher_contrast)}")
print(f"F_0 with F_G: {compare_images(original, gaussian)}")
```

```
F_0 with F_B: 82.5362
F_0 with F_C: 77.8776
F_0 with F_G: 3.3261
```

(e) [5 points] Discuss your observations. In particular, does applying the LBP operator reduce the intra-class variation due to changes in pixel intensity?

Yes, applying the LBP operator here makes a dramatic reduction in intra-class variation due to changes in pixel intensity. When comparing the original image with the brightened one, the LBP operator yielded a 92.57% decrease in distance. For comparing the original with the high-contrast image, there was a 92.99% decrease in distance. Both of the above filters transformed the pixel intensity of the original image, and LBP was a suitable way to account for this transformation. There was a minor increase in distance when comparing the original with the gaussian-filtered image, but this was expected due to the nature of the gaussian filter, which acts more to average out pixel values than a direct intensity change. The post-LBP distance in this situation is still rather small such that a matcher would reasonably match the images.

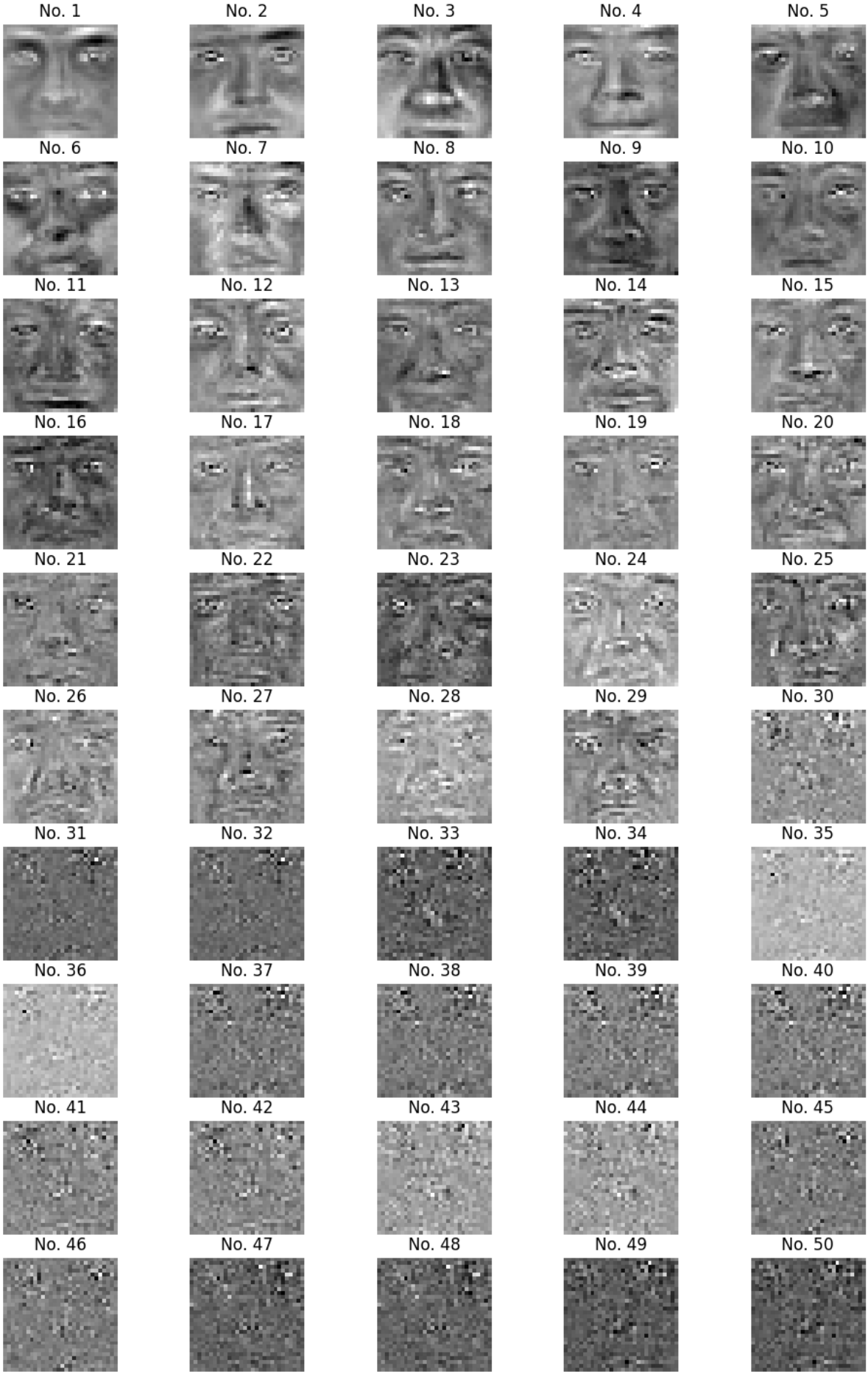
2. You are given a set of 50 face images pertaining to 10 different subjects (5 images per subject).

(a) [10 points] Compute the eigen-faces (i.e., basis images) using 30 face images (first 3 images per subject). Show the mean face as well as the eigen-faces corresponding to the top 50 eigen-values.

In [5]:
```
import matplotlib.pyplot as plt
def image_to_array(path: str) -> np.ndarray:
```

```
    im = np.array(PIL.Image.open(path).convert('L'))
    return im.reshape(len(im) * len(im[0]))

# 30 faces being used
face_images = [
    "proj3_2_face_images/user01_01.bmp",
    "proj3_2_face_images/user01_02.bmp",
    "proj3_2_face_images/user01_03.bmp",
    "proj3_2_face_images/user02_01.bmp",
    "proj3_2_face_images/user02_02.bmp",
    "proj3_2_face_images/user02_03.bmp",
    "proj3_2_face_images/user03_01.bmp",
    "proj3_2_face_images/user03_02.bmp",
    "proj3_2_face_images/user03_03.bmp",
    "proj3_2_face_images/user04_01.bmp",
    "proj3_2_face_images/user04_02.bmp",
    "proj3_2_face_images/user04_03.bmp",
    "proj3_2_face_images/user05_01.bmp",
    "proj3_2_face_images/user05_02.bmp",
    "proj3_2_face_images/user05_03.bmp",
    "proj3_2_face_images/user06_01.bmp",
    "proj3_2_face_images/user06_02.bmp",
    "proj3_2_face_images/user06_03.bmp",
    "proj3_2_face_images/user07_01.bmp",
    "proj3_2_face_images/user07_02.bmp",
    "proj3_2_face_images/user07_03.bmp",
    "proj3_2_face_images/user08_01.bmp",
    "proj3_2_face_images/user08_02.bmp",
    "proj3_2_face_images/user08_03.bmp",
    "proj3_2_face_images/user09_01.bmp",
    "proj3_2_face_images/user09_02.bmp",
    "proj3_2_face_images/user09_03.bmp",
    "proj3_2_face_images/user10_01.bmp",
    "proj3_2_face_images/user10_02.bmp",
    "proj3_2_face_images/user10_03.bmp"
]

face_arrays = [image_to_array(path) for path in face_images]
mean_face = np.sum(face_arrays, axis=0) / len(face_arrays)
data_matrix = np.array([face - mean_face for face in face_arrays])
cov = np.matmul(np.transpose(data_matrix), data_matrix)
eigenvals, eigenvecs = np.linalg.eig(cov)
pairs = [(eigenvals[i], eigenvecs[:, i]) for i in range(len(eigenvals))]
pairs.sort(key=lambda p: p[0], reverse=True)

def get_n_eigenfaces(n: int) -> list[np.ndarray]:
    return [pairs[i][1] for i in range(n)]

top_50 = get_n_eigenfaces(50)

# Show the faces, starting from the mean face
%matplotlib inline
mean_mat = np.reshape(mean_face, (30, 30))
plt.gray()
plt.imshow(mean_mat)
plt.axis('off')
```

```
plt.title('Mean face')

# Show the eigenfaces
fig = plt.figure(figsize=(12, 18))
rows = 10
cols = 5
for i, eigenface in enumerate(top_50):
    eigenface_matrix = np.reshape(eigenface, (30, 30))
    fig.add_subplot(rows, cols, i+1)
    plt.imshow(np.real(eigenface_matrix))
    plt.axis('off')
    plt.title(f"No. {i+1}")
```


No. 1
No. 2
No. 3
No. 4
No. 5
No. 6
No. 7
No. 8
No. 9
No. 10
No. 11
No. 12
No. 13
No. 14
No. 15
No. 16
No. 17
No. 18
No. 19
No. 20
No. 21
No. 22
No. 23
No. 24
No. 25
No. 26
No. 27
No. 28
No. 29
No. 30
No. 31
No. 32
No. 33
No. 34
No. 35
No. 36
No. 37
No. 38
No. 39
No. 40
No. 41
No. 42
No. 43
No. 44
No. 45
No. 46
No. 47
No. 48
No. 49
No. 50

(b) [10 points] Using the mean face and the top 50 eigen-faces (i.e., eigen-vectors), compute the eigen-coefficients (i.e., the 50-dimensional feature vector) for each of the 50 images in the dataset, including the 30 face images you had used in 2a.

In [6]:
```
all_images = [
    [
        "proj3_2_face_images/user01_01.bmp",
        "proj3_2_face_images/user01_02.bmp",
        "proj3_2_face_images/user01_03.bmp",
        "proj3_2_face_images/user01_04.bmp",
        "proj3_2_face_images/user01_05.bmp"
    ],
    [
        "proj3_2_face_images/user02_01.bmp",
        "proj3_2_face_images/user02_02.bmp",
        "proj3_2_face_images/user02_03.bmp",
        "proj3_2_face_images/user02_04.bmp",
        "proj3_2_face_images/user02_05.bmp"
    ],
    [
        "proj3_2_face_images/user03_01.bmp",
        "proj3_2_face_images/user03_02.bmp",
        "proj3_2_face_images/user03_03.bmp",
        "proj3_2_face_images/user03_04.bmp",
        "proj3_2_face_images/user03_05.bmp"
    ],
    [
        "proj3_2_face_images/user04_01.bmp",
        "proj3_2_face_images/user04_02.bmp",
        "proj3_2_face_images/user04_03.bmp",
        "proj3_2_face_images/user04_04.bmp",
        "proj3_2_face_images/user04_05.bmp"
    ],
    [
        "proj3_2_face_images/user05_01.bmp",
        "proj3_2_face_images/user05_02.bmp",
        "proj3_2_face_images/user05_03.bmp",
        "proj3_2_face_images/user05_04.bmp",
        "proj3_2_face_images/user05_05.bmp"
    ],
    [
        "proj3_2_face_images/user06_01.bmp",
        "proj3_2_face_images/user06_02.bmp",
        "proj3_2_face_images/user06_03.bmp",
        "proj3_2_face_images/user06_04.bmp",
        "proj3_2_face_images/user06_05.bmp"
    ],
    [
        "proj3_2_face_images/user07_01.bmp",
        "proj3_2_face_images/user07_02.bmp",
        "proj3_2_face_images/user07_03.bmp",
        "proj3_2_face_images/user07_04.bmp",
        "proj3_2_face_images/user07_05.bmp"
    ],
```

```
    [
        "proj3_2_face_images/user08_01.bmp",
        "proj3_2_face_images/user08_02.bmp",
        "proj3_2_face_images/user08_03.bmp",
        "proj3_2_face_images/user08_04.bmp",
        "proj3_2_face_images/user08_05.bmp"
    ],
    [
        "proj3_2_face_images/user09_01.bmp",
        "proj3_2_face_images/user09_02.bmp",
        "proj3_2_face_images/user09_03.bmp",
        "proj3_2_face_images/user09_04.bmp",
        "proj3_2_face_images/user09_05.bmp"
    ],
    [
        "proj3_2_face_images/user10_01.bmp",
        "proj3_2_face_images/user10_02.bmp",
        "proj3_2_face_images/user10_03.bmp",
        "proj3_2_face_images/user10_04.bmp",
        "proj3_2_face_images/user10_05.bmp"
    ]
]

def get_feature_vector(path: str, top_eigs: np.ndarray) -> np.ndarray:
    arr = image_to_array(path)
    diff = arr - mean_face
    return np.matmul(top_eigs, diff)

def get_weighted_sums(top_eigs: np.ndarray, group: np.ndarray) -> list[list[
    return [[get_feature_vector(filename, top_eigs) for filename in group] f

weighted_sums_list = get_weighted_sums(np.array(top_50), all_images)
weighted_sums_list
```

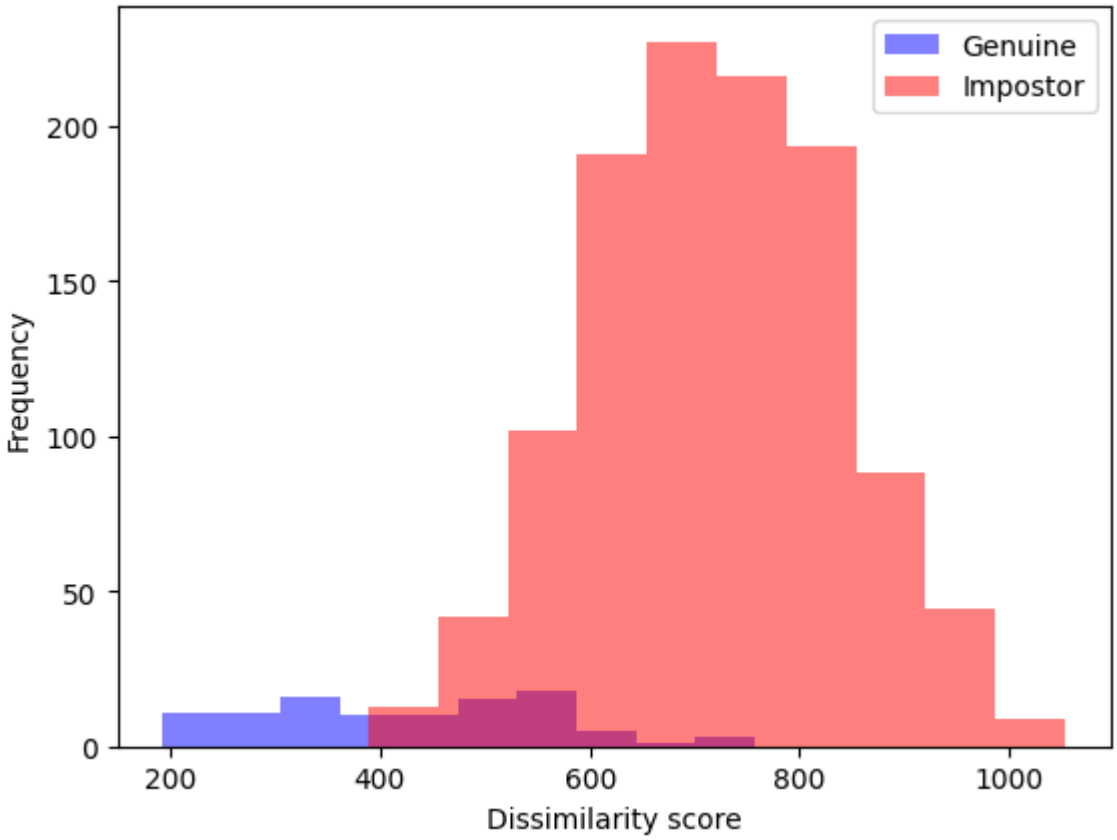
```
         -40.18870774 +0.j        ,   68.8822676  +0.j        ,
         -20.21214188 +0.j        ,  -95.23236544 +0.j        ,
        -166.68117762 +0.j        ,  -92.67836388 +0.j        ,
         -34.48780083 +0.j        ,  118.28012413 +0.j        ,
          94.28930323 +0.j        ,   66.92366826 +0.j        ,
          42.6568212  +0.j        ,   95.67893489 +0.j        ,
          11.16568002 +0.j        ,   23.36234594 +0.j        ,
         -98.42455652 +0.j        ,   51.15571096 +0.j        ,
          37.27686781 +0.j        ,  -25.59870578 +0.j        ,
          -7.17512364 +0.j        ,   12.37518824 +0.j        ,
        -114.77892056 +0.j        ,    7.44925764 +0.j        ,
           7.61070326 +0.j        ,   36.29131204 +0.j        ,
          24.08102476 +0.j        ,   18.47773652 +0.j        ,
          -0.27039439 +1.82960783j,   -0.27039439 -1.82960783j,
           5.82214733 -6.38544685j,    5.82214733 +6.38544685j,
          -0.72660978 +3.65341217j,   -0.72660978 -3.65341217j,
           1.717337   -2.24971232j,    1.717337   +2.24971232j,
           0.74247097-11.08949137j,    0.74247097+11.08949137j,
          26.2250876  +8.14804618j,   26.2250876  -8.14804618j,
          18.45018478-22.81582186j,   18.45018478+22.81582186j,
         -10.55689202+11.26016831j,  -10.55689202-11.26016831j,
          -9.4491284 -13.93336672j,   -9.4491284 +13.93336672j,
          -2.1116864  -8.53881447j,   -2.1116864  +8.53881447j]),
 array([  30.118793   +0.j        ,   63.06485272 +0.j        ,
         -94.47687197 +0.j        ,  -64.21654486 +0.j        ,
         -19.5874566  +0.j        ,   21.49300415 +0.j        ,
         -96.88584215 +0.j        , -101.54674958 +0.j        ,
         -22.62838746 +0.j        , -122.82143462 +0.j        ,
          29.03960154 +0.j        ,    7.33678122 +0.j        ,
         -35.67140881 +0.j        ,    3.4999843  +0.j        ,
          37.88737601 +0.j        ,   15.49580836 +0.j        ,
          15.25523498 +0.j        ,  -50.9745674  +0.j        ,
         -59.92824526 +0.j        ,   85.76367319 +0.j        ,
          14.04094086 +0.j        ,  -11.88544576 +0.j        ,
          50.38729117 +0.j        ,  -16.97267708 +0.j        ,
          21.50620623 +0.j        ,   -6.27220741 +0.j        ,
          22.02094204 +0.j        ,   83.37673234 +0.j        ,
         -26.08847491 +0.j        ,    1.26002162 +0.j        ,
           1.51602164 -9.92386905j,    1.51602164 +9.92386905j,
          15.42512505 +9.90944821j,   15.42512505 -9.90944821j,
          11.24970856 +1.15885485j,   11.24970856 -1.15885485j,
           7.83458587 -0.68640789j,    7.83458587 +0.68640789j,
           3.5780491  -1.21039869j,    3.5780491  +1.21039869j,
          -2.10967374+14.24803762j,   -2.10967374-14.24803762j,
          -8.34144373 +7.93401899j,   -8.34144373 -7.93401899j,
          16.21187886 +7.02010514j,   16.21187886 -7.02010514j,
          20.19988431 +3.73611471j,   20.19988431 -3.73611471j,
          10.25151793-27.55285648j,   10.25151793+27.55285648j])]]
```

(c) [10 points] Generate genuine scores and impostor scores by computing the Euclidean distance between the feature vectors of every pair of face images. Plot the histograms of genuine and impostor scores in the same graph. Use a different color for each histogram.

In [7]:
```
def get_distributions_from_sums(weighted_sums: list[list[np.ndarray]]) -> tu
    seen = set()
    genuine_scores = []
    impostor_scores = []
    for i, group in enumerate(weighted_sums):
        for j, feature_v in enumerate(group):
            for k, other_group in enumerate(weighted_sums):
                for l, other_v in enumerate(other_group):
                    if (i, j) == (k, l) or (i, j, k, l) in seen or (k, l, i,
                        continue
                    if i == k:
                        genuine_scores.append(np.linalg.norm(feature_v - oth
                    else:
                        impostor_scores.append(np.linalg.norm(feature_v - ot
                    seen.add((i, j, k, l))
    return genuine_scores, impostor_scores
genuine, impostor = get_distributions_from_sums(weighted_sums_list)
np.savetxt("proj3_2_distributions/gen_50.csv", genuine, delimiter=',')
np.savetxt("proj3_2_distributions/imp_50.csv", impostor, delimiter=',')
print(f"Number of genuine scores: {len(genuine)}")
print(f"Number of impostor scores: {len(impostor)}")

plt.figure()
plt.xlabel("Dissimilarity score")
plt.ylabel("Frequency")
plt.hist(genuine, alpha=0.5, label='Genuine', color='blue')
plt.hist(impostor, alpha=0.5, label='Impostor', color='red')
plt.legend(loc='upper right')
```

```
Number of genuine scores: 100
Number of impostor scores: 1125
```

Out[7]:
```
<matplotlib.legend.Legend at 0x121995b40>
```

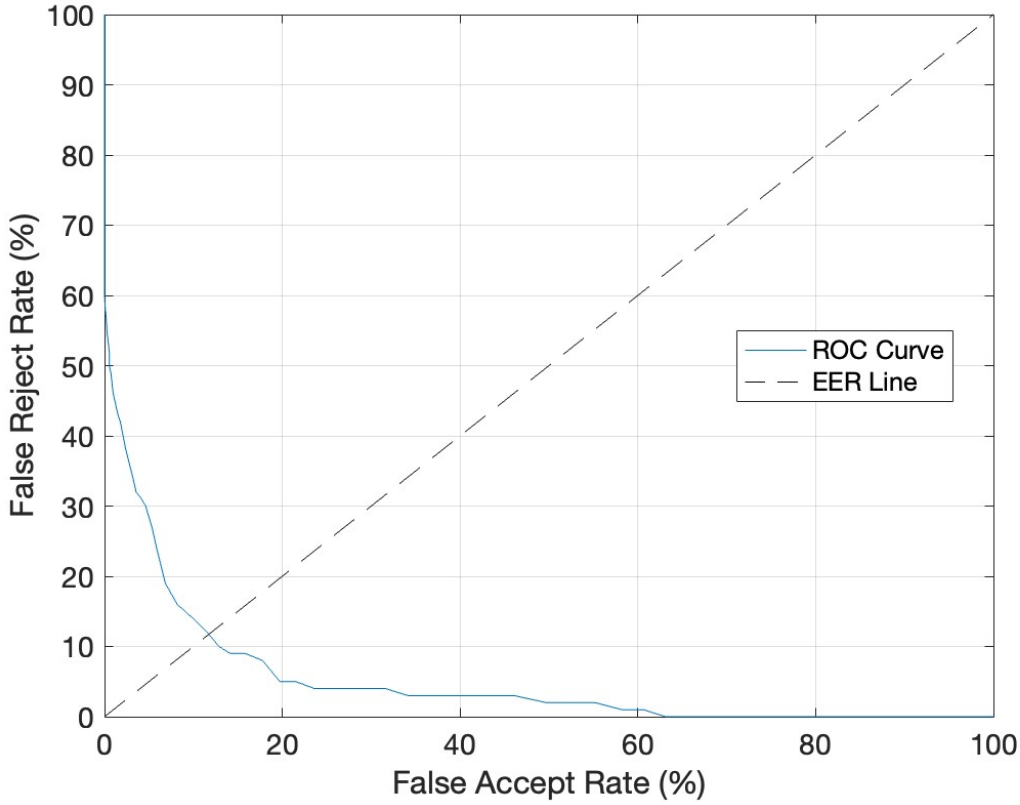


(d) [5 points] Plot the ROC curve summarizing the matching performance using these scores. You can use the matlab code available here to plot the ROC curve. The code can be invoked as roc(gen, imp, 'd'). Here, gen and imp are the set of genuine and impostor scores, respectively. 'd' denotes that the scores are distance (or dissimilarity) scores.

```
m
% drawCurves.m
gen = readmatrix('gen_50.csv');
imp = readmatrix('imp_50.csv');
drawROC(gen, imp, 'd');
```

Note: even though the project direction and provided MATLAB code refer to the ROC curve, the drawROC function itself actually plots the DET curve (since the axes are FNMR vs. FMR while an ROC curve should be 1 - FNMR vs. FMR)

Matching performance using the top 50 eigenfaces

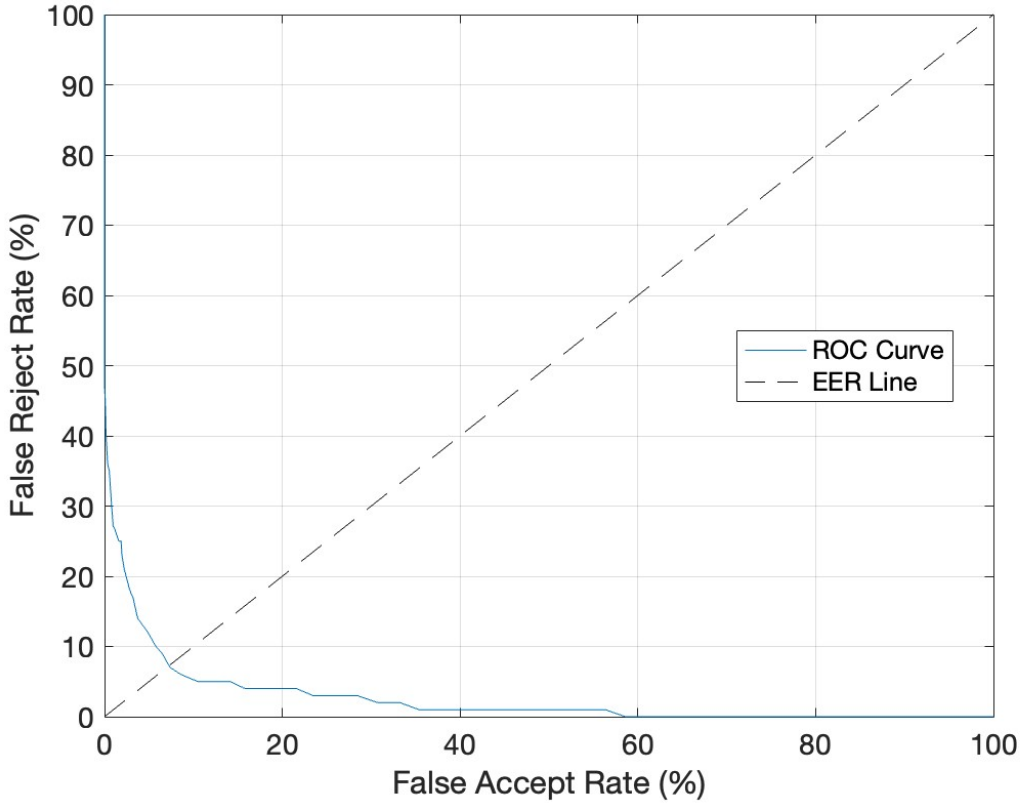


(e) [10 points] Repeat the above after selecting the top (i) 10, (ii) 20, (iii) 40 eigen-faces. Plot the ROC curves for each case. Comment on the change in matching performance as you vary the number of eigen-faces used to generate the feature vector.

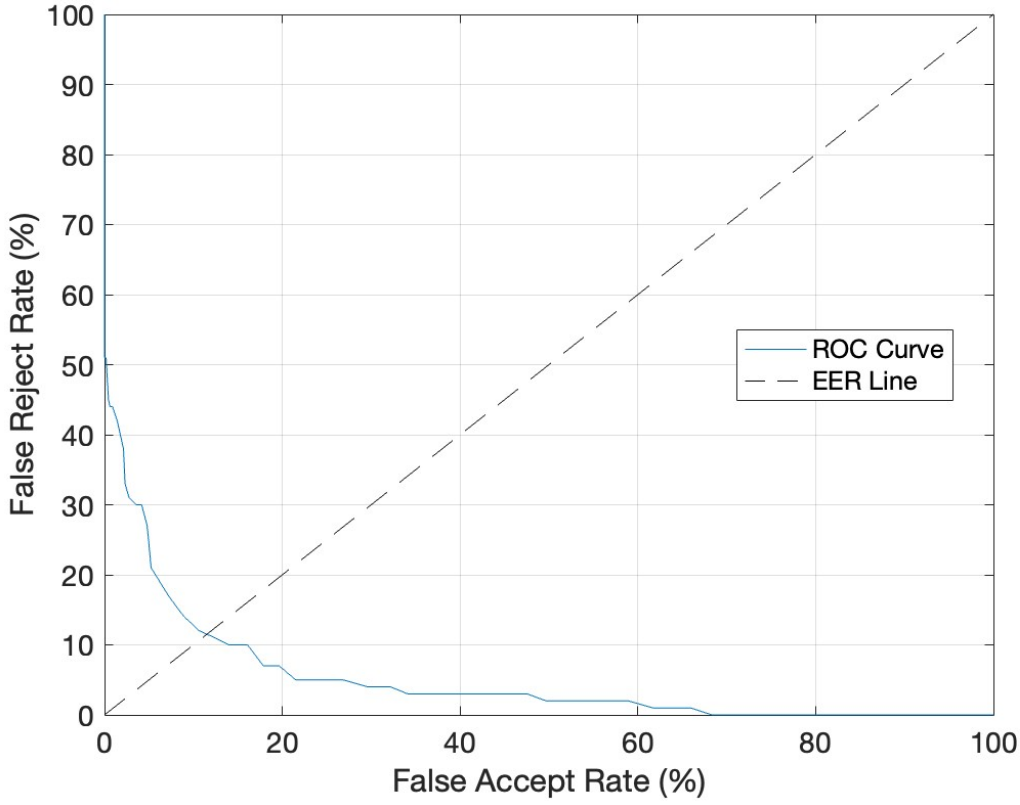
In [8]:
```
for face_count in (10, 20, 40):
    weighted_sums_list = get_weighted_sums(np.array(get_n_eigenfaces(face_co
    genuine, impostor = get_distributions_from_sums(weighted_sums_list)
    np.savetxt(f"proj3_2_distributions/gen_{face_count}.csv", genuine, delim
    np.savetxt(f"proj3_2_distributions/imp_{face_count}.csv", impostor, deli
```

```
m
% drawCurves.m
gen = readmatrix('gen_10.csv');
imp = readmatrix('imp_10.csv');
drawROC(gen, imp, 'd');

gen = readmatrix('gen_20.csv');
imp = readmatrix('imp_20.csv');
drawROC(gen, imp, 'd');

gen = readmatrix('gen_40.csv');
imp = readmatrix('imp_40.csv');
drawROC(gen, imp, 'd');
```

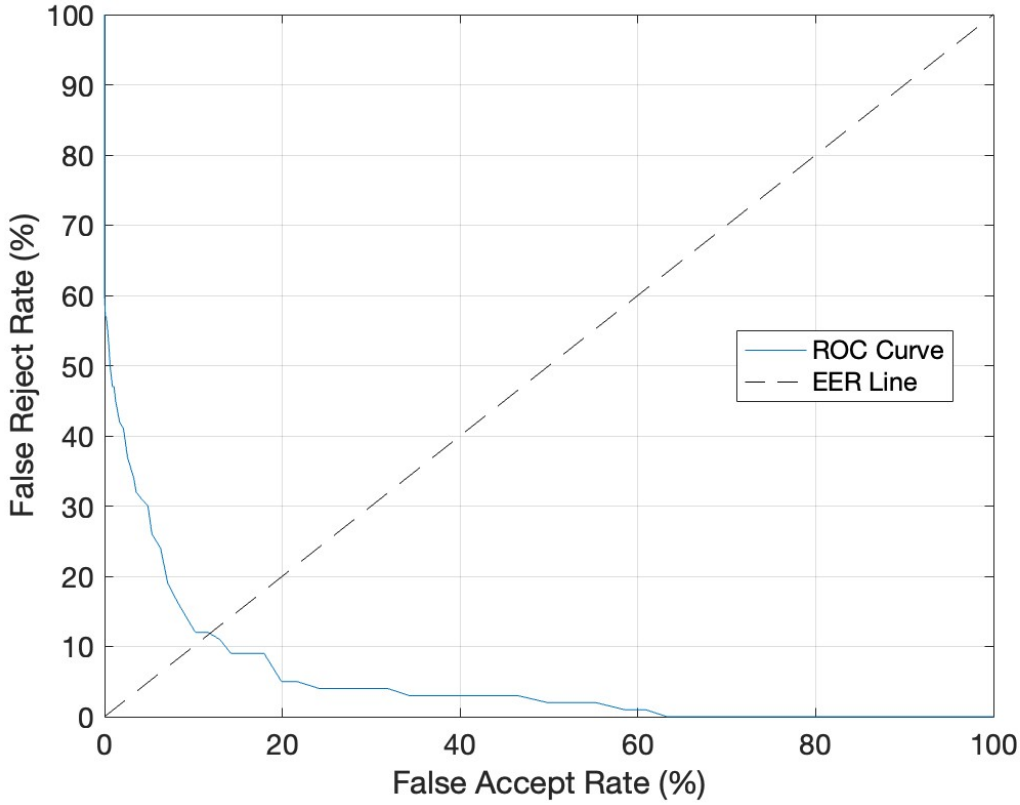
Matching performance using the top 10 eigenfaces



Matching performance using the top 20 eigenfaces



Matching performance using the top 40 eigenfaces



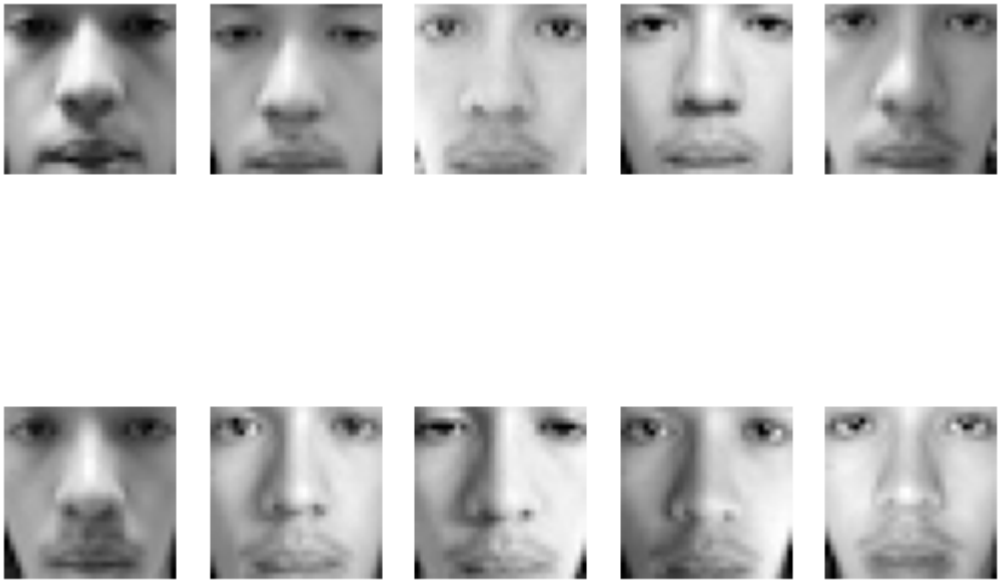
The prevailing difference between each curve is that the matcher becomes more performant when using fewer eigenfaces. Specifically, as the number of eigenfaces used decreases, the Equal Error Rate decreases and the curve itself tends closer to the origin, which for this type of curve is desirable.

3. Select 10 of your own frontal face photos (“selfies"). Crop each photo so that only the face is seen. Next, rescale each cropped face photo to a size of 30 × 30. Finally, convert the cropped and scaled photos from color to grayscale.

[5 points] Display the 10 grayscale images.

In [15]:
```
selfies = [
    [
        "proj3_3_frontal_selfies/selfie_01.jpg",
        "proj3_3_frontal_selfies/selfie_02.jpg",
        "proj3_3_frontal_selfies/selfie_03.jpg",
        "proj3_3_frontal_selfies/selfie_04.jpg",
        "proj3_3_frontal_selfies/selfie_05.jpg",
        "proj3_3_frontal_selfies/selfie_06.jpg",
        "proj3_3_frontal_selfies/selfie_07.jpg",
        "proj3_3_frontal_selfies/selfie_08.jpg",
        "proj3_3_frontal_selfies/selfie_09.jpg",
        "proj3_3_frontal_selfies/selfie_10.jpg"
    ]
]
```

```
fig = plt.figure()
plt.gray()
rows = 2
cols = 5
for i, filename in enumerate(selfies[0]):
    img = np.array(PIL.Image.open(filename).convert('L'))
    fig.add_subplot(rows, cols, i+1)
    plt.imshow(img)
    plt.axis('off')
```



[5 points] Compute the eigen-coefficients of the 10 faces using the top 25 eigen-faces (eigen-vectors) computed in 2a.

In [16]:
```
weighted_sums_list = get_weighted_sums(np.array(get_n_eigenfaces(25)), selfi
weighted_sums_list
```

Out[16]:
```
[[array([   1.26264587+0.j,  168.83554375+0.j,  111.75888458+0.j,
          -41.52495182+0.j,   22.99949179+0.j,   23.85607399+0.j,
           42.18890823+0.j,  -50.00004329+0.j,  -86.714909  +0.j,
         -153.01996011+0.j, -210.19201089+0.j,   17.380884  +0.j,
         -184.40834117+0.j,  -89.22953575+0.j,   81.16176185+0.j,
          155.11797684+0.j,  -59.32320547+0.j,  202.73656449+0.j,
          152.34354824+0.j, -102.28679755+0.j,   29.62270271+0.j,
          -74.98631801+0.j,  -11.7373303 +0.j,   62.42622823+0.j,
          -50.88322469+0.j]),
  array([  62.70535717+0.j,   10.76280415+0.j,  -11.91021072+0.j,
         -219.93872401+0.j, -213.00792013+0.j,    2.00099049+0.j,
           24.73448099+0.j,  -20.47186623+0.j, -150.05787168+0.j,
         -124.98872004+0.j, -162.7269907 +0.j,   60.31592664+0.j,
          -18.39411003+0.j,  -71.42651553+0.j,   88.51779316+0.j,
           -8.9831417 +0.j,  -82.08135729+0.j,  256.15761347+0.j,
           79.8548923 +0.j,  -15.0243753 +0.j,   91.82133116+0.j,
          -80.72185021+0.j,   53.37708462+0.j,   36.6809119 +0.j,
          -58.64926088+0.j]),
  array([  31.95106739+0.j,   11.97882484+0.j,  162.45176972+0.j,
          -66.86838939+0.j,   -1.43096595+0.j,   68.62485197+0.j,
          -16.9772454 +0.j,  -14.68403989+0.j,  -37.47008535+0.j,
          -29.55373546+0.j, -132.19781581+0.j,   14.85289313+0.j,
          -75.08905145+0.j,  -83.76790098+0.j,   68.15469479+0.j,
           64.61702834+0.j,  -26.80728818+0.j,   75.22942574+0.j,
           30.34331787+0.j,  -21.94895248+0.j,   16.10380754+0.j,
          -28.67181015+0.j,  -47.53705615+0.j,    9.37735796+0.j,
          -28.40223407+0.j]),
  array([-188.44958013+0.j,   24.19375514+0.j, -211.02403438+0.j,
           89.96571217+0.j, -168.51977568+0.j,   -0.81257234+0.j,
            1.22053137+0.j,   57.39941119+0.j, -140.82847275+0.j,
          -15.87837226+0.j,   26.40944907+0.j,    5.75454113+0.j,
            6.77177037+0.j,  -38.44693185+0.j, -121.60575181+0.j,
           85.9891267 +0.j,   96.82960903+0.j,   -4.84527098+0.j,
           71.9973437 +0.j,  -19.96843917+0.j,   64.02076829+0.j,
            4.77966702+0.j,   52.89463185+0.j,  -33.3939338 +0.j,
           32.48969829+0.j]),
  array([ -75.86525788+0.j,  -50.35358461+0.j,  191.57999252+0.j,
           63.4562531 +0.j,   91.43360511+0.j,    6.86665823+0.j,
           51.80603664+0.j,  -53.17820183+0.j,  -29.88889311+0.j,
          -82.17084803+0.j, -144.52585639+0.j,   16.66597372+0.j,
         -166.39717388+0.j, -211.61484944+0.j,   81.71531963+0.j,
           63.73632434+0.j,   42.78270623+0.j,  201.2072703 +0.j,
           20.37227309+0.j,  -40.26033507+0.j,   88.0500475 +0.j,
          -56.81592838+0.j,  -22.50541301+0.j,   54.5828647 +0.j,
          -70.09825279+0.j]),
  array([ -50.45823921+0.j,   20.84297407+0.j,   26.27909412+0.j,
          -19.64729851+0.j,  -17.67215587+0.j,  -91.55436714+0.j,
           -0.88777628+0.j,  -36.24899393+0.j, -120.90909929+0.j,
         -135.50191733+0.j, -172.38333992+0.j,  -31.77925803+0.j,
          -71.35913441+0.j, -124.89985583+0.j,   54.31544996+0.j,
           86.16510619+0.j,    1.45409247+0.j,  154.80563059+0.j,
          132.73595189+0.j,  -39.80608525+0.j,    1.9813212 +0.j,
          -84.25290364+0.j,   37.20558035+0.j,   42.95882174+0.j,
          -57.88119109+0.j]),
  array([-166.72472579+0.j,    9.77419301+0.j, -201.45935253+0.j,
           19.6271522 +0.j, -249.31523796+0.j,  -52.66913637+0.j,
```

```
        -96.51540657+0.j,  119.88485494+0.j, -164.00817265+0.j,
         30.32114052+0.j,   43.56480168+0.j,  -39.6509082 +0.j,
         63.98954294+0.j,   32.38336841+0.j, -108.45149252+0.j,
         32.07567053+0.j,   45.13298729+0.j,   27.17982703+0.j,
          9.85706813+0.j,   52.89025825+0.j,    0.53715622+0.j,
        -53.05662206+0.j,   75.66837813+0.j,  -31.07538763+0.j,
         58.50258895+0.j]),
 array([ -67.31390792+0.j,   75.36640657+0.j, -230.17580888+0.j,
        -58.55595078+0.j, -290.23405286+0.j,  -28.65162709+0.j,
       -163.4118923 +0.j,  126.43355497+0.j, -101.90924477+0.j,
         82.95883946+0.j,   43.32062336+0.j,   -4.71578538+0.j,
        102.21578731+0.j,   80.37525685+0.j, -162.14765359+0.j,
         42.15792593+0.j,   57.4334182 +0.j,  -77.95859559+0.j,
         37.42995569+0.j,   38.5815331 +0.j,   -8.84917723+0.j,
        -75.7302118 +0.j,   78.67383298+0.j,  -47.32959251+0.j,
         50.85258342+0.j]),
 array([ -55.94138764+0.j,   19.85329888+0.j, -266.63999015+0.j,
       -134.37388409+0.j, -345.93980556+0.j,  -74.04371975+0.j,
       -159.65098321+0.j,  165.25186846+0.j,  -63.73559269+0.j,
        111.07154051+0.j,   57.49827903+0.j,  -17.51495968+0.j,
        151.89560439+0.j,  106.996264  +0.j, -168.89287539+0.j,
        -32.2853179 +0.j,   25.52143349+0.j,  -56.67548956+0.j,
         -6.81980955+0.j,   84.67932652+0.j,    1.40384274+0.j,
        -40.92609557+0.j,  102.88076798+0.j,  -73.5748101 +0.j,
         82.36791359+0.j]),
 array([-139.73149243+0.j, -157.70877802+0.j,   -9.82984807+0.j,
        112.65501764+0.j,   24.28495837+0.j,  -60.68539679+0.j,
        111.85145378+0.j,   14.38672222+0.j,  -25.91017488+0.j,
        -15.68476944+0.j,  -76.11325474+0.j,  -44.73354807+0.j,
        -61.31832752+0.j, -184.44537131+0.j,   24.38327186+0.j,
         20.92294344+0.j,   18.44349249+0.j,  124.55391728+0.j,
          4.5365098 +0.j,    0.97668401+0.j,   41.15049363+0.j,
        -29.12402036+0.j,   23.97918598+0.j,   18.94758078+0.j,
          4.83925025+0.j])]]
```

[5 points] Compute the genuine scores between every pair of faces (there will be 45 genuine scores).

In [20]:
```
genuine_selfies, impostor_selfies = get_distributions_from_sums(weighted_sum
genuine_selfies
```

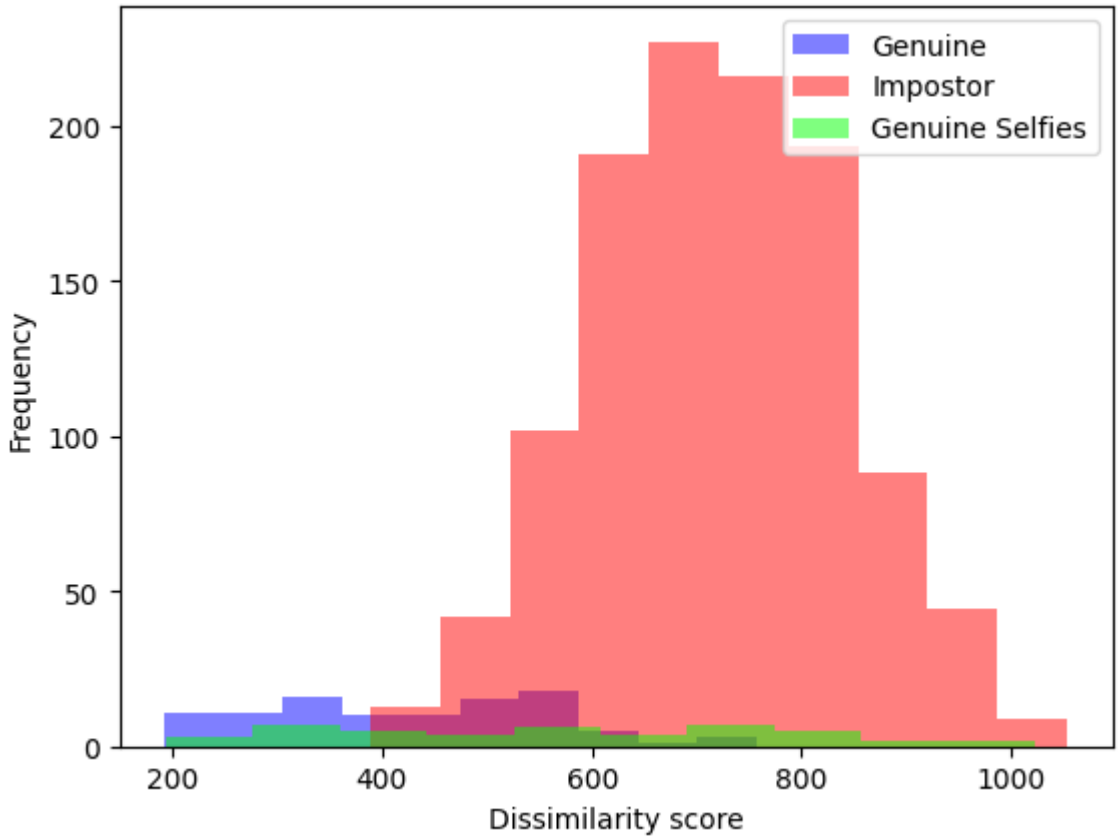
Out[20]:
```
[471.2774994422252,
 354.6660034379831,
 707.3379065405185,
 385.5601071871911,
 296.95724671937336,
 796.2925857578289,
 875.2120114821491,
 994.2719663942612,
 569.4153218674782,
 445.3244155690478,
 659.3184736722197,
 578.0593022504605,
 395.9362947481049,
 652.393349498715,
 720.7983352138725,
 771.2526472780551,
 598.616734052113,
 612.2744124531857,
 333.716088800271,
 330.1921176381471,
 661.2386305070543,
 724.4235779814918,
 821.1334926634519,
 434.49545191831294,
 698.6216576459695,
 540.361952760772,
 257.3609222617082,
 387.5016877693467,
 514.6878992321816,
 492.90871854507424,
 353.7708272470921,
 799.4448616556874,
 921.6037891045975,
 1022.8964606958832,
 355.0371859688007,
 606.329985344973,
 715.889255822392,
 822.9054083401885,
 383.1737695677913,
 235.78075009073368,
 326.3363213310721,
 585.5246056151456,
 193.6204915878241,
 740.5391073919089,
 817.5961257519889]
```

[5 points] Plot the histogram of 45 genuine scores on the same graph as 2c. For this part, you can re-plot the histograms from the previous question and use them here.

In [22]:
```
weighted_sums_list_50 = get_weighted_sums(np.array(top_50), all_images)
genuine_50, impostor_50 = get_distributions_from_sums(weighted_sums_list_50)

plt.figure()
plt.xlabel("Dissimilarity score")
plt.ylabel("Frequency")
```

```
plt.hist(genuine_50, alpha=0.5, label='Genuine', color='blue')
plt.hist(impostor_50, alpha=0.5, label='Impostor', color='red')
plt.hist(genuine_selfies, alpha=0.5, label='Genuine Selfies', color='lime')
plt.legend(loc='upper right')
```

Out[22]: <matplotlib.legend.Legend at 0x12711b730>



[5 points] Comment on the accuracy of the face matcher.

Before incorporating my own face images, the matcher appeared to adequately match unseen faces with one another. However, the results after incorporating my own face images show that the matcher wasn't trained enough to output the low distance scores expected of genuine scores. These results are likely due to two reasons.

1. None of the training data included my face images. If some number of my own faces were part of the training dataset, the matcher may have been able to discover features that are more closely based on my own face, and thus match them better.
2. There was intra-class variance within the selfie set. The selfies themselves were taken under a variety of lighting environments, and the images, although cropped all to the same size, may not have been aligned in exactly the same way. This variance may explain the wide range of genuine scores seen in the histogram above.