Task 1: Corner Detection

1.2, 1.3

```
def conv2(img, conv_filter):
    # flip the filter
    f_siz_1, f_size_2 = conv_filter.shape
    conv_filter = conv_filter[range(f_siz_1 - 1, -1, -1),
:][:, range(f_siz_1 - 1, -1, -1)]
    pad = (conv_filter.shape[0] - 1) // 2
    result = np.zeros((img.shape))
    img = np.pad(img, pad_width=pad, mode='constant',
constant_values=0)
    filter_size = conv_filter.shape[0]
    for r in np.arange(img.shape[0] - filter_size + 1):
        for c in np.arange(img.shape[1] - filter_size +
1):
            curr_region = img[r:r + filter_size, c:c +
filter_size]
            curr_result = curr_region * conv_filter
            conv_sum = np.sum(curr_result)  # Summing the
result of multiplication.
            result[r, c] = conv_sum  # Saving the
summation in the convolution layer feature map.

    return result


def fspecial(shape=(3, 3), sigma=0.5):
    """
```

```
    Create gaussian filter based on specified parameter, shape and sigma.
    The 2DGaussian will be a function of distance from the center and the provided sigma.

    P ~ Norm(0, sigma)
    h[i, j] = P(d) where P(d) is the probability density function of the specified gaussian distribution,
    and d is distance from center (0,0) to the posiution i, j.

    The closer d is to 0, the higher h[i, j] is.
    The output matrix (kernel) will be normalized to have sum of 1
    """
    m, n = [(ss - 1.) / 2. for ss in shape]
    y, x = np.ogrid[-m:m + 1, -n:n + 1]
    h = np.exp(-(x * x + y * y) / (2. * sigma * sigma))
    h[h < np.finfo(h.dtype).eps * h.max()] = 0
    sumh = h.sum()
    if sumh != 0: # shouldn't this be if sumh ≠ 1?
        h /= sumh
    return h



# Parameters, add more if needed
sigma = 5
# thresh = 0.01
harris_kernel_shape = (5, 5)
k = 0.05

# Derivative masks
dx = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])
dy = dx.transpose()

try:
    bw = plt.imread('0.png')
except:
    bw = plt.imread('Task1/Harris-1.jpg')
    bw = cv2.cvtColor(bw, cv2.COLOR_RGB2GRAY)
```

```
57
58
59  bw = np.array(bw * 255, dtype=int)
60
61  # computer x and y derivatives of image
62  Ix = conv2(bw, dx)
63  Iy = conv2(bw, dy)
64
65  # gaussian filter
66  g = fspecial(
67      (
68          max(1, np.floor(3 * sigma) * 2 + 1),
69          max(1, np.floor(3 * sigma) * 2 + 1)
70      ),
71      sigma
72  )
73
74  Iy2 = conv2(np.power(Iy, 2), g)
75  Ix2 = conv2(np.power(Ix, 2), g)
76  Ixy = conv2(Ix * Iy, g)
77
78  ############################################################
    ############
79  # Task: Compute the Harris Cornerness
80  ############################################################
    ############
81
82  # if w(x, y) is (1, 1), don't need to apply convolution
83  if harris_kernel_shape == (1, 1):
84      lambda_1 = Ix2
85      lambda_2 = Iy2
86  # else, for each pixel position, we will add up Ix2 and
    Iy2 of each pixel that overlaps with the fitler
87  else:
88      kernel = np.ones(harris_kernel_shape)
89      lambda_1 = conv2(Ix2, kernel)
90      lambda_2 = conv2(Iy2, kernel)
91
92  # calculate determinant and trace of matrix M for each
    pixel in the image, and finally compute R
```

```
det_m = lambda_1 * lambda_2 - np.square(conv2(Ixy, kernel))
trace_m = lambda_1 + lambda_2
R = det_m - (k * np.square(trace_m))

############################################################
# Task: Perform non-maximum suppression and
#       thresholding, return the N corner points
#       as an Nx2 matrix of x and y coordinates
############################################################

def non_maximum_suppression(R):
    """
    Perform non-maximum suppression by finding pixels that have intensity greater than 8 neighbors.

    input:
        R: 2-d numpy array of corner response values
    returns:
        corners: list of tuples indicating corners ditected
        arr: 2-d numpy binary array where arr[i, j] == 1 indicates corner as detected by non-maximum suppression
    """

    corners = []
    for i in range(1, len(R)-1):
        for j in range(1, len(R[0])-1):
            # check if R[i,j] is a corner by comparing each pixel to 8 neighbors
            # collect corners
            if (
                (R[i,j] >= R[i+1,j]) & (R[i,j] >= R[i-1,j]) & (R[i,j] >= R[i,j+1]) & (R[i,j] >= R[i,j-1]) &
                (R[i,j] >= R[i+1,j+1]) & (R[i,j] >= R[i-1,j-1]) & (R[i,j] >= R[i-1,j+1]) & (R[i,j] >= R[i-1,j-1])
            ):
                corners += (i, j),

```

```
125     # create a binary array from corners array to
    visualize corners
126     arr = np.zeros(R.shape)
127     for corner in corners:
128         arr[corner[0], corner[1]] = 1
129     return corners, arr
130 
131 corners, Nx2 = non_maximum_suppression(R)
```

1.4

Fixed parameters

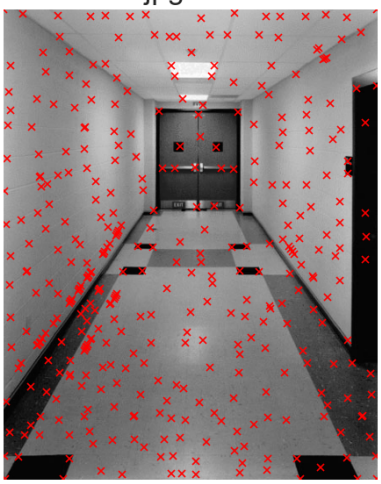
Fixed parameters: sigma=5, harris_kernel_width=5, k=0.05

Harris-1.jpg 239 corners

Harris-2.jpg 314 corners

Harris-3.jpg 6290 corners

Harris-4.jpg 374 corners



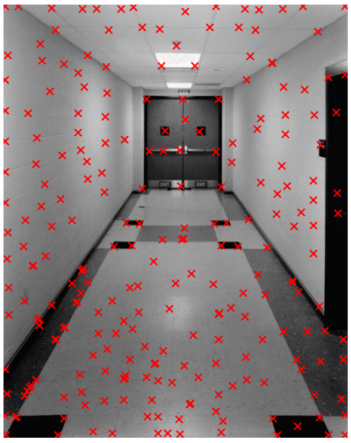
Variable parameters

Harris-1.jpg
sigma=5 kernel_size=5 k=0.05
239 corners

Harris-2.jpg
sigma=5 kernel_size=5 k=0.04
301 corners

Harris-3.jpg
sigma=9 kernel_size=11 k=0.05
684 corners

Harris-4.jpg
sigma=5 kernel_size=11 k=0.02
242 corners



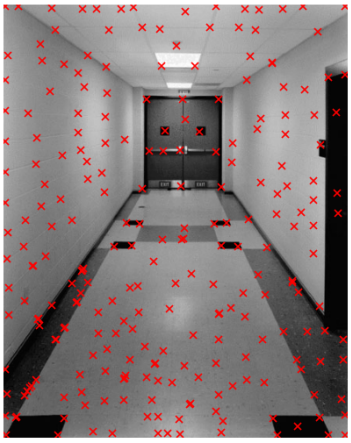
1.5. Compare results with `cv2.cornerHarris()`

Harris-1.jpg
sigma=5 kernel_size=5 k=0.05
239 corners

Harris-2.jpg
sigma=5 kernel_size=5 k=0.04
301 corners

Harris-3.jpg
sigma=9 kernel_size=11 k=0.05
684 corners

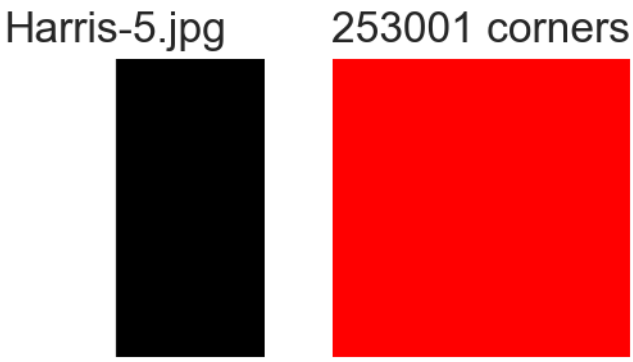
Harris-4.jpg
sigma=5 kernel_size=11 k=0.02
242 corners



Factors that affect the performance of Harris corner detection

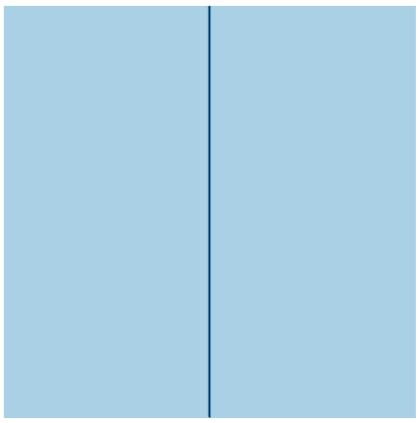
Harris corner detection is invariant to location, but not scale. As such, the performance will depend on the degree of gaussian smoothing determined by `sigma`, the kernel size of the sobel filter, the size of neighborhood considered when determining whether `img[i][j]` is a corner or not, and the k value in calculating the cornerness score R.

1.6



Analyse the results why we cannot get corners by discussing and visualising your corner response scores of the image.

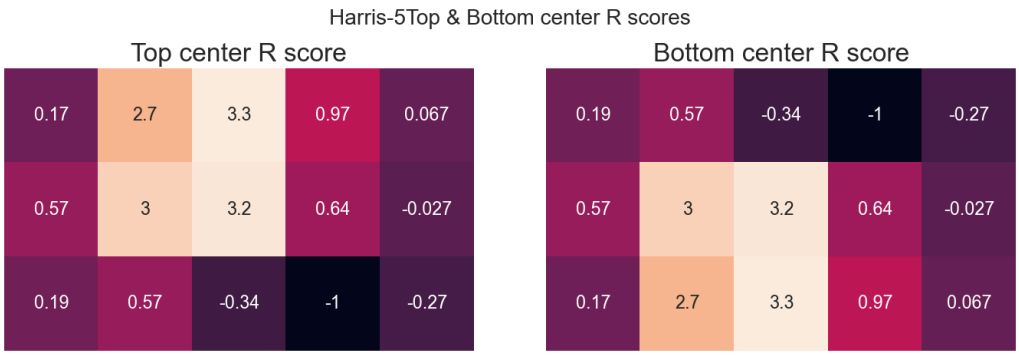
Cornerness R visualized using `plt.imshow()` with cmap `RdBu_r` (blue is lower).



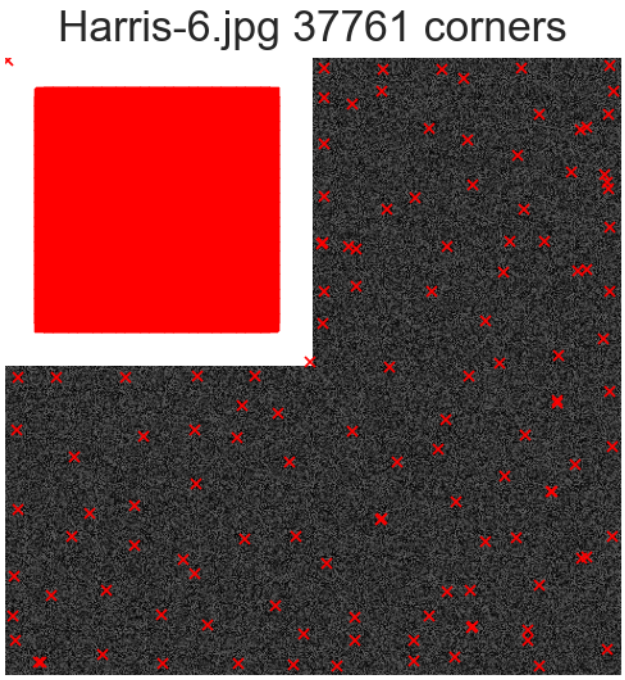
R value is approximately 0 for almost all pixel positions in the image. The only pixels with high R scores are at top and bottom left corners, and top and bottom center because the right half of the image is all 0s. Corners were detected on the left corners only because of the 0 padding of width 1 that's applied to the image in `conv2()` function. Corners were also detected in the top and bottom center because of the boundary between the white and black regions, in addition to the 0 padding. Aside from the influence from 0 padding, no corners were detected because there are no corners in the image.

```
Top left corner
[[1.95029123 2.09187119 0.11344908]
 [2.09187119 2.41787998 0.51097243]
 [0.11344908 0.51097243 0.17606731]]

Top right corner
[[0. 0. 0.]
```

```
 [0. 0. 0.]
 [0. 0. 0.]]

Bottom left corner
[[0.11344908 0.51097243 0.17606731]
 [2.09187119 2.41787998 0.51097243]
 [1.95029123 2.09187119 0.11344908]]

Bottom right corner
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]

Top center
[[ 0.17165154  2.65415093  3.26432971  0.96915069
0.06661764]
 [ 0.57056862  2.97199509  3.20967762  0.64287179
-0.0273949 ]
 [ 0.19054354  0.57293854 -0.33744827 -1.04884253
-0.27012009]]

Bottom center
[[ 0.19054354  0.57293854 -0.33744827 -1.04884253
-0.27012009]
 [ 0.57056862  2.97199509  3.20967762  0.64287179
-0.0273949 ]
 [ 0.17165154  2.65415093  3.26432971  0.96915069
0.06661764]]
```



1.7



Task 2: KMeans

2.1 KMeans

```
class KMeans:
    def __init__(self, X: np.ndarray, k: int):
        """initialize parameters and randomly create centroids in the domain of input data"""
        self.X = X
        self.k = k
        self.centroids = np.random.uniform(size=(self.k, self.X.shape[1]))
        self.pred = np.zeros(X.shape[0])

    @staticmethod
    def euclidean_distance(X, centroids):
        """calculate pair-wise euclidean distance"""
        k = centroids.shape[0]

        diff = X[np.newaxis,:,:] - centroids[:,np.newaxis,:]
```

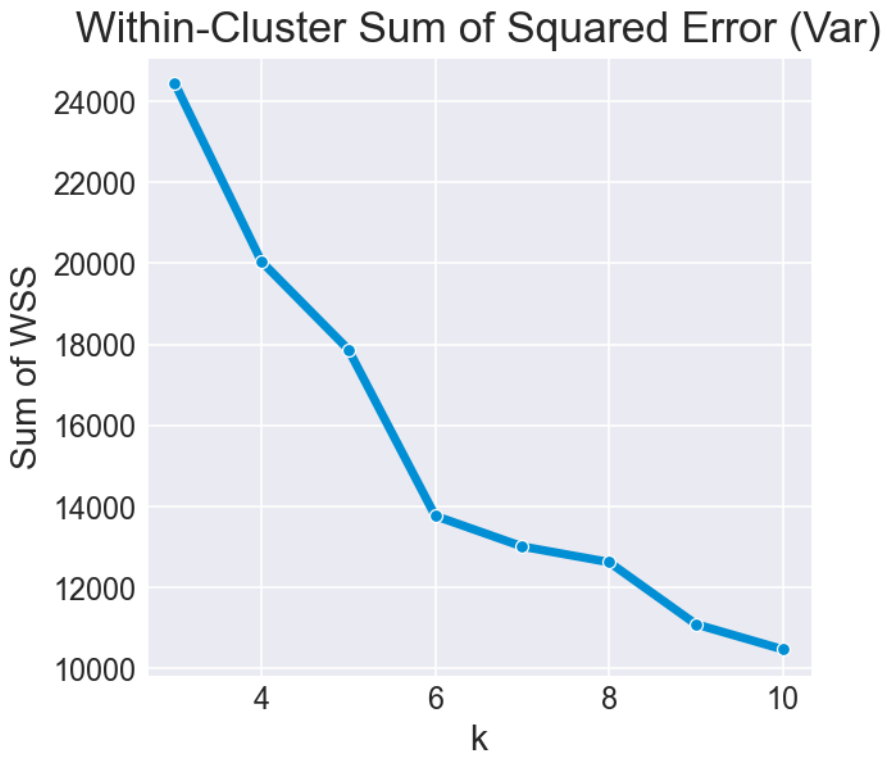
```
        assert diff.shape == (k, X.shape[0], X.shape[1])

        dist = np.sqrt(np.sum(diff**2, axis=-1))
        dist = dist.transpose()
        assert dist.shape == (X.shape[0], k)

        return dist

    def predict(self, verbose=False):
        i = 1
        start_time = time.time()

        while True:
            # calculate distance between every point and every centroid
            dist = KMeans.euclidean_distance(self.X, self.centroids)

            # assign each point to a class with nearest centroid
            pred = np.argmin(dist, axis=1)

            # calculate new centroid and wss for each cluster
            new_centroids = []
            wss = []
            for c in range(self.k):
                if len(self.X[pred==c]) == 0:
                    new_centroid = np.random.uniform(size=self.centroids.shape[1])
                else:
                    new_centroid = np.mean(self.X[pred==c], axis=0)
                new_centroids += new_centroid,

                # within-cluster sum of squared difference
                wss += np.sum((self.X[pred==c] - new_centroid) ** 2),

            new_centroids = np.array(new_centroids)
```

```
            assert self.centroids.shape ==
new_centroids.shape
            self.centroids = new_centroids
            self.wss = np.array(wss)

            if all(pred == self.pred):
                return time.time() - start_time
            else:
                if i % 5 == 0 and verbose:
                    elapsed = time.time() - start_time
                    print(f'iteration: {i:>3}\twss:
{self.wss.sum():^10.2f}\ttime in seconds:
{elapsed:>5.2f}')

                last_wss = self.wss
                self.pred = pred
                i += 1

def my_kmeans(data, start, end):
    """
    Run KMeans for various k values in the given range and
return the scores and time to converge
    """
    scores = []
    models = []
    conv_time = []

    assert start < end

    for k in range(start, end):
        kmeans = KMeans(data, k=k)
        t_elapsed = kmeans.predict()

        wss = kmeans.wss.sum()
        scores += wss,
        models += kmeans,
        conv_time += t_elapsed,
    return np.array(scores), conv_time
```

2.2

With pixel coordinates

Image 1: M&M



Using k=8

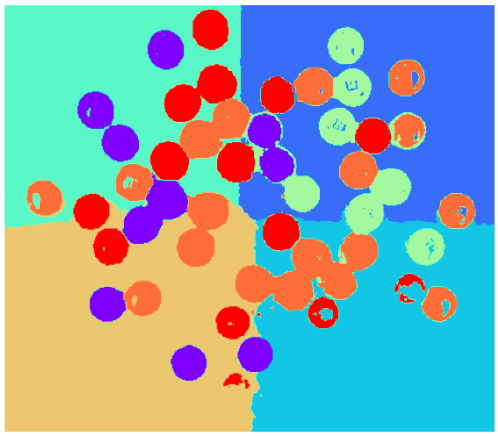
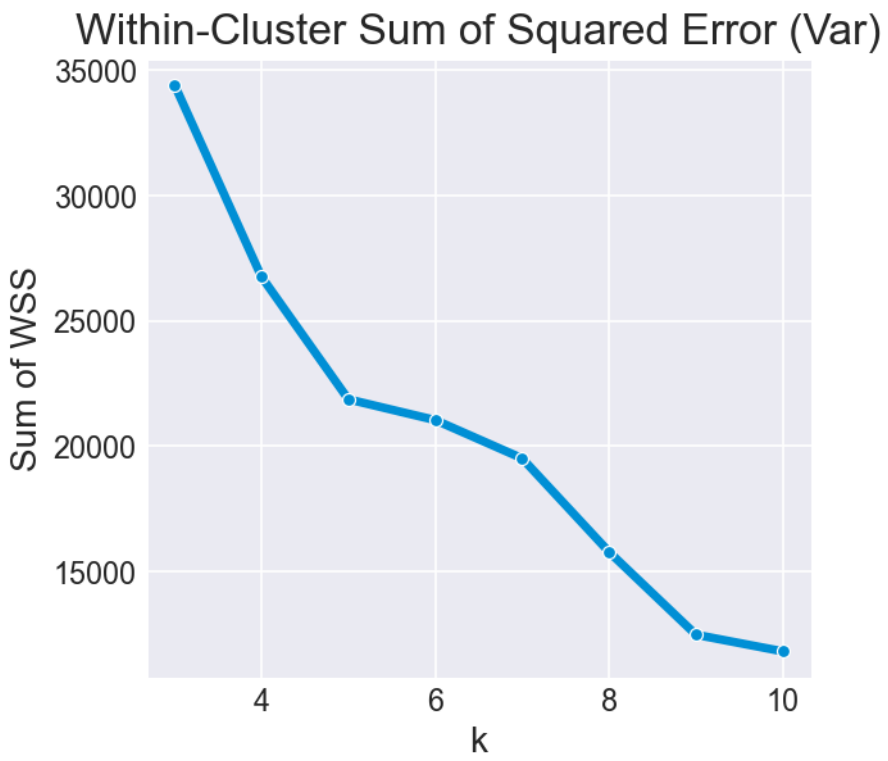
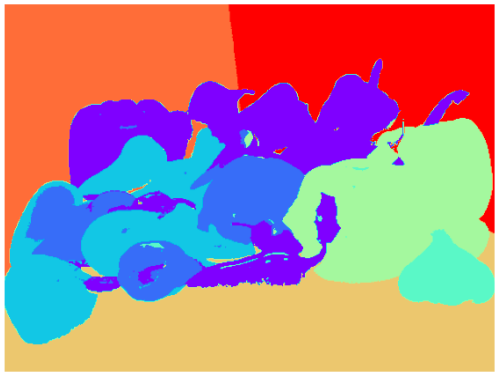


Image 2: Peppers

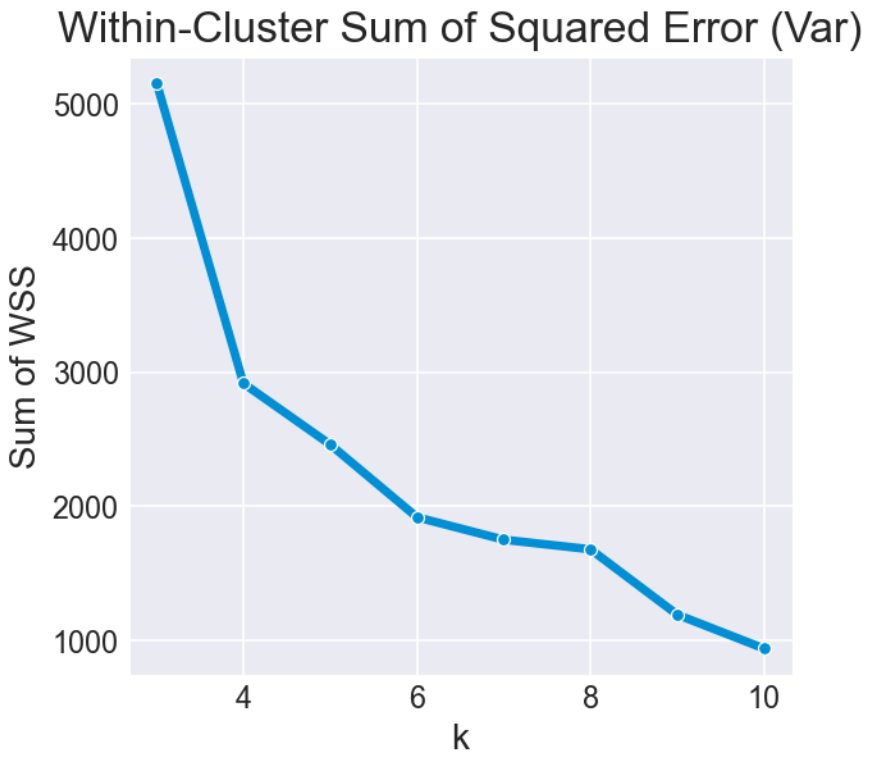


Using k=8



Without pixel coordinates

Image 1: M&M



Using k=7

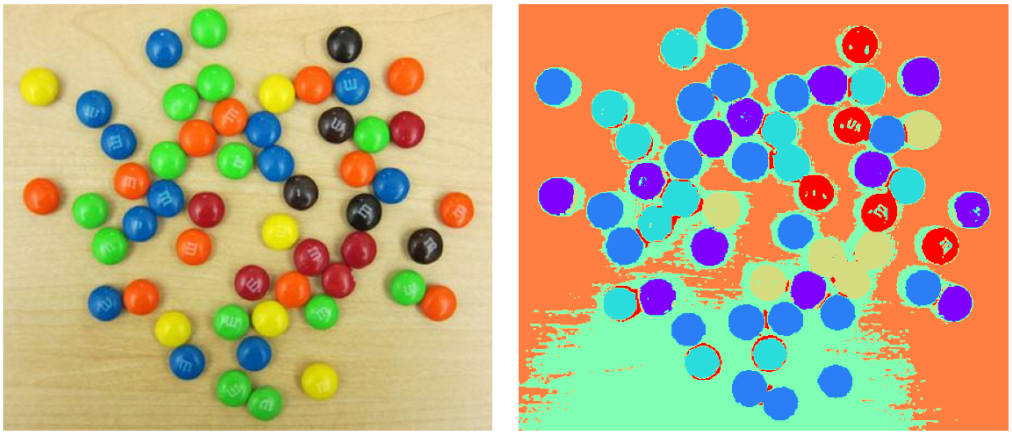
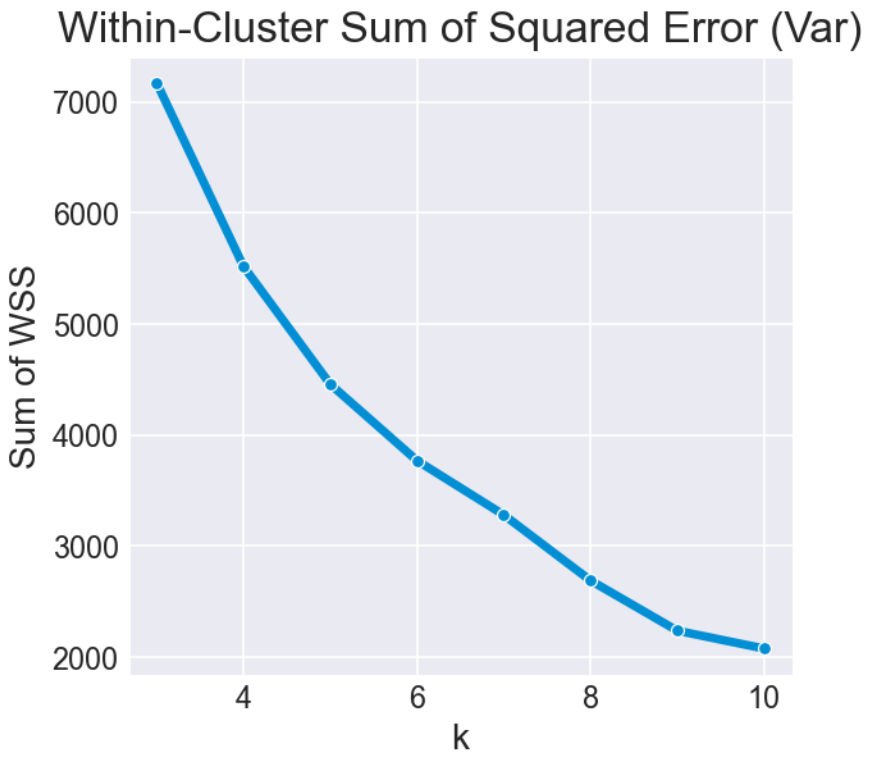
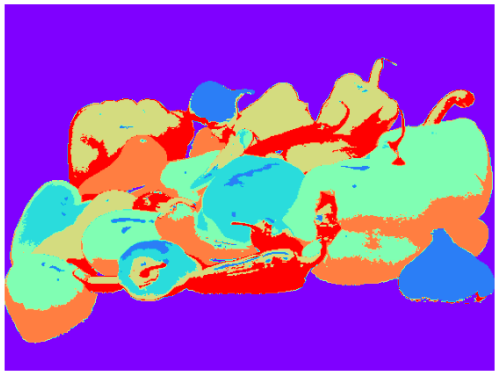


Image 2: Peppers



Using k=7



2.3 KMeans++

Key differences

The key difference between (Naive) KMeans and KMeans++ is in the initialization of the centroids.
While KMeans randomly initializes centroids in the input data space using uniform distribution, KMeans++ randomly picks one data point as the first centroid, and picks the rest of k-1 centroids by choosing the data point that is most distant from the closest centroid.

```
1 class KMeansPP:
```

```
    def __init__(self, X: np.ndarray, k: int):
        self.X = X
        self.k = k
        self.initialize_centroids()
        self.pred = np.zeros(self.X.shape[0])

    @staticmethod
    def euclidean_distance(X, centroids):
        """calculate pair-wise euclidean distance"""
        k = centroids.shape[0]

        diff = X[np.newaxis,:,:] -
centroids[:,np.newaxis,:]
        assert diff.shape == (k, X.shape[0], X.shape[1])

        dist = np.sqrt(np.sum(diff**2, axis=-1))
        dist = dist.transpose()
        assert dist.shape == (X.shape[0], k)

        return dist

    def initialize_centroids(self):
        X = copy.deepcopy(self.X)
        idx = np.random.randint(low=0, high=X.shape[0])
        centroid = X[idx]
        centroids = np.array([centroid])
        X = np.delete(X, idx, 0)

        for i in range(1, self.k):
            # calculate distance to the nearest centroid
for all points that are not used as a centroid
            dist = KMeansPP.euclidean_distance(X,
centroids)
            dist_to_nearest_centroid = np.min(dist, 1)

            # use the point furthest from the nearest
cluster as the next centroid
            idx = np.argmax(dist_to_nearest_centroid ** 2)
            centroid = X[idx]
            centroids = np.vstack((centroids, centroid))
            X = np.delete(X, idx, 0)
```

```
39
40          self.centroids = centroids
41
42      def predict(self, verbose=False):
43          i = 1
44          start_time = time.time()
45
46          while True:
47              # calculate distance between every point and
    every centroid
48              dist = KMeans.euclidean_distance(self.X,
    self.centroids)
49
50              # assign each point to a class with nearest
    centroid
51              pred = np.argmin(dist, axis=1)
52
53              # calculate new centroid and wss for each
    cluster
54              new_centroids = []
55              wss = []
56              for c in range(self.k):
57                  if len(self.X[pred==c]) == 0:
58                      new_centroid =
    np.random.uniform(size=self.centroids.shape[1])
59                  else:
60                      new_centroid =
    np.mean(self.X[pred==c], axis=0)
61                  new_centroids += new_centroid,
62
63                  # within-cluster sum of squared difference
64                  wss += np.sum((self.X[pred==c] -
    new_centroid) ** 2),
65
66              new_centroids = np.array(new_centroids)
67              assert self.centroids.shape ==
    new_centroids.shape
68              self.centroids = new_centroids
69              self.wss = np.array(wss)
70
71              if all(pred == self.pred):
```

```
                return time.time() - start_time
            else:
                if i % 5 == 0 and verbose:
                    elapsed = time.time() - start_time
                    print(f'iteration: {i:>3}\twss:
{self.wss.sum():^10.2f}\ttime in seconds:
{elapsed:>5.2f}')

                self.pred = pred
                i += 1

def my_kmeanspp(data, start, end):
    scores = []
    models = []
    conv_time = []

    assert start < end

    for k in range(start, end):
        kmeanspp = KMeansPP(data, k=k)
        t_elapsed = kmeanspp.predict()

        wss = kmeanspp.wss.sum()
        scores += wss,
        models += kmeanspp,
        conv_time += t_elapsed,
    return np.array(scores), conv_time
```

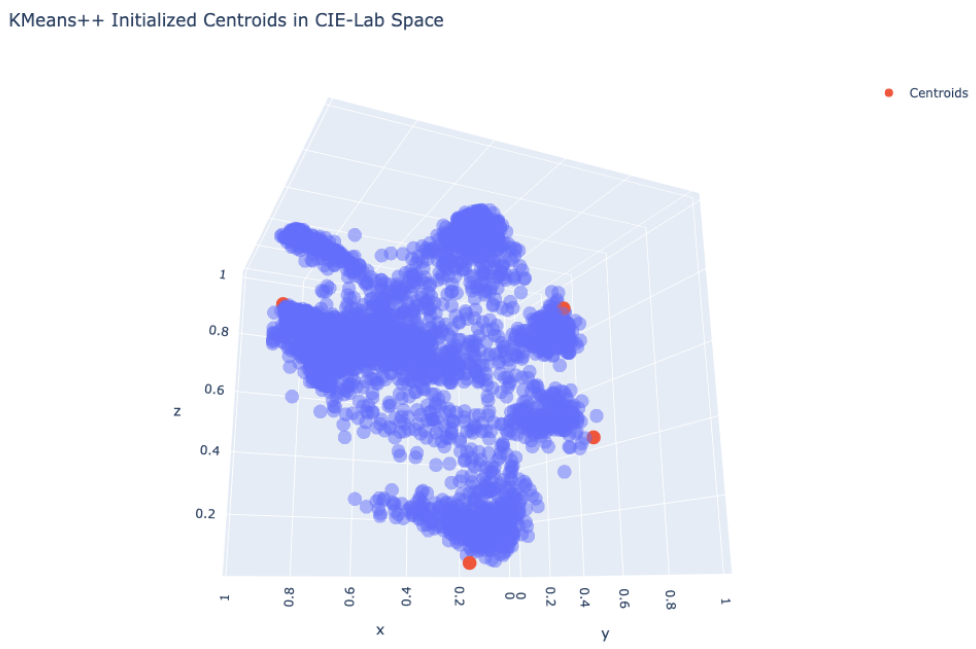
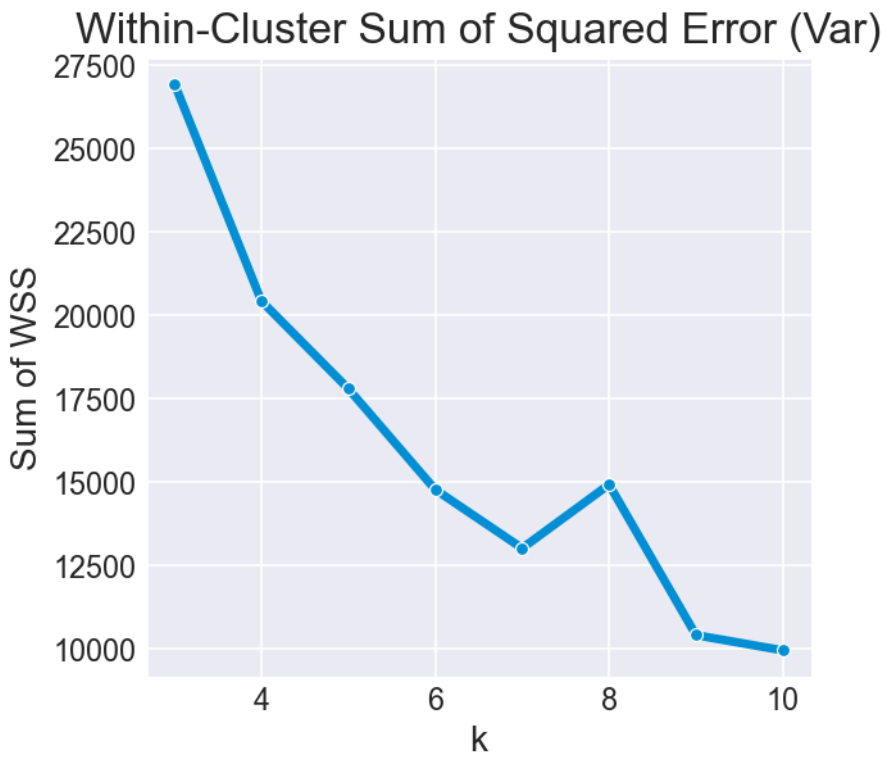


Image 1: M&M



Using k=7

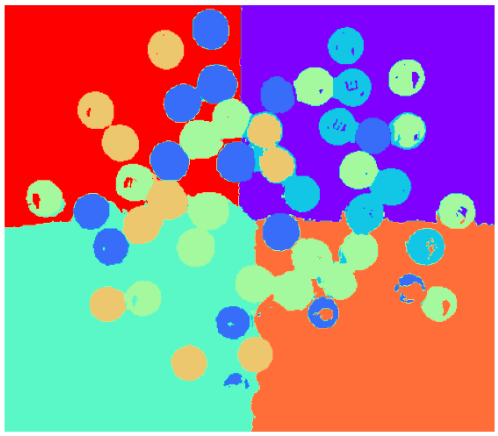
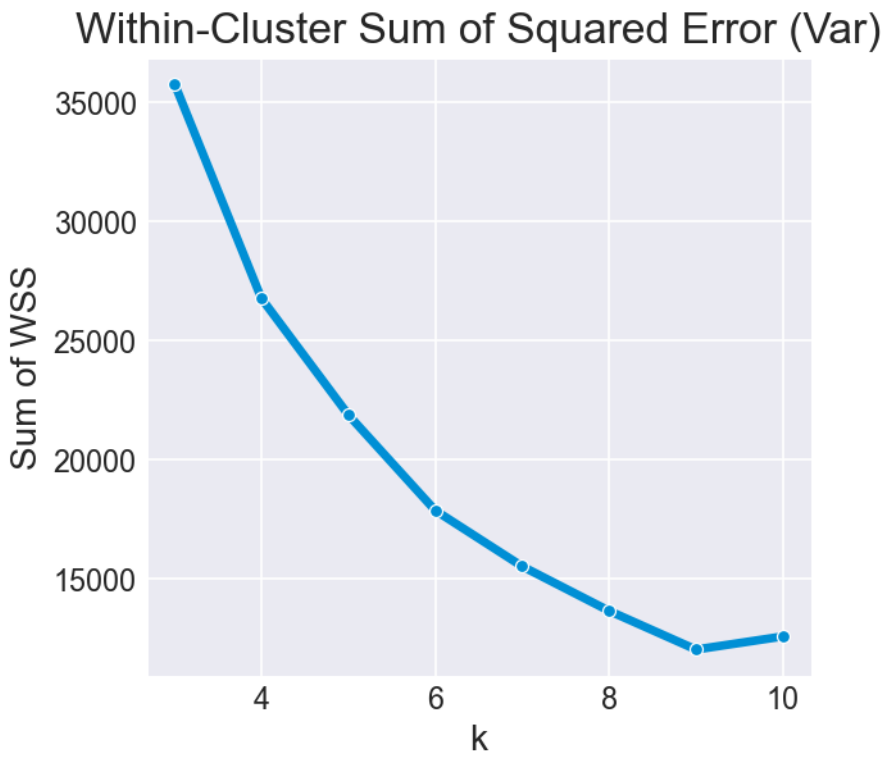
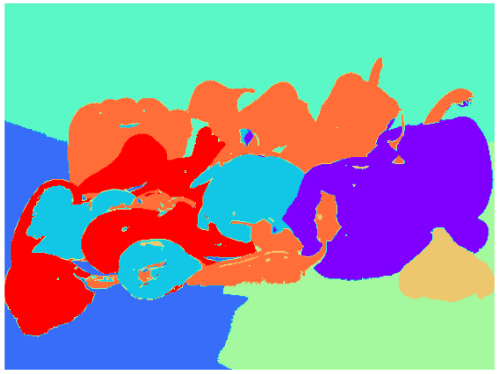


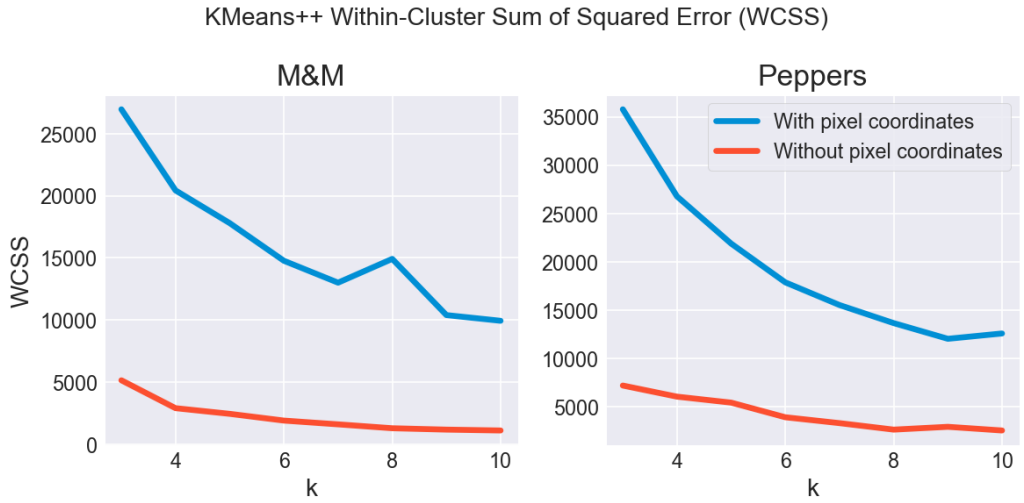
Image 2: Peppers

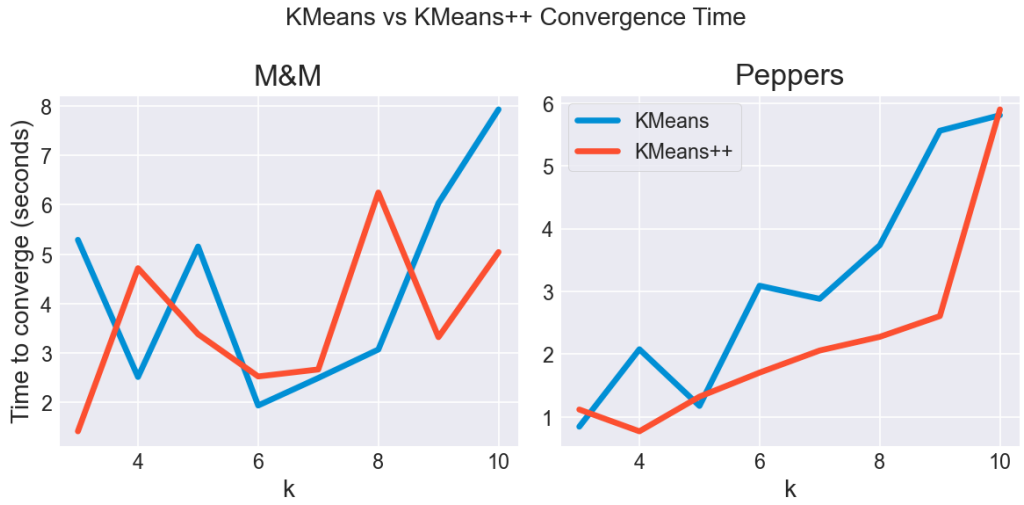


Using k=8



Model evaluation & comparison





Task 3: Eigenface

3.1 Explain why alignment is necessary for eigenface

Alignment is necessary because face detection using eigenface is not invariant to position or scale. Each pixel coordinate position i, j is essentially treated as a distinct random variable (that may or may not correlate with others). If the images are not aligned, it will be difficult to find correlation or direction of maximum variance.

3.2. Perform PCA and show mean face

```
import numpy as np
from numpy import linalg as LA

class PCA:
    def fit_transform(self, X: np.ndarray, n_components: int) → Tuple[np.ndarray, np.ndarray]:
        """
        input:
            X: n by m where n is the number of data points and m is the number of features
        returns:
            Xapprox: rank k approximation of X where k = n_components
        """
        self.X = X

        # create mean face by averaging the pixel intensity values across all our flattened face images
        self.mean = self.X.mean(0)
        self.X_centered = X - self.mean

        U, s, VT = LA.svd(X, full_matrices=False)
        self.svd_output = (U, s, VT)
        self.ev = self.explained_variance(s)

        S = np.diag(s)
        Xapprox = U[:,:n_components] @ S[:n_components,:n_components] @ VT[:n_components,:]
        return Xapprox + self.mean

    def explained_variance(self, s):
        """calculate explained variance"""
        ev = s ** 2 / np.sum(s ** 2)
        return ev

k = 15
model = PCA()
A_approx = model.fit_transform(A, n_components=k)
```

Mean face

Given the high dimensionality, directly performing eigen value decomposition of the covariance matrix would be slow. Find a faster way to compute eigen values and vectors, explain the reason.

Assumption: We don't care about eigenvalues that are approximately 0.

There are two ways (among other more complex methods) to compute eigenvalues and eigenvectors for non-square matrices with high dimensionality.

Let A be a real matrix $A \in \mathbb{R}^{n \times m}$ where $m > n$

1. Eigendecomposition

   If A is a real matrix, then AA^T and A^TA are symmetric square matrices.

   Matrix $AA^T = L \in \mathbb{R}^{n \times n}$ and $A^TA = L \in \mathbb{R}^{m \times m}$.

   Since the number of nonzero eigenvectors of AA^T is not greater than the number of eigenvectors of A^TA, we can perform eigendecomposition of $AA^T = L \in \mathbb{R}^{n \times n}$ to get eigenvectors and eigenvalues of $A^TA = L \in \mathbb{R}^{m \times m}$. Since $AA^T = L \in \mathbb{R}^{n \times n}$ is smaller (135 x 135) than (45045, 45045), it is much faster.

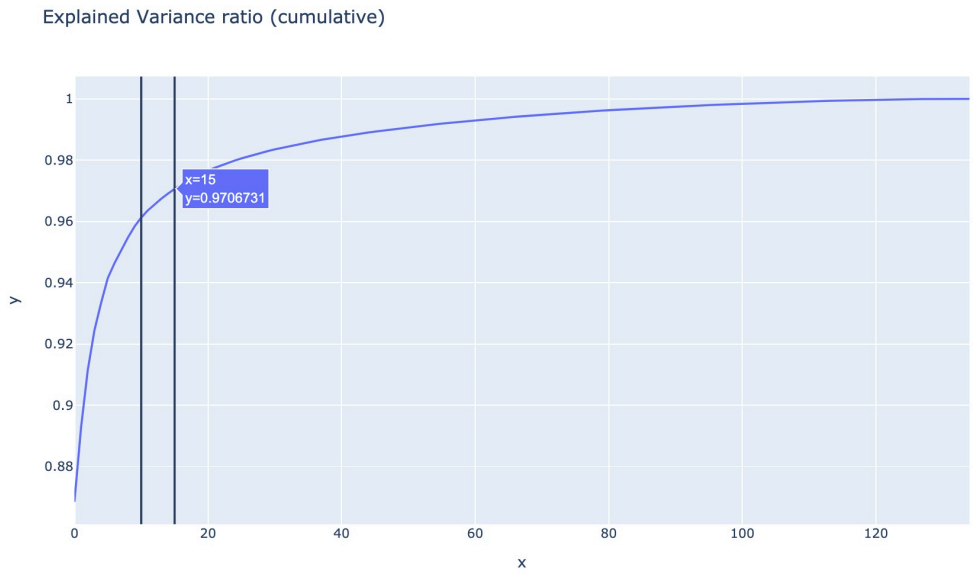
   $\therefore$ If v is an eigenvector with nonzero eigenvalue λ of A^TA, then Av is an eigenvector with the same eigenvalue of AA^T.

2. Singular Value Decomposition

Perform SVD directly on A, taking advantage of existing algorithms that can obtain eigenvalues and eigenvectors of the covariance matrix of A without having to compute $A^T A$.

3.3 Determine top k principal components and visualize top k eigenfaces

x axis represents the number of principal components, and y axis is the cumulative explained variance ratio.



First 15 principal components

3.4 Project test data to facespace spanned by first k eigenvectors and perform nearest neighbor search

```
idx = np.random.randint(0, test_images.shape[0])
x = test_images[idx]
x.shape
```

```
(45045,)
```

```
x_projected = x @ VT[:k,:].T
x_projected.shape
```

```
(15,)
```

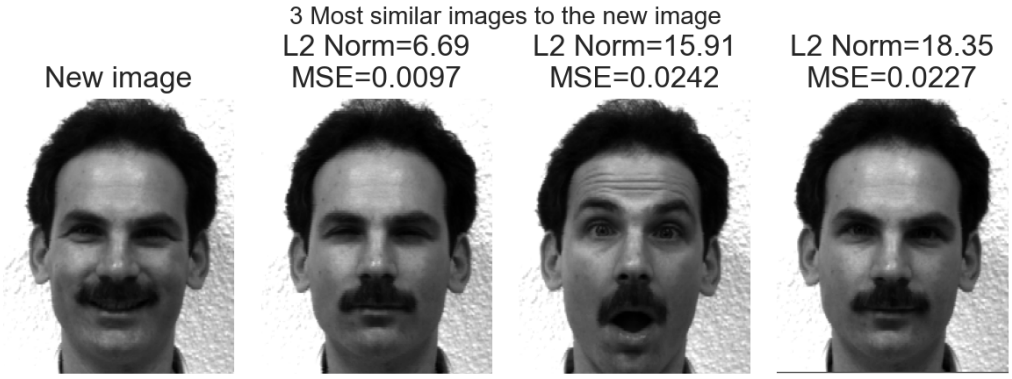
```
train_images_projected = A @ VT[:k,:].T
train_images_projected.shape
```

```
(135, 15)
```

```
dist = np.sqrt(np.sum((train_images_projected - x_projected) ** 2, axis=1))
dist.shape
```

```
(135,)
```

```
plt.figure(figsize=(10, 4))
plt.suptitle('3 Most similar images to the new image')
nearest_idx = np.argsort(dist)

plt.subplot(1, 4, 1)
plt.imshow(x.reshape(dim), cmap='gray')
plt.grid(False)
plt.xticks([])
plt.yticks([])
plt.title('New image')

for i in range(3):
    idx = nearest_idx[i]
    mse = np.mean((x - A[idx]) ** 2)
    plt.subplot(1, 4, i+2)
    plt.imshow(A[idx].reshape(dim), cmap='gray')
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    plt.title(f'L2 Norm={dist[idx]:.2f}\nMSE={mse:.4f}')

plt.tight_layout()

print('MSE range [0, 1]')
```



3.5 Repeat the above step with a photo of self

```
selfies = np.array(selfies)
selfies = selfies.reshape(10, -1)
selfies.shape
```

```
(10, 45045)
```

```
idx = np.arange(selfies.shape[0], dtype=int)
np.random.shuffle(idx)
idx
```

```
array([7, 2, 5, 6, 4, 9, 8, 1, 0, 3])
```

```
selfies_test = selfies[idx[0]]
selfies_train = selfies[idx[1:]]
```

```
selfies_test_projected = selfies_test @ VT[:k,:].T
print(selfies_test_projected.shape)

selfies_train_projected = selfies_train @ VT[:k,:].T
print(selfies_train_projected.shape)
```

```
(15,)
(9, 15)
```

```
dist = np.sqrt(np.sum((train_images_projected - selfies_test_projected) ** 2, axis=1))
nearest_idx = np.argsort(dist)
```

```
plt.figure(figsize=(10, 4))
plt.suptitle('3 Most similar images to new image')

plt.subplot(1, 4, 1)
plt.imshow(selfies_test.reshape(dim), cmap='gray')
plt.grid(False)
plt.xticks([])
plt.yticks([])
plt.title('New image')

for i in range(3):
    idx = nearest_idx[i]
    plt.subplot(1, 4, i+2)
    plt.imshow(A[idx].reshape(dim), cmap='gray')
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    plt.title(f'L2 Norm={dist[idx]:.2f}')

plt.tight_layout()
```

3.6 Repeat the above, with rest of selfies used as part of training set

```
train_images = A

combined_train_images_projected = np.vstack([train_images_projected, selfies_train_projected])
combined_train_images = np.vstack([train_images, selfies_train])
```

```
dist = np.sqrt(np.sum((combined_train_images_projected - selfies_test_projected) ** 2, axis=1))
nearest_idx = np.argsort(dist)
```

```
plt.figure(figsize=(10, 4))
plt.suptitle('3 Most similar images to new image')


plt.subplot(1, 4, 1)
plt.imshow(selfies_test.reshape(dim), cmap='gray')
plt.grid(False)
plt.xticks([])
plt.yticks([])
plt.title('New image')

for i in range(3):
    idx = nearest_idx[i]
    plt.subplot(1, 4, i+2)
    plt.imshow(combined_train_images[idx].reshape(dim), cmap='gray')
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    plt.title(f'L2 Norm={dist[idx]:.2f}')

plt.tight_layout()
```