4.1 PCA and Image Reconstruction [4 pts]

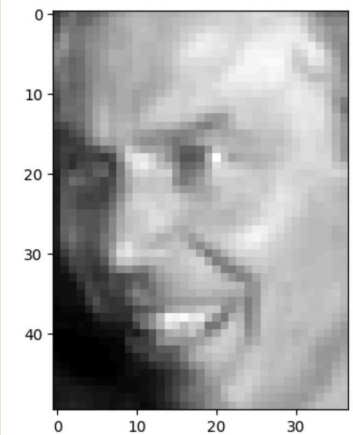
Before attempting automated facial recognition, you will investigate a general problem with images. That is, images are typically represented as thousands (in this project) to millions (more generally) of pixel values, and a high-dimensional vector of pixels must be reduced to a reasonably low-dimensional vector of features.

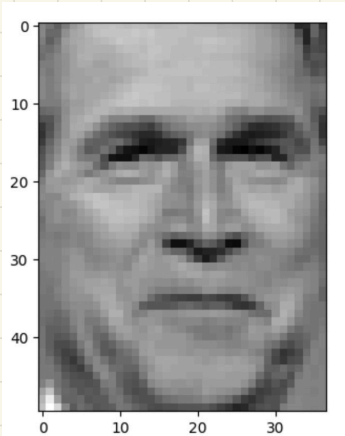
[1] http://vis-www.cs.umass.edu/lfw/

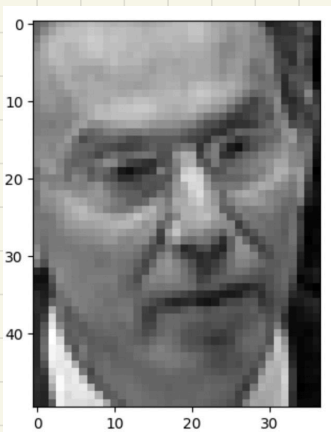
(a) As always, the first thing to do with any new dataset is to look at it. Use `get_lfw_data(...)` to get the LFW dataset with labels, and plot a couple of the input images using `show_image(...)`. Then compute the mean of all the images, and plot it. (Remember to include all requested images in your writeup.) Comment briefly on this "average" face.

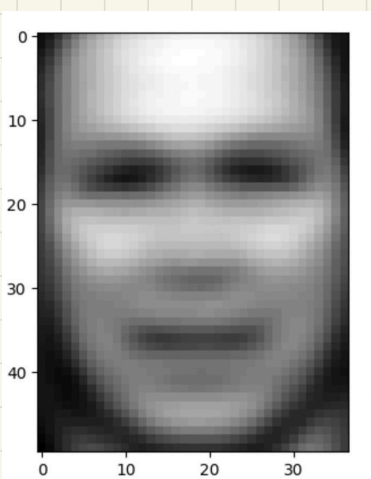
```
# part 1: explore LFW data set
# part a

X, y = util.get_lfw_data()
util.show_image(X[0])
util.show_image(X[1])
util.show_image(X[2])
util.show_image(np.mean(X, axis=0))
```

```
Total dataset size:
        num_samples: 1867
        num_features: 1850
        num_classes: 19
```









This is the mean of all the images. It makes sense that it would be forward facing and have distinguishable features such as eyes, nose, mouth and chin. However, it lacks uniqueness and definition in these features. Overall, it is quite ordinary.

(b) Perform PCA on the data using `util.PCA(...)`. This function returns a matrix `U` whose columns are the principal components, and a vector `mu` which is the mean of the data. If you want to look at a principal component (referred to in this setting as an eigenface), run `show_image(vec_to_image(v))`, where `v` is a column of the principal component matrix. (This function will scale vector `v` appropriately for image display.) Show the top twelve eigenfaces:

```
plot_gallery([vec_to_image(U[:,i]) for i in range(12)])
```

Comment briefly on your observations. Why do you think these are selected as the top eigenfaces?

```
# part b
## how the top twelve eigenfaces using PCA

U, mu = util.PCA(X)
util.plot_gallery([util.vec_to_image(U[:,i]) for i in range(12)])
```

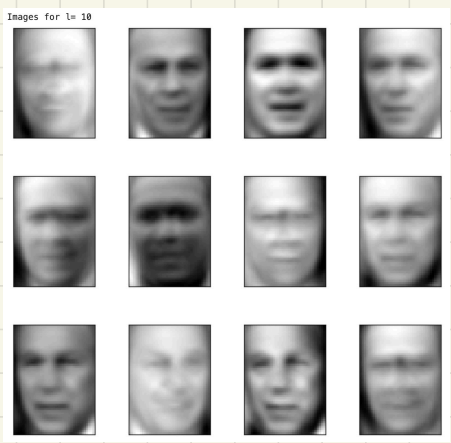
Each face has a distinct appearance. The features vary in many ways such as lighting and facial definition. It stands to reason that these images were chosen as the top eigenfaces because they are the most diverse and distinct.

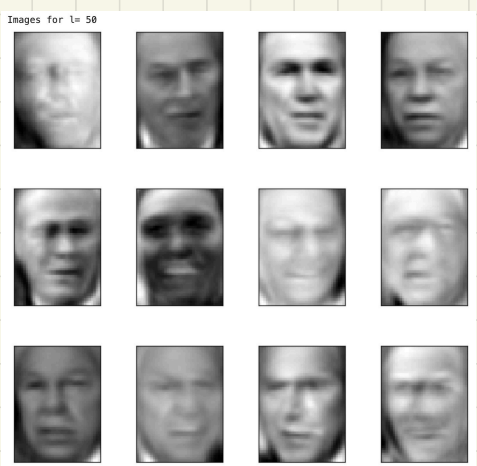
(c) Explore the effect of using more or fewer dimensions to represent images. Do this by:

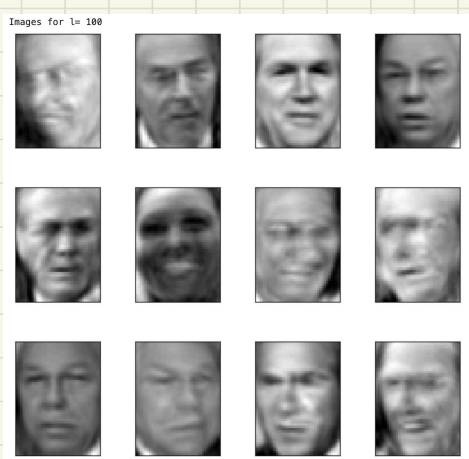
- Finding the principal components of the data
- Selecting a number l of components to use
- Reconstructing the images using only the first l principal components
- Visually comparing the images to the originals

To perform PCA, use `apply_PCA_from_Eig(...)` to project the original data into the lower-dimensional space, and then use `reconstruct_from_PCA(...)` to reconstruct high-dimensional images out of lower dimensional ones. Then, using `plotGallery(...)`, submit a gallery of the first 12 images in the dataset, reconstructed with l components, for $l = 1, 10, 50, 100, 500, 1288$. Comment briefly on the effectiveness of differing values of l with respect to facial recognition.

```
# part c
## compare the original image with the reconstructions

l_values = [1, 10, 50, 100, 500, 1288]
for l_value in l_values:
    print('Images for l=', l_value)
    Z, UI = util.apply_PCA_from_Eig(X, U, l_value, mu)
    X_rec = util.reconstruct_from_PCA(Z, UI, mu)
    title = f"Reconstructed for l = {l_value}"
    util.plot_gallery(X_rec, title=title)
```







The images become clearer and more distinct as l increases. This is because "l" corresponds to the number of components we keep from the original image. Low values of l such as $l=1$ all look similar to the mean of all images from part a and lack clarity and distinct features. This would be poor for facial recognition. $l=500$ and $l=1288$ are most easily distinguishable as people. However, there is little improvement between $l=500$ and $l=1288$, which may mean that there are diminishing returns for large values of l. Therefore, you want a value of l that is large but not overly large for facial recognition purposes.

4.2 K-Means and K-Medoids [16 pts]

Next, we will explore clustering algorithms in detail by applying them to a toy dataset. In particular, we will investigate k-means and k-medoids (a slight variation on k-means).

(a) In k-means, we attempt to find k cluster centers $\boldsymbol{\mu}_j \in \mathbb{R}^d$, $j \in \{1, \ldots, k\}$ and n cluster assignments $c^{(i)} \in \{1, \ldots, k\}$, $i \in \{1, \ldots, n\}$, such that the total distance between each data point and the nearest cluster center is minimized. In other words, we attempt to find $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$ and $c^{(1)}, \ldots, c^{(n)}$ that minimizes

$$J(\boldsymbol{c}, \boldsymbol{\mu}) = \sum_{i=1}^{n} ||\boldsymbol{x}^{(i)} - \boldsymbol{\mu}_{c^{(i)}}||^2.$$

To do so, we iterate between assigning $\boldsymbol{x}^{(i)}$ to the nearest cluster center $c^{(i)}$ and updating each cluster center $\boldsymbol{\mu}_j$ to the average of all points assigned to the j^{th} cluster.

Instead of holding the number of clusters k fixed, one can think of minimizing the objective function over $\boldsymbol{\mu}$, $\boldsymbol{c}$, *and* k. Show that this is a bad idea. Specifically, what is the minimum possible value of $J(\boldsymbol{c}, \boldsymbol{\mu}, k)$? What values of $\boldsymbol{c}$, $\boldsymbol{\mu}$, and k result in this value?

The minimum value of $J(c, \mu, k)$ is zero. To achieve this we set k equal to n, the number of data points, which assigns each point to its own cluster. Therefore, the distance between any point and its cluster is zero, making $J(c, \mu, k) = 0$.

Therefore, assuming x_i is a point in the dataset:

$c = i$

$\mu_i = x_i$

$k = n$

(b) To implement our clustering algorithms, we will use Python classes to help us define three abstract data types: `Point`, `Cluster`, and `ClusterSet`. Read through the documentation for these classes. (You will be using these classes later, so make sure you know what functionality each class provides!) Some of the class methods are already implemented, and other methods are described in comments. Implement all of the methods marked `TODO` in the `Cluster` and `ClusterSet` classes.

```
def centroid(self) :
    """
    Compute centroid of this cluster.

    Returns
    --------------------
        centroid -- Point, centroid of cluster
    """

    ### ========== TODO : START ========== ###
    # part 2b: implement
    # set the centroid label to any value (e.g. the most common label in this cluster
    total_attrs = np.array([p.attrs for p in self.points])
    attrs = np.mean(total_attrs, axis = 0)

    label = stats.mode([p.label for p in self.points])
    name = label
    centroid = Point(name, label, attrs)
    return centroid
    ### ========== TODO : END ========== ###
```

```
def medoid(self) :
    """
    Compute medoid of this cluster, that is, the point in this cluster
    that is closest to all other points in this cluster.

    Returns
    ----------------------
        medoid -- Point, medoid of this cluster
    """

    ### ========== TODO : START ========== ###
    # part 2b: implement
    distance_matrix = []
    for p in self.points:
      temp_matrix = []
      for p2 in self.points:
        temp_matrix.append(p.distance(p2))
      distance_matrix.append(temp_matrix)
    row_sums = np.sum(distance_matrix, axis=0).tolist()
    medoid = self.points[np.nanargmin(row_sums)]
    return medoid
    ### ========== TODO : END ========== ###
```

```
  def centroids(self) :
      """
      Return centroids of each cluster in this cluster set.

      Returns
      --------------------
          centroids -- list of Points, centroids of each cluster in this cluster set
      """

      ### ========== TODO : START ========== ###
      # part 2b: implement
      centroids = [m.centroid() for m in self.members]
      return centroids
      ### ========== TODO : END ========== ###


  def medoids(self) :
      """
      Return medoids of each cluster in this cluster set.

      Returns
      --------------------
          medoids -- list of Points, medoids of each cluster in this cluster set
      """

      ### ========== TODO : START ========== ###
      # part 2b: implement
      medoids = [m.medoid() for m in self.members]
      return medoids
      ### ========== TODO : END ========== ###
```

(c) Next, implement `random_init(...)` and `kMeans(...)` based on the specifications provided in the code.

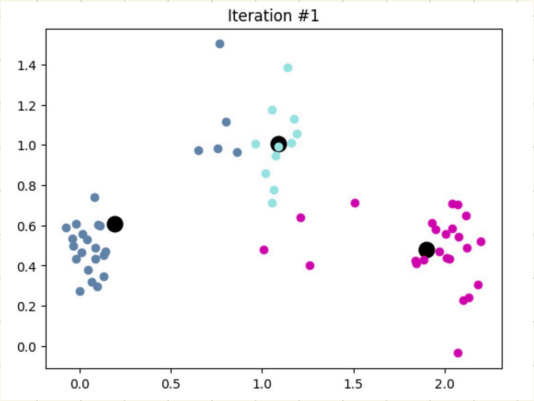
```
def random_init(points, k) :
    """
    Randomly select k unique elements from points to be initial cluster centers.

    Parameters
    -----------------------
        points         -- list of Points, dataset
        k              -- int, number of clusters

    Returns
    -----------------------
        initial_points -- list of k Points, initial cluster centers
    """
    ### ========== TODO : START ========== ###
    # part 2c: implement (hint: use np.random.choice)
    return np.random.choice(points, size=k, replace=False).tolist()
    ### ========== TODO : END ========== ###
```

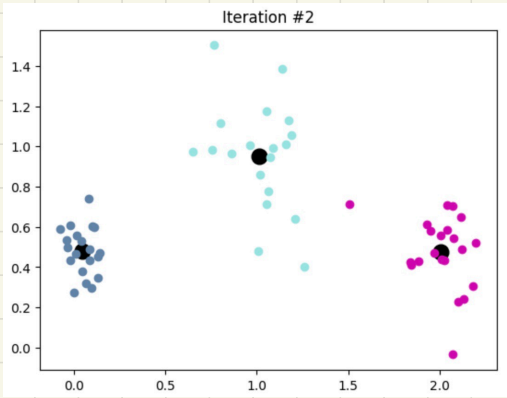
```
def kMeans(points, k, init='random', plot=False) :
    """
    Cluster points into k clusters using variations of k-means algorithm.

    Parameters
    -----------------------
        points  -- list of Points, dataset
        k       -- int, number of clusters
        init    -- string, method of initialization
                   allowable:
                       'cheat'  -- use cheat_init to initialize clusters
                       'random' -- use random_init to initialize clusters
        plot    -- bool, True to plot clusters with corresponding averages
                         for each iteration of algorithm

    Returns
    -----------------------
        k_clusters -- ClusterSet, k clusters
    """

    ### ========== TODO : START ========== ###
    # part 2c: implement
    # Hints:
    #   (1) On each iteration, keep track of the new cluster assignments
    #       in a separate data structure. Then use these assignments to create
    #       a new ClusterSet object and update the centroids.
    #   (2) Repeat until the clustering no longer changes.
    #   (3) To plot, use plot_clusters(...).
    return kAverages(points, k, ClusterSet.centroids, init, plot)
    ### ========== TODO : END ========== ###
```

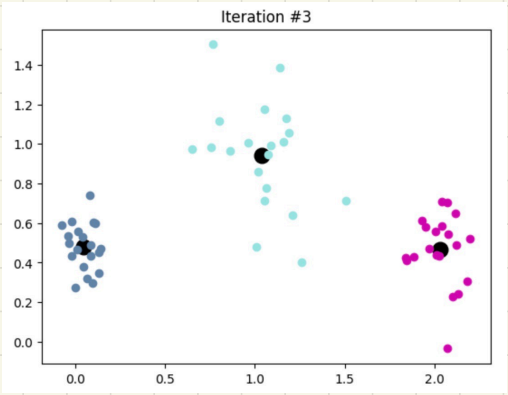
(d) Now test the performance of k-means on a toy dataset.

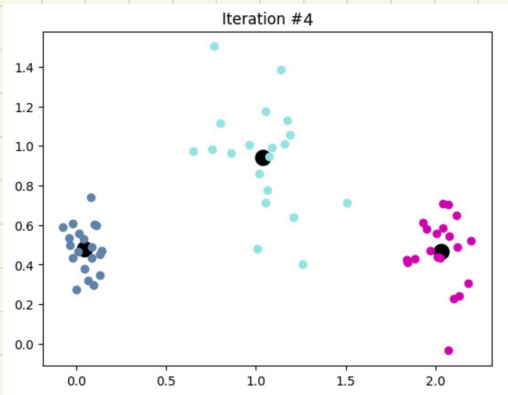
Use `generate_points_2d(...)` to generate three clusters each containing 20 points. (You can modify `generate_points_2d(...)` to test different inputs while debugging your code, but be sure to return to the initial implementation before creating any plots for submission.) You can plot the clusters for each iteration using the `plot_clusters(...)` function.

In your writeup, include plots for the k-means cluster assignments and corresponding cluster "centers" *for each iteration* when using random initialization and "k = 3".







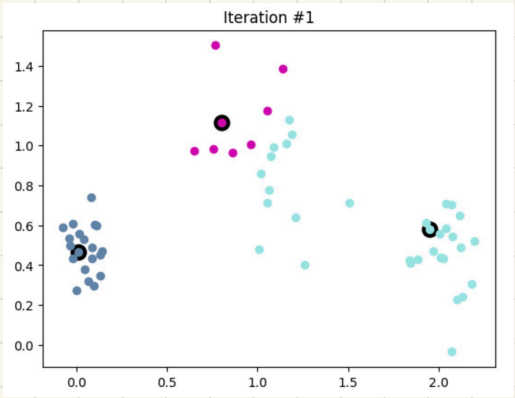


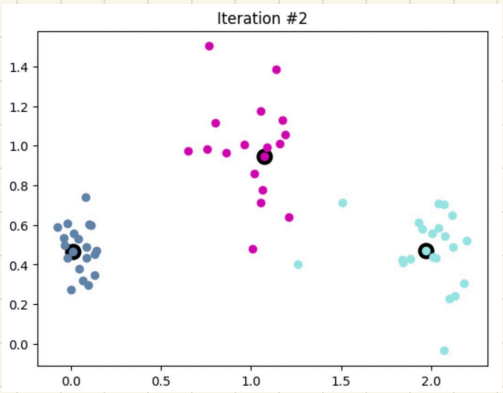
The 4th iteration of k-means is the same the 3rd. Therefore, k-means converges after 3 iterations.

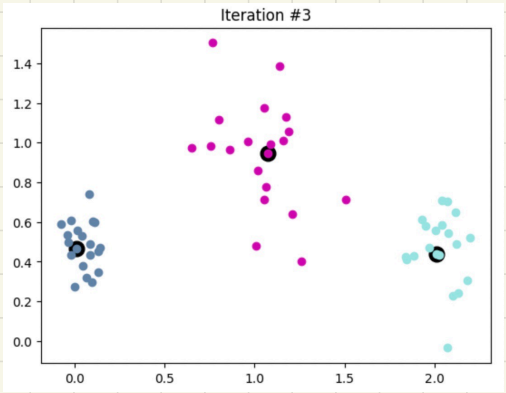
(e) Implement `kMedoids(...)` based on the provided specification.

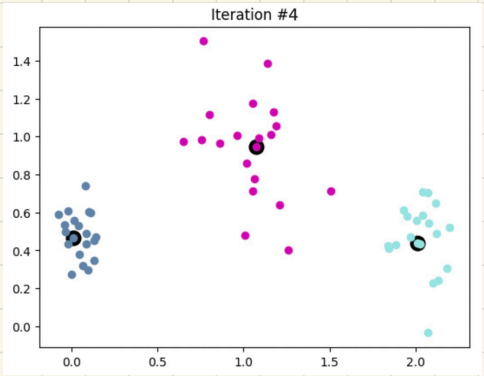
Hint: Since k-means and k-medoids are so similar, you may find it useful to refactor your code to use a helper function `kAverages(points, k, average, init='random', plot=False)`, where `average` is a method that determines how to calculate the average of points in a cluster (so it can take on values `ClusterSet.centroids` or `ClusterSet.medoids`).[2]

As before, include plots for k-medoids clustering *for each iteration* when using random initialization and "k = 3".









The 4th iteration of k Medoids is the same the 3rd. Therefore, k Medoids converges after 3 iterations.

```
def kAverages(points, k, average, init='random', plot=False) :
    """
    Cluster points into k clusters using variations of k-means algorithm.

    Parameters
    --------------------
        points  -- list of Points, dataset
        k       -- int, number of clusters
        average -- method of ClusterSet
                   determines how to calculate average of points in cluster
                   allowable: ClusterSet.centroids, ClusterSet.medoids
        init    -- string, method of initialization
                   allowable:
                       'cheat'  -- use cheat_init to initialize clusters
                       'random' -- use random_init to initialize clusters
        plot    -- bool, True to plot clusters with corresponding averages
                     for each iteration of algorithm

    Returns
    --------------------
        k_clusters -- ClusterSet, k clusters
    """
    ### ========== TODO : START ========== ###
    # part 2c,2d: implement
    groups = {}
    if init == 'cheat':
      initial_points = cheat_init(points)
    elif init == 'random':
      initial_points = random_init(points, k)

    for point in initial_points:
      groups[point] = []

    for point in points:
      current_lst = [point.distance(c) for c in initial_points]
      closest_centroid = initial_points[np.nanargmin(current_lst)]
      groups[closest_centroid].append(point)

    initial_clusterset = ClusterSet()
    for cluster in groups.values():
      initial_clusterset.add(Cluster(cluster))

    itera = 0
    while True:

      if plot == True:
        plot_clusters(initial_clusterset, "Iteration #" + str(itera + 1), average)
      itera += 1
```

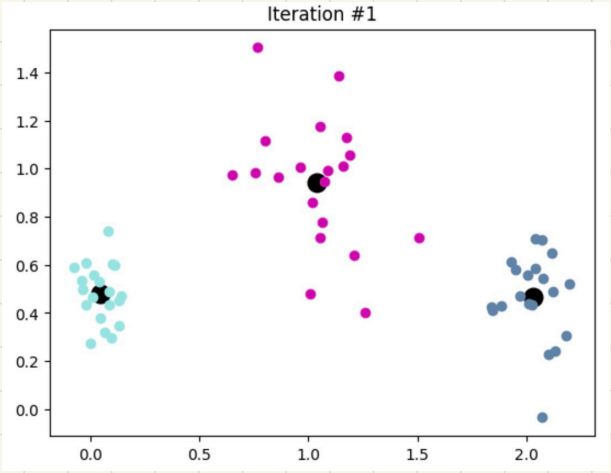
```
    current_assignments = {}
    current_initial_points = average(initial_clusterset)
    for point in current_initial_points:
      current_assignments[point] = []

    for point in points:
      current_lst = [point.distance(c) for c in tmp_initial_points]
      current_assignments[current_initial_points[np.nanargmin(current_lst)]].append(point)

    current_clusterset = ClusterSet()
    for cluster in current_assignments.values():
      current_clusterset.add(Cluster(cluster))

    if current_clusterset.equivalent(initial_clusterset):
      break
    else:
      initial_clusterset = current_clusterset
  if average == ClusterSet.centroids:
    print("kMeans: " + str(itera))
  else:
    print("kMedoids: " + str(itera))
  k_clusters = initial_clusterset

  return k_clusters

  ### ========== TODO : END ========== ###
```

```
def kMedoids(points, k, init='random', plot=False) :
    """
    Cluster points in k clusters using k-medoids clustering.
    See kMeans(...).
    """
    ### ========== TODO : START ========== ###
    # part 2e: implement
    return kAverages(points, k, ClusterSet.medoids, init, plot)

    ### ========== TODO : END ========== ###
```

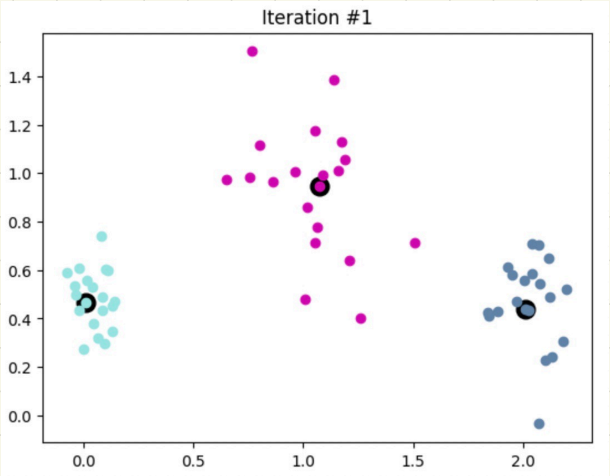
(f) Finally, we will explore the effect of initialization. Implement `cheat_init(...)`.

Now compare clustering by initializing using `cheat_init(...)`. Include plots for k-means and k-medoids using "k = 3" *for each iteration.*

kMeans:



k Medoids:



For kMeans and kMedoids using cheat_init, the first iteration and subsequent iterations were the same showing that they converge in one iteration.

```python
def cheat_init(points) :
    """
    Initialize clusters by cheating!

    Details
    - Let k be number of unique labels in dataset.
    - Group points into k clusters based on label (i.e. class) information.
    - Return medoid of each cluster as initial centers.

    Parameters
    --------------------
        points         -- list of Points, dataset

    Returns
    --------------------
        initial_points -- list of k Points, initial cluster centers
    """
    ### ========== TODO : START ========== ###
    # part 2f: implement
    initial_points = []
    labels = list(set([p.label for p in points]))
    for l in labels:
      c = Cluster([p for p in points if p.label == l])
      initial_points.append(c.medoid())

    return initial_points
    ### ========== TODO : END ========== ###
```

```python
### ========== TODO : START ========== ###
# part 2d-2f: cluster toy dataset
np.random.seed(1234) ## don't change the seed !!
points = generate_points_2d(20)
kMeans(points, 3, init='random', plot=True)
kMedoids(points, 3, init='random', plot=True)

print("Testing with cheat_init...")
kMeans(points, 3, init='cheat', plot=True)
kMedoids(points, 3, init='cheat', plot=True)

### ========== TODO : END ========== ###
```

4.3 Clustering Faces [12 pts]

Finally (!), we will apply clustering algorithms to the image data. To keep things simple, we will only consider data from four individuals. Make a new image dataset by selecting 40 images each from classes 4, 6, 13, and 16, then translate these images to (labeled) points: [3]

```
X1, y1 = util.limit_pics(X, y, [4, 6, 13, 16], 40)
points = build_face_image_points(X1, y1)
```

(a) Apply k-means and k-medoids to this new dataset with $k = 4$ and initializing the centroids randomly. Evaluate the performance of each clustering algorithm by computing the average cluster purity with `ClusterSet.score(...)`. As the performance of the algorithms can vary widely depending upon the initialization, run both clustering methods 10 times and report the average, minimum, and maximum performance along with runtime. How do the clustering methods compare in terms of clustering performance and runtime?

	average purity	min purity	max purity	average time	min time	max time
k-means						
k-medoids						

cluster purity with `ClusterSet.score(...)`. As the performance of the algorithms can vary widely depending upon the initialization, run both clustering methods 10 times and report the average, minimum, and maximum performance along with runtime. How do the clustering methods compare in terms of clustering performance and runtime?

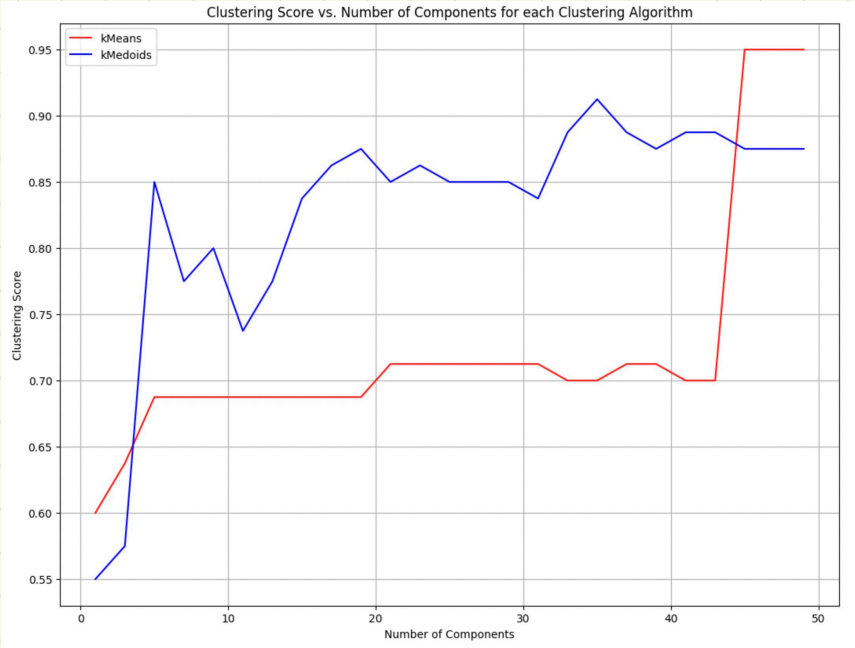
	average purity	min purity	max purity	average time	min time	max time
k-means	0.583	0.519	0.688	21.993	13.378	35.908
k-medoids	0.638	0.519	0.756	25.787	18.284	40.849

k-means and k-medoids are very similar in performance. k-means has a lower average runtime while k-medoids has a better average purity. k-means also has a lower min/max time while k-medoids has a higher min/max purity.

(b) Explore the effect of lower-dimensional representations on clustering performance. To do this, compute the principal components for the entire image dataset, then project the newly generated dataset into a lower dimension (varying the number of principal components), and compute the scores of each clustering algorithm.

So that we are only changing one thing at a time, use `init='cheat'` to generate the same initial set of clusters for k-means and k-medoids. For each value of l, the number of principal components, you will have to generate a new list of points using `build_face_image_points(...)`.)

Let $l = 1, 3, 5, \ldots, 49$. The number of clusters $K = 2$. Then, on a single plot, plot the clustering score versus the number of components for each clustering algorithm (be sure to label the algorithms). Discuss the results in a few sentences.



As the number of components increases, the overall trend is that the clustering score of both kMeans and kMedoids improves. This is logical as more components allows for more data to be stored and compared. It appears kMedoids has a greater clustering score when there are less than 2 components or more than 45. Between 2 to about 43 components kMedoids has a higher cluster score by a wide margin. This could indicate that kMedoids is better for facial recognition and having the cluster center as a data point performs better than a calculated mean.

(c) Experiment with the data to find a pair of individuals that clustering can discriminate very well and another pair that it finds very difficult (assume you have 40 images for each individual, projected to the top 50 principal components space). Describe your methodology (you may choose any of the clustering algorithms you implemented). Report these twp pairs of individuals (most similar pair and most discriminative pair) in your writeup (display each pair of images using `plot_representative_images`), and comment briefly on the results.

Most discriminative:


Most similar:


To determine the most discriminative and most similar faces, I put all combinations of 2 images from the dataset into pairs. I selected 40 images from each class and combined them into one image, and used kMedoids algorithm with 2 clusters due to there being 2 images. I chose this amount and algorithm due to the results found in part b, which suggests kMedoids generally performs better for facial recognition.

The images 0 and 6 were deemed most discriminative with a high clustering score of 0.9875. This makes sense as there are many different characteristics such as which direction they're facing, eye shape, facial lines, gender, and hair.

The images 1 and 8 were deemed most similar with a low clustering score of 0.5125. This also makes sense as they are facing the same direction and their features are very similar in size, position, and shape.