1 AdaBoost [5 pts]

In the lecture on ensemble methods, we said that in iteration t, AdaBoost is picking (h_t, β_t) that minimizes the objective:

$$\begin{aligned}(h_t^*(\boldsymbol{x}), \beta_t^*) &= \arg\min_{(h_t(\boldsymbol{x}),\beta_t)} \sum_n w_t(n) e^{-y_n \beta_t h_t(\boldsymbol{x}_n)} \\ &= \arg\min_{(h_t(\boldsymbol{x}),\beta_t)} (e^{\beta_t} - e^{-\beta_t}) \sum_n w_t(n) \mathbb{I}[y_n \neq h_t(\boldsymbol{x}_n)] \\ &\qquad + e^{-\beta_t} \sum_n w_t(n)\end{aligned}$$

We define the weighted misclassification error at time t, ϵ_t to be $\epsilon_t = \sum_n w_t(n)\mathbb{I}[y_n \neq h_t(\boldsymbol{x}_n)]$. Also the weights are normalized so that $\sum_n w_t(n) = 1$.

(a) Take the derivative of the above objective function with respect to β_t and set it to zero to solve for β_t and obtain the update for β_t.

1a $J = (e^{\beta_t} - e^{-\beta_t}) \sum_n w_t(n) I[y_n \neq h_t(x_n)] + e^{-\beta_t} \sum w_t(n)$ (ϵ_t; $\sum w_t(n) = 1$)

$J = (e^{\beta_t} - e^{-\beta_t})\epsilon_t + e^{-\beta_t}$

$\frac{dJ}{d\beta_t} = \epsilon_t(e^{\beta_t} + e^{-\beta_t}) - e^{-\beta_t} = 0$

$= \epsilon_t e^{\beta_t} + \epsilon_t e^{-\beta_t} - e^{-\beta_t} = 0$

$= e^{-\beta_t} - \epsilon_t e^{-\beta_t} = \epsilon_t e^{\beta_t}$

$= e^{-\beta_t}(1-\epsilon_t) = \epsilon_t e^{\beta_t}$

$= e^{-\beta_t} = e^{\beta_t}\left(\frac{\epsilon_t}{1-\epsilon_t}\right)$ Take log both sides

$= -\beta_t = \beta_t + \log\left(\frac{\epsilon_t}{1-\epsilon_t}\right)$

$\beta_t = -\frac{1}{2}\log\left(\frac{\epsilon_t}{1-\epsilon_t}\right)$

$\boxed{\beta_t = \frac{1}{2}\log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)}$

(b) Suppose the training set is linearly separable, and we use a hard-margin linear support vector machine (no slack) as a base classifier. In the first boosting iteration, what would the resulting β_1 be?

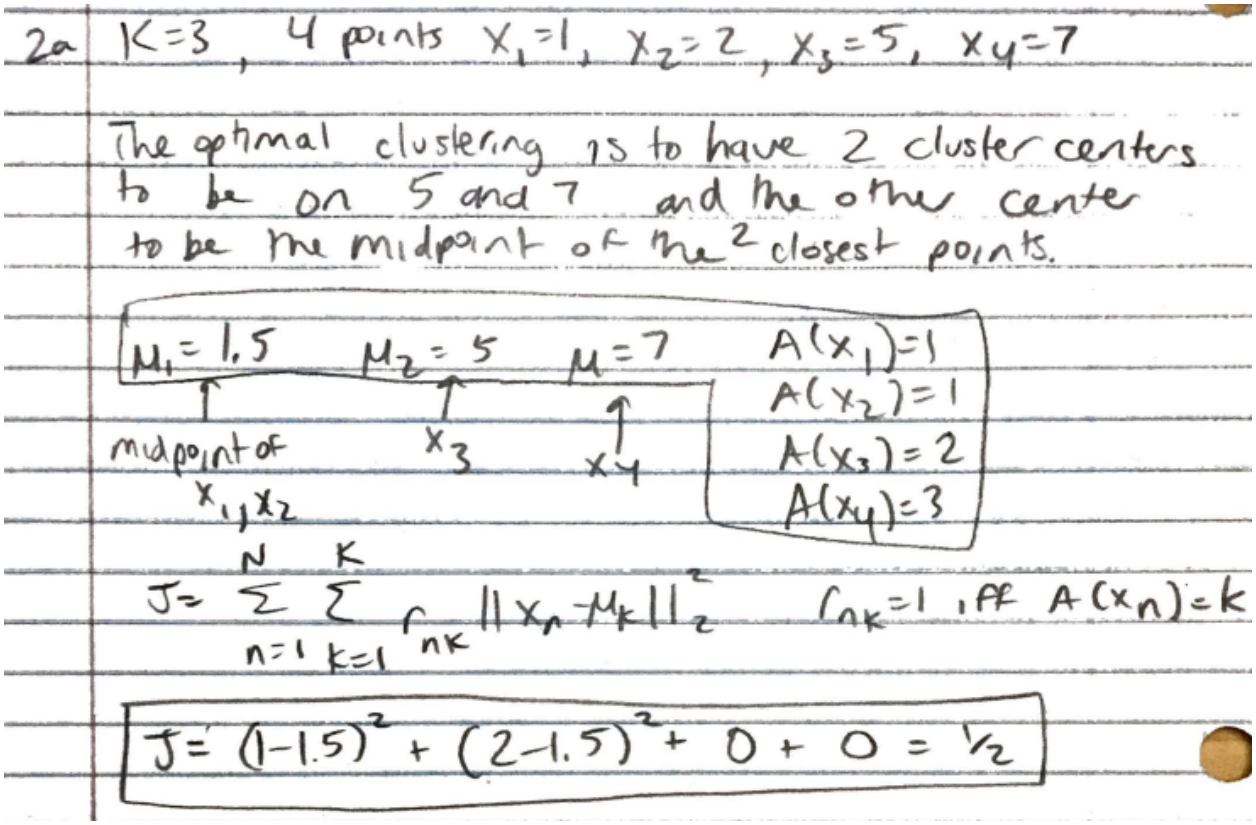
1b Since the data is linearly separable and we use hard-margin SVM, the error $\epsilon_t = 0$. As $\epsilon_1 \to 0$, $\beta_1 \to \infty$

Clearly, this classifier is not a weak classifier as it classifies all points correctly.

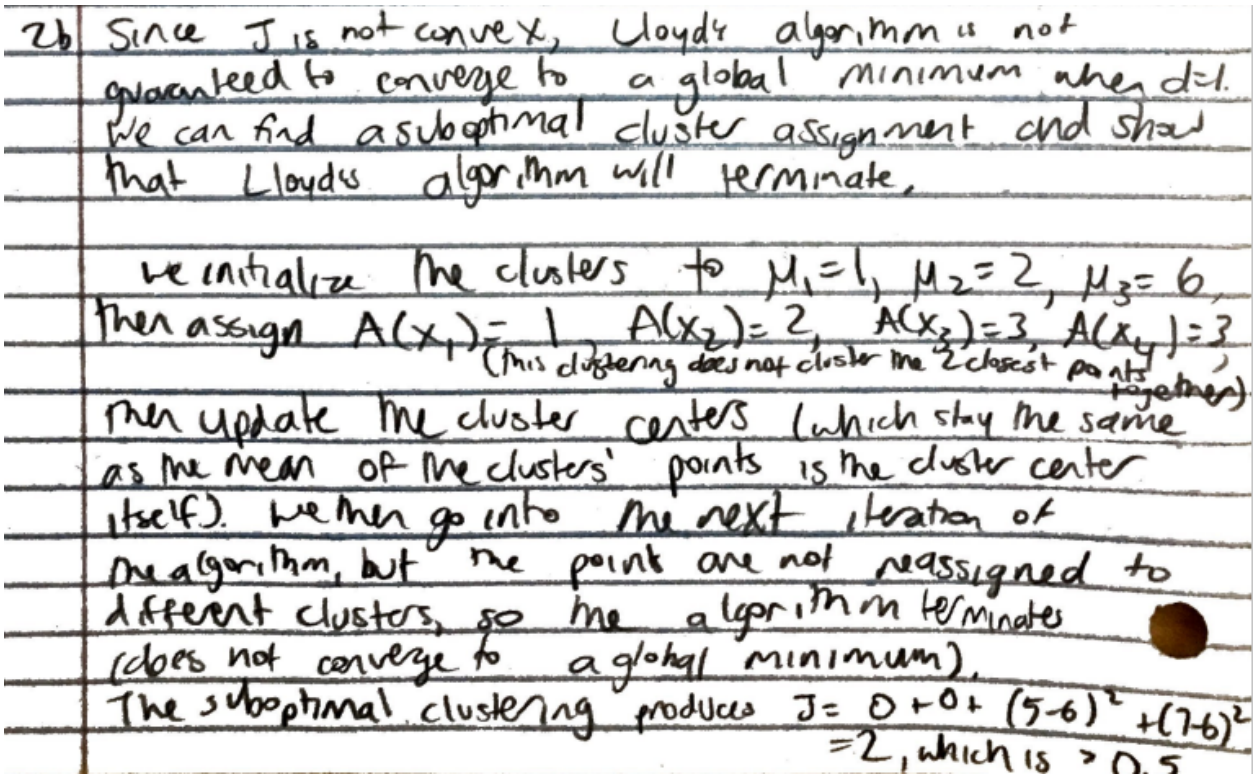
2 K-means for single dimensional data [5 pts]

In this problem, we will work through K-means for a single dimensional data.

(a) Consider the case where $K = 3$ and we have 4 data points $x_1 = 1, x_2 = 2, x_3 = 5, x_4 = 7$. What is the optimal clustering for this data ? What is the corresponding value of the objective ?



(b) One might be tempted to think that Lloyd's algorithm is guaranteed to converge to the global minimum when $d = 1$. Show that there exists a suboptimal cluster assignment (*i.e.*, initialization) for the data in the above part that Lloyd's algorithm will not be able to improve (to get full credit, you need to show the assignment, show why it is suboptimal *and* explain why it will not be improved).



The suboptimal cluster is $\mu_1 = 1$, $\mu_2 = 2$, $\mu_3 = 6$ (which is 2 clusters on x_1 and x_2 and 1 cluster in between x_3 and x_4). x_1 is assigned to μ_1, x_2 is assigned to μ_2, and x_3 and x_4 are assigned to μ_3. Lloyd's algorithm is unable to improve this cluster assignment since no points are reassigned to a better clustering, and the cluster centers are already the means of their points, so the algorithm terminates, to a local minimum. This assignment is suboptimal because the cost function is 2, which is greater than 0.5.

3 Gaussian Mixture Models [8 pts]

We would like to cluster data $\{x_1, \ldots, x_N\}$, $x_n \in \mathbb{R}^d$ using a Gaussian Mixture Model (GMM) with K mixture components. To do this, we need to estimate the parameters $\boldsymbol{\theta}$ of the GMM, *i.e.*, we need to set the values $\boldsymbol{\theta} = \{\omega_k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}_{k=1}^K$ where ω_k is the mixture weight associated with mixture component k, and $\boldsymbol{\mu}_k$ and Σ_k denote the mean and the covariance matrix of the Gaussian distribution associated with mixture component k.

If we knew which cluster each sample x_n belongs to (we had complete data), we showed in the lecture on Clustering that the log likelihood l is what we have below and we can compute the maximum likelihood estimate (MLE) of all the parameters.

$$\begin{aligned} l(\boldsymbol{\theta}) &= \sum_n \log p(\boldsymbol{x}_n, z_n) \\ &= \sum_k \sum_n \gamma_{nk} \log \omega_k + \sum_k \left\{ \sum_n \gamma_{nk} \log N(\boldsymbol{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right\} \end{aligned} \tag{1}$$

Since we do not have complete data, we use the EM algorithm. The EM algorithm works by iterating between setting each γ_{nk} to the posterior probability $p(z_n = k|\boldsymbol{x}_n)$ (step 1 on slide 26 of the lecture on Clustering) and then using γ_{nk} to find the value of $\boldsymbol{\theta}$ that maximizes l (step 2 on slide 26). We will now derive updates for one of the parameters, *i.e.*, $\boldsymbol{\mu}_j$ (the mean parameter associated with mixture component j).

(a) To maximize l, compute $\nabla_{\boldsymbol{\mu}_j} l(\boldsymbol{\theta})$: the gradient of $l(\boldsymbol{\theta})$ with respect to $\boldsymbol{\mu}_j$.

3a $\nabla_{\mu_j} l(\theta) = ?$ $\quad N(x_n | \mu_k, \Sigma_k) \xrightarrow{k=j} \mu=\mu_j,\ \sigma^2=\Sigma_j$

$$l(\theta) = \sum_k \sum_n \gamma_{nk} \log \omega_k + \sum_k \left\{ \sum_n \gamma_{nk} \log N(x_n | \mu_k, \Sigma_k) \right\}$$

$$\frac{dl(\theta)}{d\mu_j} = \sum_n \gamma_{nj} \frac{d}{d\mu_j} \left[\log \left(\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x_n - \mu_j}{\sigma}\right)^2} \right) \right]$$

$$= \sum_n \gamma_{nj} \frac{d}{d\mu_j} \left[-\log(\sigma\sqrt{2\pi}) + \log e^{\frac{-(x_n-\mu_j)^2}{2\sigma^2}} \right]$$

$$= \sum_n \gamma_{nj} \frac{d}{d\mu_j} \left[\frac{-(x_n-\mu_j)^2}{2\sigma^2} \right]$$

$$= \sum_n \gamma_{nj} \frac{(x_n - \mu_j)}{\sigma^2} \rightarrow \boxed{\frac{dl(\theta)}{d\mu_j} = \sum_n \gamma_{nj} \frac{(x_n - \mu_j)}{\Sigma_j}}$$

(b) Set the gradient to zero and solve for $\boldsymbol{\mu}_j$ to show that $\boldsymbol{\mu}_j = \frac{1}{\sum_n \gamma_{nj}} \sum_n \gamma_{nj} \boldsymbol{x}_n$.

3b. $$\frac{dl(\theta)}{d\mu_j} = \sum_n \gamma_{nj} \frac{(x_n - \mu_j)}{\Sigma_j} = 0$$

$$= \sum_n \gamma_{nj} (x_n - \mu_j) = 0$$

$$= \sum_n (\gamma_{nj} x_n - \gamma_{nj} \mu_j) = 0$$

$$= \sum_n \gamma_{nj} x_n - \mu_j \sum_n \gamma_{nj} = 0$$

$$\boxed{\mu_j = \frac{\sum_n \gamma_{nj} x_n}{\sum_n \gamma_{nj}}}$$

(c) Suppose that we are fitting a GMM to data using $K = 2$ components. We have $N = 5$ samples in our training data with $x_n, n \in \{1, \ldots, N\}$ equal to: $\{5, 15, 25, 30, 40\}$.

We use the EM algorithm to find the maximum likeihood estimates for the model parameters, which are the mixing weights for the two components, ω_1 and ω_2, and the means for the two components, μ_1 and μ_2. The standard deviations for the two components are fixed at 1. Suppose that at the end of step 1 of iteration 5 in the EM algorithm, the soft assignment γ_{nk} for the five data items are as shown in Table 1.

γ_1	γ_2
0.2	0.8
0.2	0.8
0.8	0.2
0.9	0.1
0.9	0.1

Table 1: Entry in row n and column k of the table corresponds to γ_{nk}

What are updated values for the parameters ω_1, ω_2, μ_1, and μ_2 at the end of step 2 of the EM algorithm?

3c $\omega_1 = \frac{\sum_n \gamma_{n1}}{\sum_k \sum_n \gamma_{nk}} = \frac{0.2+0.2+0.8+0.9+0.9}{1+1+1+1+1} = 3/5 = 0.6$ $\boxed{\omega_1 = 0.6}$

$N = 5$

$\omega_2 = \frac{\sum_n \gamma_{n2}}{\sum_k \sum_n \gamma_{nk}} = \frac{0.8+0.8+0.2+0.1+0.1}{5} = 2/5$ $\boxed{\omega_2 = 0.4}$

$\mu_1 = \frac{\sum_n \gamma_{n1} x_n}{\sum_n \gamma_{n1}} = \frac{1}{3}(0.2 \cdot 5 + 0.2 \cdot 15 + 0.8 \cdot 25 + 0.9 \cdot 30 + 0.9 \cdot 40)$

$\boxed{\mu_1 = 29}$

$\mu_2 = \frac{\sum_n \gamma_{n2} x_n}{\sum_n \gamma_{n2}} = \frac{1}{2}(0.8 \cdot 5 + 0.8 \cdot 15 + 0.2 \cdot 25 + 0.1 \cdot 30 + 0.1 \cdot 40)$

$\boxed{\mu_2 = 14}$

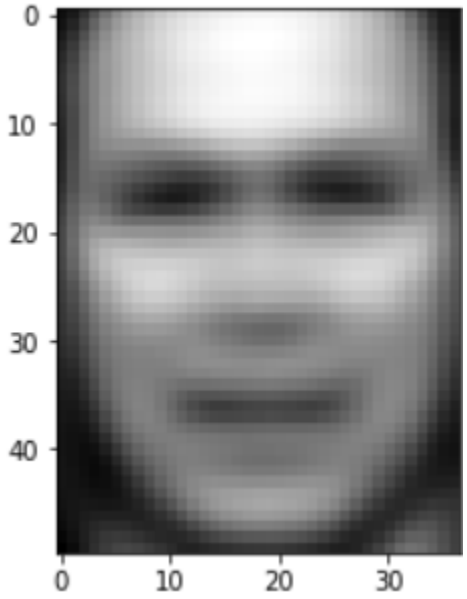
4 Implementation: Clustering and PCA [32 pts]

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.

(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.

Average image:



The average image looks like a mask with eye and mouth holes. It is facing forward and has no other distinctive features; it does not even have a nose shape. This makes sense, since an average face will not have specific distinctive features, but instead will be like a generic template for any face.

```
# part 1a: show a couple images, then show average image
X, y = get_lfw_data()
show_image(X[1866])
show_image(X[2])
show_image(np.mean(X, axis=0))
```

Random images (a couple):


0
10
20
30
40
0
10
20
30
0
10
20
30
40
0
10
20
30

(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 xrange(12)])
```

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

The eigenfaces all look different, with different lighting, contours, and unique qualities (defined nose shapes, presence of smile, facial hair, etc.). These are selected as the top 12 eigenfaces because they must be the 12 most unique faces in the dataset; these eigenfaces preserve almost all of the unique features while reducing the number of features.

```
# part 1b: perform PCA, show top 12 eigenfaces
U, mu = PCA(X)
plot_gallery([vec_to_image(U[:,i]) for i in range(12)])
```

(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.

We will revisit PCA in the last section of this project.

l = 1

I = 10

I = 50

I = 100

I = 500

---

l = 1288

As l increases, the faces become more defined (and look more like faces and less like a template of a face) as they get more distinct features, but we reduce dimensionality less. At l = 1, there is only 1 principal component, which makes all the eigenfaces look similar; as we increase l, the eigenfaces get more distinct features. However, as we increase l, it means we use more principal components, which means we are not reducing dimensionality by a large amount. We can see this clearly with the difference in the eigenfaces when we use l=500 and l=1288; the eigenfaces for each look very similar, while l=500 reduces dimensionality more. So, it seems that l=500 is a good compromise, since l=1288 hardly reduces dimensionality while only giving very marginally better results than when we use l=500.

```
# part 1c: explore using fewer or more dimensions to represent images
for l in [1, 10, 50, 100, 500, 1288]:
  Z, Ul = apply_PCA_from_Eig(X, U, l, mu)
  X_rec = reconstruct_from_PCA(Z, Ul, mu)
  plot_gallery(X_rec)
```

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?

Optimizing over the number of clusters k instead of keeping it fixed is bad because the minimum possible value of J is 0, since the algorithm would just choose to have as many clusters as there are data points (k = n), so each data point would be assigned to its own cluster, and the distance between each data point and its cluster would be 0. This is undesirable since the whole point of clustering is to find similarities between groups of points, so if we have k=n, then we would not be finding any similarities, and instead would just be classifying each point as its own group (no clustering). The values of **c** and **μ** can be any; as long as k=n, then J = 0.

(b) To implement our clustering algorithms, we will use Python classes to help us define three abstract data types: `Point`, `Cluster`, and `ClusterSet` (available in `cluster.py`). 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)
    labels = collections.defaultdict(lambda: 1)
    for p in self.points:
        labels[p.label] += 1
    most_common_label, frequency = None, -1
    for k, v in labels.items():
        if v > frequency:
            most_common_label, frequency = k, v
    centroid_attrs = np.mean(np.array([p.attrs for p in self.points]), axis = 0)
    centroid = Point('centroid', most_common_label, centroid_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
    lst = collections.defaultdict(lambda: 0)
    for p in self.points:
        for k in self.points:
            if k != p:
                lst[p] += p.distance(k)
    medoid = min(lst, key = lst.get)
    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
    return [cluster.centroid() for cluster in self.members]
    ### ========== 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
    return [cluster.medoid() for cluster in self.members]
    ### ========== 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, k, replace=False)
    ### ========== TODO : END ========== ###
```

kAverages implementation:

```
def kAverages(points, k, average, init='random', plot=True) :
    """
    Cluster points into k clusters using variations of averaging method (kMeans or kMedoids).

    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
    """
```

```
cur_centroids = random_init(points, k) if init == 'random' else cheat_init(points)
prev_clustering = None
i = 1
cur_clustering = ClusterSet()
while True: # repeat until clustering no longer changes (prev_clustering == cur_clustering)
    cur_clustering = ClusterSet()
    cluster_dict = collections.defaultdict(lambda: [])
    for p in points:
        min_dist, min_i = np.inf, -1
        for index in range(len(cur_centroids)):
            dist = p.distance(cur_centroids[index])
            if dist < min_dist:
                min_dist = dist
                min_i = index
        cluster_dict[min_i].append(p)
    for p in cluster_dict.values():
        cur_clustering.add(Cluster(p))
    if plot:
        if average == ClusterSet.centroids:
            s = "KMeans Clustering with " + init + " initialization, iteration " + str(i)
        else:
            s = "KMedoids Clustering with " + init + " initialization, iteration " + str(i)
        plot_clusters(cur_clustering, s, average)
    if prev_clustering is None or not cur_clustering.equivalent(prev_clustering):
        cur_centroids = average(cur_clustering)
        prev_clustering = cur_clustering
        i += 1
    else:
      break
k_clusters = cur_clustering
return k_clusters
```

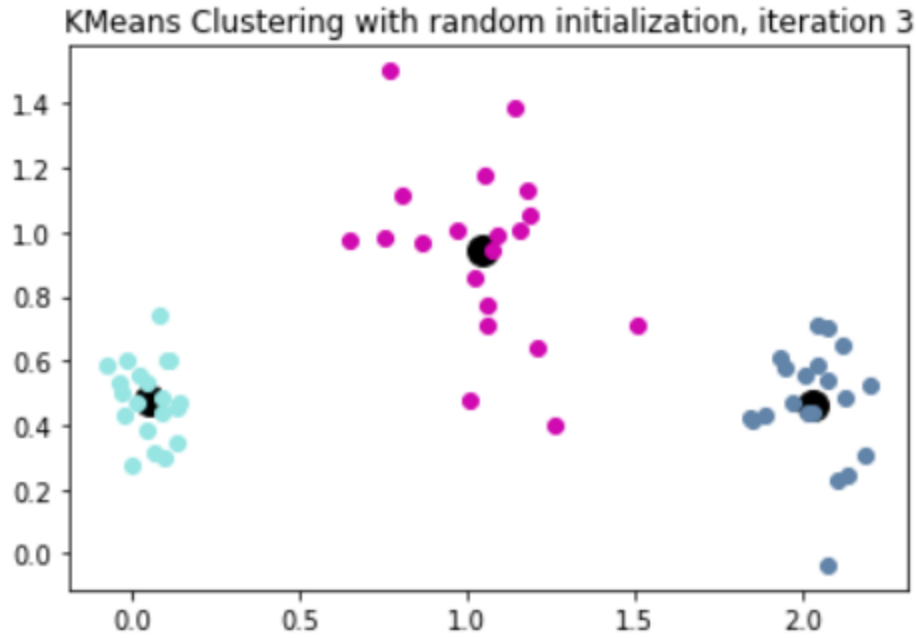
KMeans implementation:

```
### ========== TODO : START ========== ###
# part 2c: implement
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.



KMeans using random initialization takes 4 iterations to converge (it takes 3 iterations to reach the final assignment of points to clusters, and the 4th iteration does not reassign any clusters or points, which is why the 3rd and 4th plots are identical).

```
# part 2d-2f: cluster toy dataset
np.random.seed(1234)
points = generate_points_2d(20)
# part 2d: KMeans
kMeans_clusters = kMeans(points, 3, 'random', True)
```

(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=True)`, 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.

```
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 ========== ###
```

KMedoids using random initialization takes 3 iterations to converge (it takes 2 iterations to reach the final assignment of points to clusters, and the 3rd iteration does not reassign any clusters or points, which is why the 2nd and 3rd plots are identical).

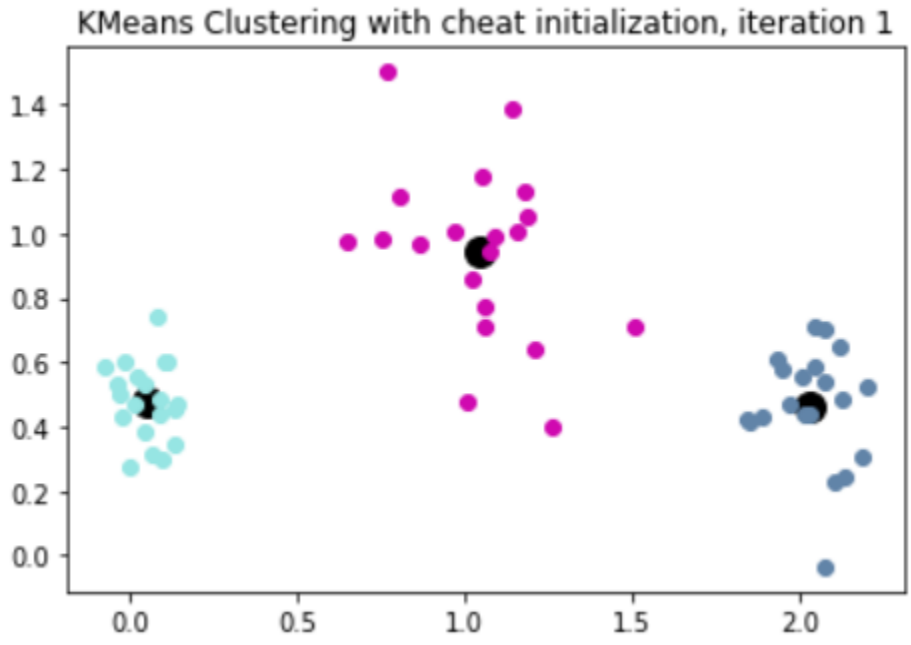
```
# part 2e: KMedoids
kMedoids_clusters = kMedoids(points, 3, 'random', True)
```

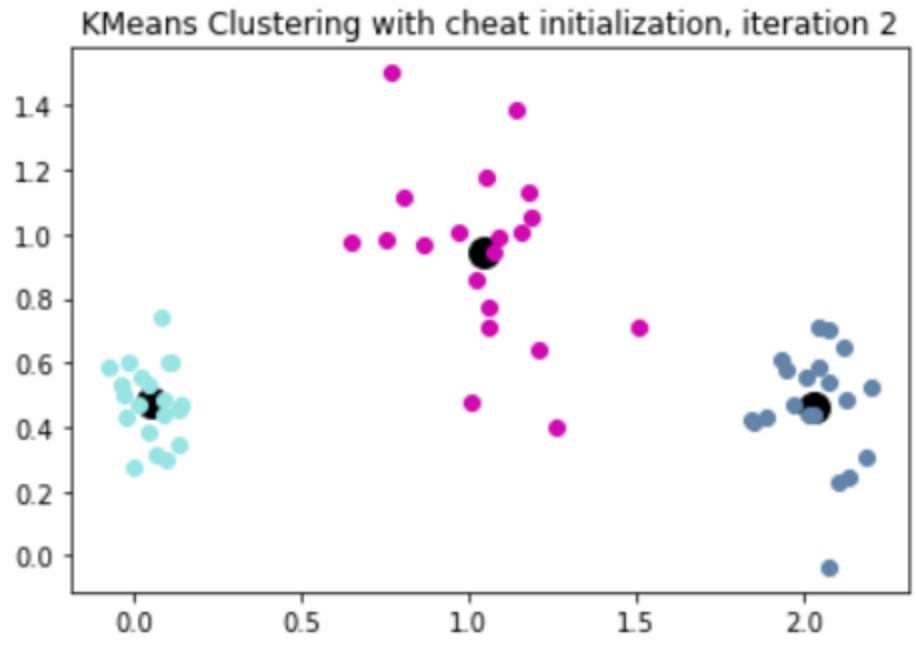
(f) Finally, we will explore the effect of initialization. Implement `cheat_init(...)`.

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

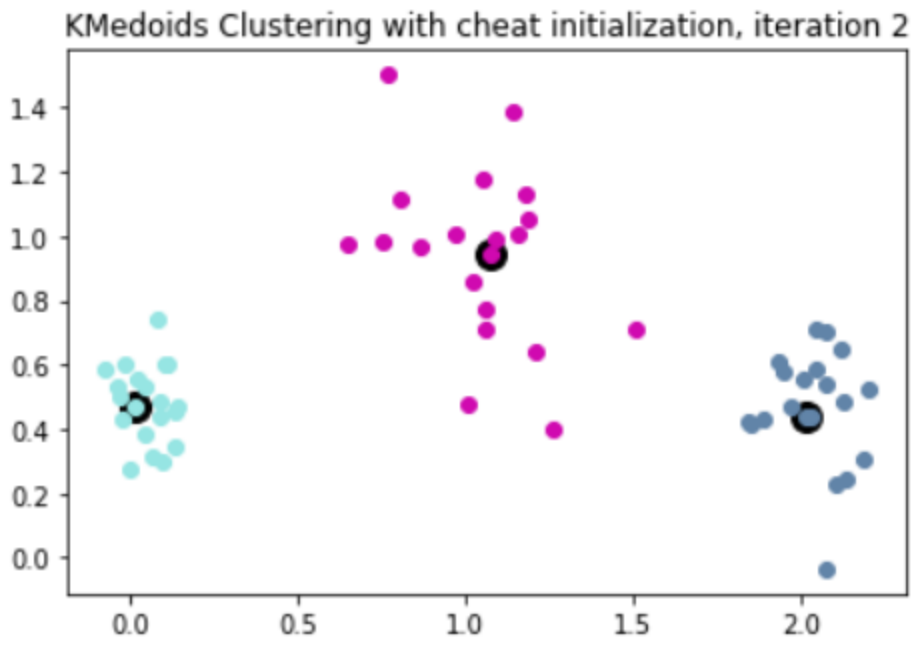
```
### ========== TODO : START ========== ###
# part 2f: implement
labels = collections.defaultdict(lambda: [])
for p in points:
    labels[p.label].append(p)
initial_points = [Cluster(v).medoid() for v in labels.values()]
return initial_points
### ========== TODO : END ========== ###
```

KMeans with cheat_init:





KMedoids with cheat_init:



Both KMeans and KMedoids find the optimal assignment of points to clusters in 1 iteration, and the second iteration for each does not update any clusters or reassign any points, so the second plot is identical to the first plot for each.
KMeans and KMedoids differ in the fact that the cluster for the KMedoids algorithm is always on a point, whereas for KMeans, the cluster does not have to be on a point.

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.

	average	min	max
k-means			
k-medoids			

How do the clustering methods compare in terms of clustering performance and runtime?

	average	min	max	Average time
k-means	**0.6175**	**0.55**	**0.775**	**0.165**
k-medoids	**0.6325**	**0.575**	**0.725**	**0.271**

K-medoids has slightly better average performance, with a higher min and lower max, than K-means. It also takes longer to run K-medoids.

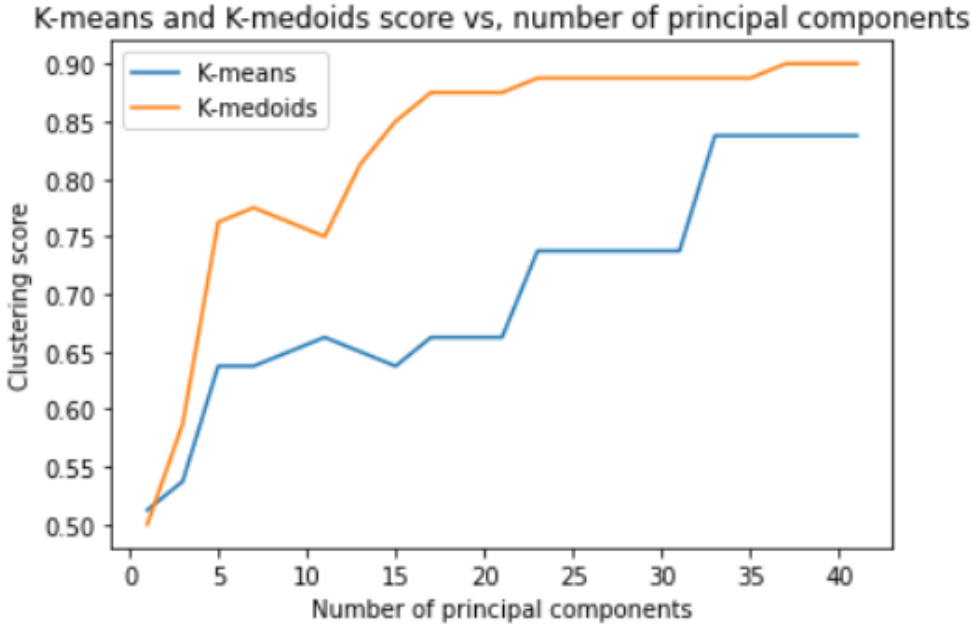
```
# part 3a: cluster faces
np.random.seed(1234)
X1, y1 = util.limit_pics(X, y, [4, 6, 13, 16], 40)
points = build_face_image_points(X1, y1)
kMeans_scores, kMedoids_scores = [], []
kMeans_times, kMedoids_times = [], []
import time
for _ in range(10):
    t = time.time()
    kmeans = kMeans(points, 4)
    kMeans_times.append(time.time()-t)
    kMeans_scores.append(kmeans.score())
    t = time.time()
    kmedoids = kMedoids(points, 4)
    kMedoids_times.append(time.time()-t)
    kMedoids_scores.append(kmedoids.score())
means_avg = np.mean(np.array(kMeans_scores))
means_max = max(kMeans_scores)
means_min = min(kMeans_scores)
medoids_avg = np.mean(np.array(kMedoids_scores))
medoids_max = max(kMedoids_scores)
medoids_min = min(kMedoids_scores)
kMeans_time = np.mean(np.array(kMeans_times))
kMedoids_time = np.mean(np.array(kMedoids_times))
print ("K-means time: %s" % (str(kMeans_time)))
print ("K-medoids time: %s" % (str(kMedoids_time)))
print ("K-means average: %s, min: %s, max: %s" % (str(means_avg), str(means_min), str(means_max)))
print ("K-medoids average: %s, min: %s, max: %s" % (str(medoids_avg), str(medoids_min), str(medoids_max)))
```

Now construct another dataset by selecting 40 images each from two individuals 4 and 13.

(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 $l = 1, 3, 5, \ldots, 41$. 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 we increase the number of principal components, the K-means and K-medoids scores improve; limiting the number of principal components results in the scores being lower because there is simply too little information to cluster points off of. With more principal components, the algorithms have more features to cluster points by. If we keep increasing the number of principal components beyond 40, the scores will not increase that much (as we see the scores reach a plateau of around score=0.90 and 0.85).

K-medoids algorithm produces a better score than K-means. This is likely because for facial recognition, it is better for the cluster centroid to be an actual point of the dataset instead of just being the average of the points that are assigned to it, since averaging the attributes creates unrealistic outcomes that are less akin to a face as opposed to the centroid being an actual attribute on a face.

```python
# part 3b: explore effect of lower-dimensional representations on clustering performance
np.random.seed(1234)
X2, y2 = util.limit_pics(X, y, [4, 13], 40)
l_kmeans = []
l_kmedoids = []
lst = [i for i in range(1, 42) if i%2 == 1]
for l in lst:
    Z, Ul = apply_PCA_from_Eig(X2, U, l, mu)
    X2_rec = reconstruct_from_PCA(Z, Ul, mu)
    points = build_face_image_points(X2_rec, y2)
    l_kmeans.append(kMeans(points, 2, init='cheat').score())
    l_kmedoids.append(kMedoids(points, 2, init='cheat').score())
plt.plot(lst, l_kmeans, label='K-means')
plt.plot(lst, l_kmedoids, label='K-medoids')
plt.title('K-means and K-medoids score vs, number of principal components')
plt.xlabel('Number of principal components')
plt.ylabel('Clustering score')
plt.legend()
plt.show()
```

Some pairs of people are more similar to one another and some more different.

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

I used the KMedoids algorithm with cheat initialization, since KMedoids produces a better clustering score than KMeans does, and cheat initialization runs in fewer iterations than with random initialization. My implementation iterates through all possible pairs of images out of the 19, gets the score for each pair, and finds the max (most discriminative) and min (least discriminative) score and pairs.

Most discriminative pair of images:

Least discriminative pair of images:

For the most discriminative clustering, the faces are very clearly different; they differ in face shape, skin tone, forehead size, teeth/no teeth, etc. So, the KMedoids clustering algorithm is able to cluster the images separately.
For the least discriminative clustering, the faces are very similar (skin tone, wrinkles, eye size, forehead size, nose, mouth, etc.). So, the clustering algorithm is unable to easily assign datapoints to different clusters.

```
# part 3c: determine ``most discriminative'' and ``least discriminative'' pairs of images
np.random.seed(1234)
max_score = -1
min_score = np.inf
max_score_pair = None, None
min_score_pair = None, None
for i in range(19):
    for j in range(19):
        if i != j:
            X_ij, y_ij = util.limit_pics(X, y, [i,j], 40)
            points = build_face_image_points(X_ij, y_ij)
            score = kMedoids(points, 2, init='cheat').score() # KMedoids better than KMeans
            if score < min_score:                             # cheat_init --> fewer iters
                min_score = score
                min_score_pair = i, j
            if score > max_score:
                max_score = score
                max_score_pair = i, j
plot_representative_images(X, y, min_score_pair, 'Least discriminative pair of images')
plot_representative_images(X, y, max_score_pair, 'Most discriminiative pair of images')
```