CM146, Winter 2024
Problem Set 4: Boosting, Unsupervised learning
Due March 15, 11:59pm (Math), March 17, 11:59pm (Coding)

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) **(3 pts)** 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.
Solution: Let $\epsilon_t = \sum_n w_t(n)\mathbb{I}[y_n \neq h_t(\boldsymbol{x}_n)]$ and $\delta_t = \sum_n w_t(n)$. Since we normalize the weights, $\delta_t = 1$. So, the objective function is:

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

Taking the derivative with respect to β_t and setting it to zero, we get:

$$\frac{\partial L}{\partial \beta_t} = \beta_t e^{\beta_t}\epsilon_t + \beta_t e^{-\beta_t}\epsilon_t - \beta_t e^{-\beta_t} := 0$$

Multiplying both sides by $\frac{e^{\beta_t}}{\beta_t}$, we get:

$$e^{2\beta_t}\epsilon_t + \epsilon_t - 1 = 0.$$

$$e^{2\beta_t} = \frac{1}{\epsilon_t} - 1.$$

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

Thus,

$$\boxed{\beta_t = \frac{1}{2}\ln\left(\frac{1}{\sum_n w_t(n)\mathbb{I}[y_n \neq h_t(\boldsymbol{x}_n)]} - 1\right).}$$

Parts of this assignment are adapted from course material by Jenna Wiens (UMich) and Tommi Jaakola (MIT).

(b) **(2 pts)** 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?

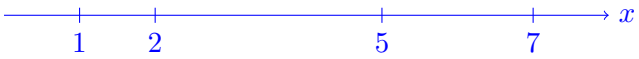
Solution: Since the training set is linearly separable, the weighted misclassification error ϵ_1 is 0. Thus, $\beta_1 = \frac{1}{2}\ln\left(\frac{1}{0} - 1\right) = \boxed{\infty}$. Since the data is linearly separable, this classifier is perfect, so it makes sense that we give it infinite weight.

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

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

(a) **(2 pts)** 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 K-means objective ?

Solution: Here is a visualization of the data:



Since x_1 and x_2 are the closest, they should be in the same cluster. Thus, the optimal clustering is $\{1, 2\}, \{5\}, \{7\}$ with corresponding means of $\mu_1 = 1.5$, $\mu_2 = 5$, and $\mu_3 = 7$.

Substituting these values into the K-means objective function, we get:

$$\begin{aligned} J(r_{nk}, \mu_k) &= \sum_{n=1}^{4}\sum_{k=1}^{3} r_{nk}||x_n - \mu_k||_2^2 \\ &= \left[(1-1.5)^2 + (2-1.5)^2\right] + \left[(5-5)^2\right] + \left[(7-7)^2\right] \\ &= 0.5 \end{aligned}$$

Thus, the optimal clustering is $\boxed{\{1, 2\}, \{5\}, \{7\}}$ and the corresponding value of the K-means objective is $\boxed{0.5}$.

(b) **(3 pts)** One might be tempted to think that Lloyd's algorithm is guaranteed to converge to the global minimum when dimension $d = 1$. 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).

Solution: Initialize $\mu_1 = 1$, $\mu_2 = 2$, and $\mu_3 = 6$. Minimizing J over r_{nk} assuming the μ_k are fixed leads to the clustering $\{1\}, \{2\}, \{5, 7\}$, as this minimizes the distance between the means and the data points. Now assuming the r_{nk} are fixed, we recompute the means for each cluster to minimize J. This leads to $\mu_1 = 1$, $\mu_2 = 2$, and $\mu_3 = 6$. This is the same as the initialization, which means the algorithm has converged. However, this clustering is suboptimal, as the value of the objective function is greater than the optimal value found in part (a):

$$J(r_{nk}, \mu_k) = \left[(1-1)^2\right] + \left[(2-2)^2\right] + \left[(5-6)^2 + (7-6)^2\right] = 2.$$

Thus, even when $d = 1$, Lloyd's algorithm is not guaranteed to converge to the global minimum.

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 (i.e., we have the 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 the 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) **(2 pts)** To maximize l, compute $\nabla_{\boldsymbol{\mu}_j} l(\boldsymbol{\theta})$: the gradient of $l(\boldsymbol{\theta})$ with respect to $\boldsymbol{\mu}_j$.

Solution: The entire first term and all except the j^{th} term in the sum over k in the second term are constant with respect to $\boldsymbol{\mu}_j$, so we can ignore them when taking the gradient. Thus, we are effectively taking the gradient of $\sum_n \gamma_{nj} \log N(\boldsymbol{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)$. Substituting in the expression for the Gaussian distribution, we get:

$$\begin{aligned} \nabla_{\boldsymbol{\mu}_j} l(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\mu}_j} \sum_n \gamma_{nj} \log \left(\frac{1}{(2\pi)^{d/2} |\boldsymbol{\Sigma}_j|^{1/2}} e^{-\frac{1}{2}(\boldsymbol{x}_n - \boldsymbol{\mu}_j)^T \boldsymbol{\Sigma}_j^{-1} (\boldsymbol{x}_n - \boldsymbol{\mu}_j)} \right) \\ &= \nabla_{\boldsymbol{\mu}_j} \sum_n \gamma_{nj} \left(\log \frac{1}{(2\pi)^{d/2} |\boldsymbol{\Sigma}_j|^{1/2}} - \frac{1}{2} (\boldsymbol{x}_n - \boldsymbol{\mu}_j)^T \boldsymbol{\Sigma}_j^{-1} (\boldsymbol{x}_n - \boldsymbol{\mu}_j) \right) \\ &= \nabla_{\boldsymbol{\mu}_j} \sum_n \gamma_{nj} \left(-\frac{1}{2} (\boldsymbol{x}_n - \boldsymbol{\mu}_j)^T \boldsymbol{\Sigma}_j^{-1} (\boldsymbol{x}_n - \boldsymbol{\mu}_j) \right) \end{aligned}$$

We know that $\nabla_{\boldsymbol{x}} \boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x} = (\boldsymbol{A} + \boldsymbol{A}^T)\boldsymbol{x}$, so taking the gradient with respect to $\boldsymbol{\mu}_j$ yields

$$\nabla_{\boldsymbol{\mu}_j} l(\boldsymbol{\theta}) = -\frac{1}{2} \sum_n \gamma_{nj} \left[\boldsymbol{\Sigma}_j^{-1} + (\boldsymbol{\Sigma}_j^{-1})^T \right] (\boldsymbol{x}_n - \boldsymbol{\mu}_j)(-1).$$

Further, since $\boldsymbol{\Sigma}_j$ is symmetric, we have $\boldsymbol{\Sigma}_j^{-1} = (\boldsymbol{\Sigma}_j^{-1})^T$, so the gradient simplifies to

$$\boxed{\nabla_{\boldsymbol{\mu}_j} l(\boldsymbol{\theta}) = \sum_n \gamma_{nj} \boldsymbol{\Sigma}_j^{-1} (\boldsymbol{x}_n - \boldsymbol{\mu}_j)}.$$

(b) **(2 pts)** Set the gradient to zero and solve for $\boldsymbol{\mu}_j$ to show that $\boldsymbol{\mu}_j = \frac{1}{\sum_n \gamma_{nj}} \sum_n \gamma_{nj} \boldsymbol{x}_n$.

Solution: Setting the gradient to zero and factoring out $\boldsymbol{\Sigma}_j^{-1}$ and $\boldsymbol{\mu}_j$ (because they are not dependent on n), we get:

$$\begin{aligned} 0 &= \sum_n \gamma_{nj} \boldsymbol{\Sigma}_j^{-1} (\boldsymbol{x}_n - \boldsymbol{\mu}_j) \\ &= \boldsymbol{\Sigma}_j^{-1} \left[\sum_n \gamma_{nj} \boldsymbol{x}_n - \boldsymbol{\mu}_j \sum_n \gamma_{nj} \right] \end{aligned}$$

Multiplying both sides by $\boldsymbol{\Sigma}_j$ and rearranging, we get:

$$\boldsymbol{\mu}_j \sum_n \gamma_{nj} = \sum_n \gamma_{nj} \boldsymbol{x}_n.$$

Thus, the MLE for $\boldsymbol{\mu}_j$ is

$$\boxed{\boldsymbol{\mu}_j = \frac{1}{\sum_n \gamma_{nj}} \sum_n \gamma_{nj} \boldsymbol{x}_n}.$$

(c) **(4 pts)** 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?

Solution: The updated values for the parameters are:

$$\omega_1 = \frac{1}{5} \sum_n \gamma_{n1} = \frac{1}{5}(0.2 + 0.2 + 0.8 + 0.9 + 0.9) = 0.6$$

$$\omega_2 = \frac{1}{5} \sum_n \gamma_{n2} = \frac{1}{5}(0.8 + 0.8 + 0.2 + 0.1 + 0.1) = 0.4$$

$$\mu_1 = \frac{1}{\sum_n \gamma_{n1}} \sum_n \gamma_{n1} \boldsymbol{x}_n = \frac{(0.2 \cdot 5 + 0.2 \cdot 15 + 0.8 \cdot 25 + 0.9 \cdot 30 + 0.9 \cdot 40)}{0.2 + 0.2 + 0.8 + 0.9 + 0.9} = 29$$

$$\mu_2 = \frac{1}{\sum_n \gamma_{n2}} \sum_n \gamma_{n2} \boldsymbol{x}_n = \frac{(0.8 \cdot 5 + 0.8 \cdot 15 + 0.2 \cdot 25 + 0.1 \cdot 30 + 0.1 \cdot 40)}{0.8 + 0.8 + 0.2 + 0.1 + 0.1} = 14.$$

4 Implementation: Clustering and PCA [32 pts]

Machine learning techniques have been applied to a variety of image interpretation problems. In this project, you will investigate facial recognition, which can be treated as a clustering problem ("separate these pictures of Joe and Mary").

For this project, we will use a small part of a huge database of faces of famous people (Labeled Faces in the Wild [LFW] people dataset[1]). The images have already been cropped out of the original image, and scaled and rotated so that the eyes and mouth are roughly in alignment; additionally, we will use a version that is scaled down to a manageable size of 50 by 37 pixels (for a total of 1850 "raw" features). Our dataset has a total of 1867 images of 19 different people. You will apply dimensionality reduction using principal component analysis (PCA) and explore clustering methods such as k-means and k-medoids to the problem of facial recognition on this dataset.

Starter Files

---

code and data

- Code: CS146-Winter2024-PS4.ipynb – Code for the `Point`, `Cluster`, and `ClusterSet` classes, on which you will build the clustering algorithms and the main code for the project.
- Utility code: util.py – Utility methods for manipulating data, including through PCA.

Please use your *@g.ucla.edu* email id to access the code. Similar to previous problem sets, copy the colab notebook to your drive and make the changes. Mount the drive appropriately. To work on this HW: you need to download util.py from here. Then, copy/upload this file to your own Google drive.

The notebook has marked blocks where you need to code.

$### ========= TODO : START ========= ###$

$### ========= TODO : END ========== ###$

Note: For the questions requiring you to complete a piece of code, you are expected to copy-paste your code as a part of the solution in the submission pdf. Tip: If you are using LaTeX, check out the Minted package (example) for code highlighting.

Please note that you do not necessarily have to follow the skeleton code perfectly. We encourage you to include your own additional methods and functions. For this project, *you are not allowed to use any* `scikit-learn` *classes or functions other than those already imported in the skeleton code.*

4.1 PCA and Image Reconstruction [4 pts]

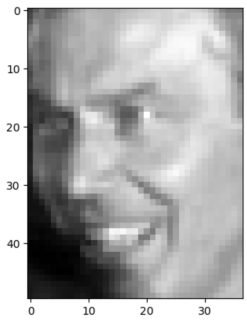
Before attempting automated facial recognition, you will investigate a general problem with images. That is, images are typically represented as thousands (in this project) to millions (more generally) of pixel values, and a high-dimensional vector of pixels must be reduced to a reasonably low-dimensional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

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

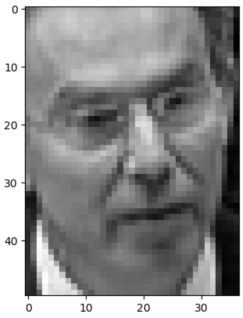
(a) **(1 pts)** As always, the first thing to do with any new dataset is to look at it. Use `get_lfw_data(...)` to get the LFW dataset with labels, and plot a couple of the input images using `show_image(...)`. Then compute the mean of all the images, and plot it. (Remember to include all requested images in your writeup.) Comment briefly on this "average" face.

```
X, Y = get_lfw_data()
```

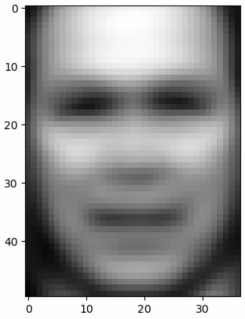
Solution: The average face is blurry and not a specific person's face, but it does have some general facial features. There are clear eyes, nose, mouth, and facial outline, but the features are not distinctive enough to identify a specific individual.



(a) Sample Face 1



(b) Sample Face 2



(c) Average Face

(b) **(1 pts)** 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?

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

Solution: The top eigenfaces capture the most variation in the data. Each of the 12 images have different lighting, angles, and contrast, indicating that these are the most important features for distinguishing between the images.

Figure 2: Top 12 Eigenfaces

(c) **(2 pts)** Explore the effect of using more or fewer dimensions to represent images. Do this by:

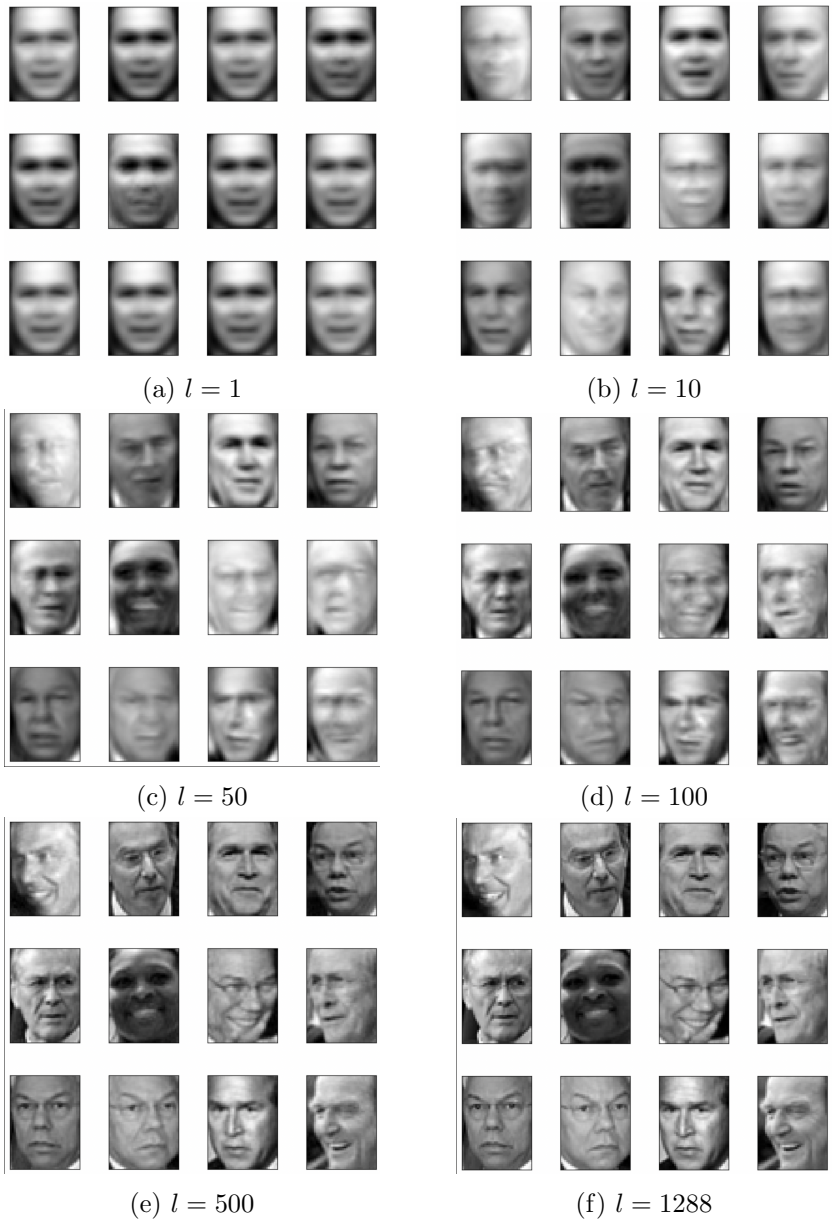
- Finding the principal components of the data
- Selecting a number l of components to use
- Reconstructing the images using only the first l principal components
- Visually comparing the images to the originals

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.

```
num_components = [1, 10, 50, 100, 500, 1288]
print("Original Images:")
plot_gallery([vec_to_image(X[:,i]) for i in range(12)])

for n in num_components:
    print(f"l={n} components: ")
    Z, Ul = apply_PCA_from_Eig(X, U, n, mu)
    X_rec = reconstruct_from_PCA(Z, Ul, mu)
    plot_gallery(X_rec[:12])
```

Solution: The reconstructed images are shown in Figure 3. As l increases, the images become clearer and more detailed. This is because the higher the number of principal components, the more information is retained. However, the images are still recognizable even with only 10 principal components, which is only 0.78% of the original 1288 dimensions. This indicates that the most important features for facial recognition are captured in the first few

principal components.



(a) $l = 1$ (b) $l = 10$

(c) $l = 50$ (d) $l = 100$

(e) $l = 500$ (f) $l = 1288$

Figure 3: Reconstructed Images

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

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) **(1 pts)** 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?

Solution: The minimum possible value of $J(\boldsymbol{c}, \boldsymbol{\mu}, k)$ is 0. This occurs when $k = n$, $\boldsymbol{c} = \{1, \ldots, n\}$, and $\boldsymbol{\mu}_j = \boldsymbol{x}^{(j)}$ for all $j \in \{1, \ldots, n\}$. We are effectively defining each data point as its own cluster, which does not help us find any meaningful structure in the data.

(b) **(3 pts)** 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.

Solution:

```
class Cluster(object):
    ...
    def centroid(self) :
            labels = [point.label for point in self.points]
            most_common_label = np.argmax(np.bincount(labels))
            mean = np.mean([point.attrs for point in self.points], axis=0)
            centroid = Point("centroid", most_common_label, mean)
            return centroid
    def medoid(self) :
        distances = [] # (point, distance)
        for point in self.points:
            distance = sum([point.distance(other) for other in self.points])
            distances.append((point, distance))
        distances.sort(key=lambda x: x[1])
        medoid = distances[0][0]
        return medoid

class ClusterSet(object):
    ...
    def centroids(self):
        return [cluster.centroid() for cluster in self.members]
    def medoids(self):
        return [cluster.medoid() for cluster in self.members]
```

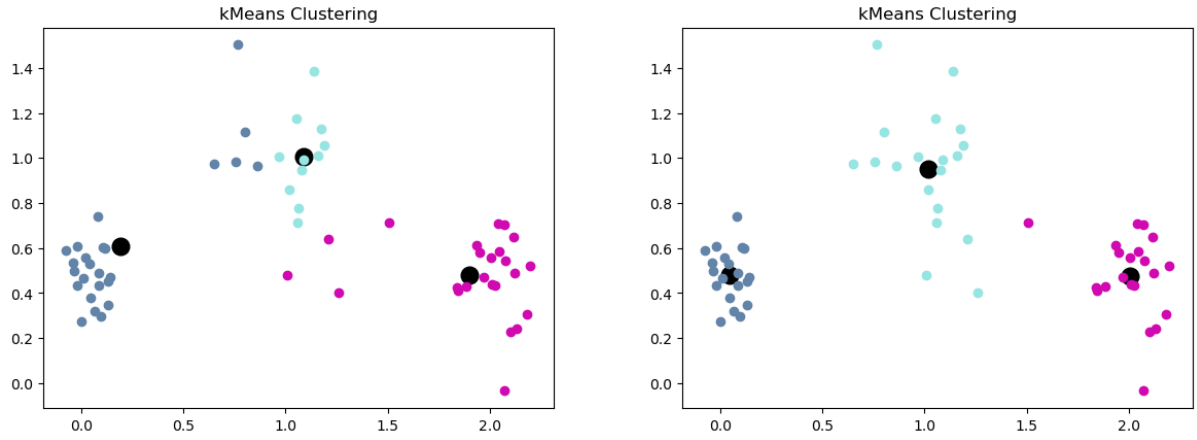
(c) **(6 pts)** Next, implement `random_init(...)` and `kMeans(...)` based on the specifications provided in the code.
Solution:

```
def random_init(points, k) :
    return np.random.choice(points, k, replace=False)


def kMeans(points, k, init='random', plot=False) :

    prev, curr = ClusterSet(), ClusterSet()
    prev.add(Cluster([]))  # dummy initialization to enter loop
    means = random_init(points, k) if init == 'random' else cheat_init(points)

    while not prev.equivalent(curr):

        # udpate clusters
        clusters = {mean: [] for mean in means}
        for point in points:
            closest_mean = min(means, key=lambda mean: point.distance(mean))
            clusters[closest_mean].append(point)

        # update ClusterSet and means
        prev = curr
        curr = ClusterSet()
        for cluster in clusters.values():
            curr.add(Cluster(cluster))
        means = curr.centroids()

    if plot: plot_clusters(curr, "kMeans Clustering", ClusterSet.centroids)

    return curr
```

(d) **(1 pts)** Now test the performance of k-means on a toy dataset.

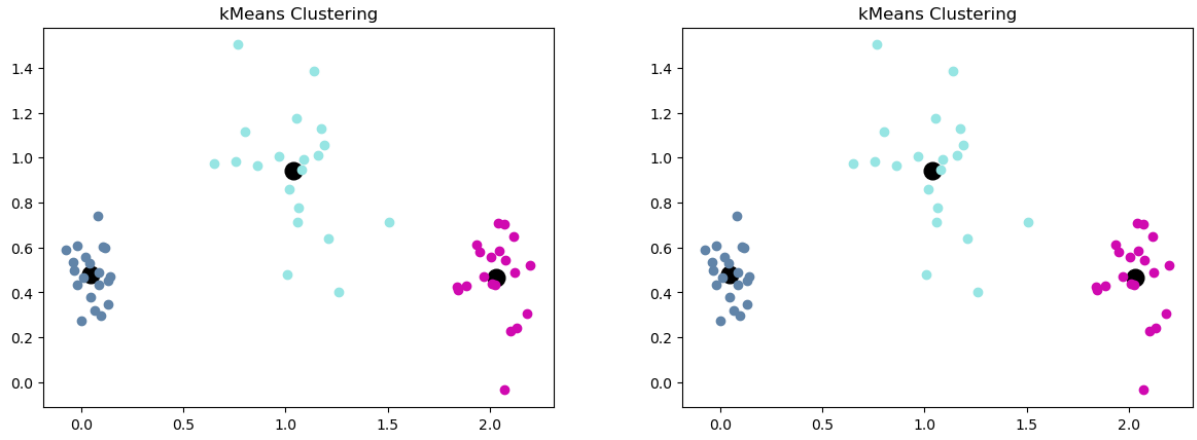
Use `generate_points_2d(...)` to generate three clusters each containing 20 points. (You can modify `generate_points_2d(...)` to test different inputs while debugging your code, but be sure to return to the initial implementation before creating any plots for submission.) You can plot the clusters for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



(a) Iteration 1 (random initialization)

(b) Iteration 2



(c) Iteration 3

(d) Iteration 4

Figure 4: Clustering by k-means for each iteration

(e) **(3 pts)** Implement `kMedoids(...)` based on the provided specification.

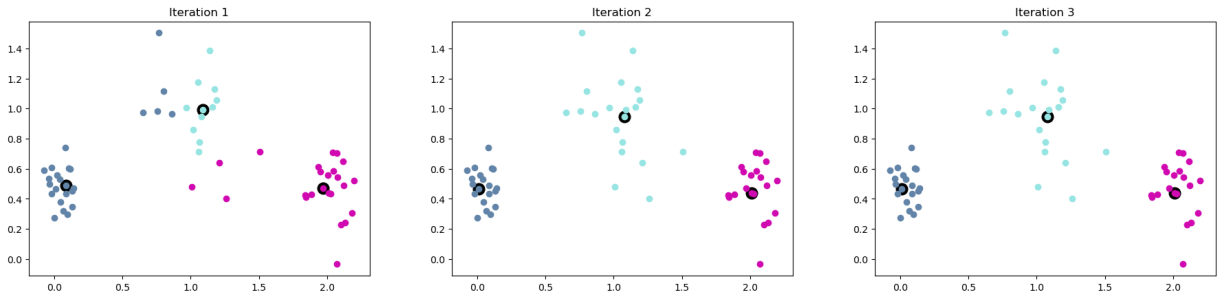
Hint: Since k-means and k-medoids are so similar, you may find it useful to refactor your code to use a helper function `kAverages(points, k, average, init='random', plot=False)`, where `average` is a method that determines how to calculate the average of points in a cluster (so it can take on values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

Solution:

```
def kMedoids(points, k, init='random', plot=False):
    return kAverages(points, k, ClusterSet.medoids, init, plot)

def kAverages(points, k, average, init='random', plot=False):
    prev, curr = ClusterSet(), ClusterSet()
    prev.add(Cluster([]))
    means = random_init(points, k) if init == 'random' else cheat_init(points)
    i=1
```

[2]In Python, if you have a function stored to the variable `func`, you can apply it to parameters `arg` by callling `func(arg)`. This works even if `func` is a class method and `arg` is an object that is an instance of the class.

```
    while not prev.equivalent(curr):

        clusters = {mean: [] for mean in means}
        for point in points:
            closest_mean = min(means, key=lambda mean: point.distance(mean))
            clusters[closest_mean].append(point)

        prev = curr
        curr = ClusterSet()
        for cluster in clusters.values():
            curr.add(Cluster(cluster))
        means = average(curr)

        if plot: plot_clusters(curr, f"Iteration {i}", average)
        i += 1

    return curr
```



(a) Iteration 1 (b) Iteration 2 (c) Iteration 3

Figure 5: Clustering by k-medoids for each iteration

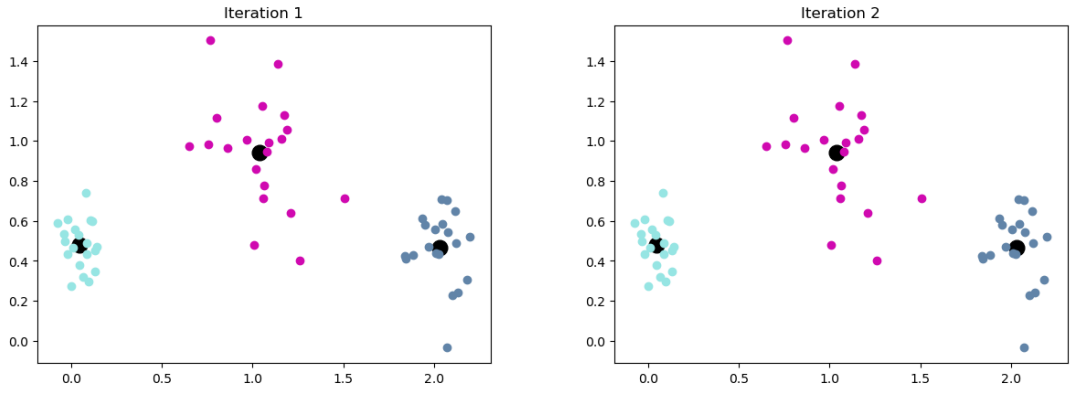
As before, include plots for k-medoids clustering *for each iteration* when using random initialization and "k = 3".

(f) **(2 pts)** 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.*

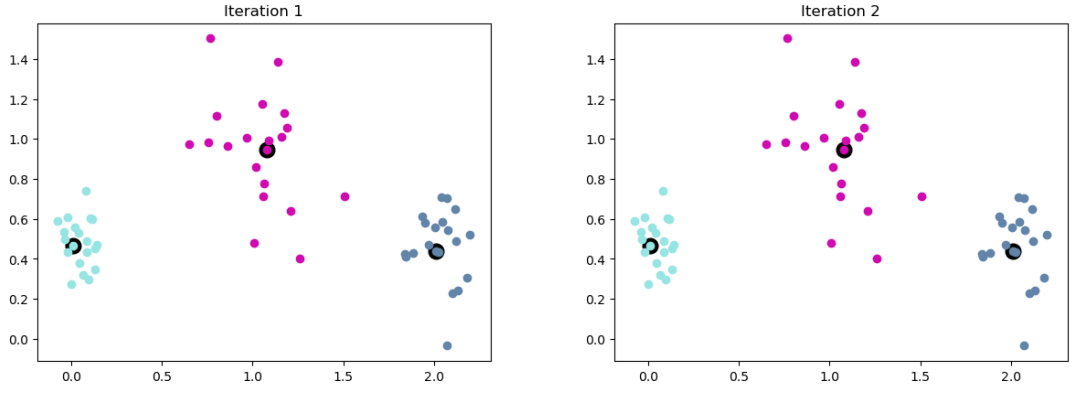
Solution:

```
def cheat_init(points):
    label_to_points = defaultdict(list)
    for point in points:
        label_to_points[point.label].append(point)
    return [Cluster(points).medoid() for points in label_to_points.values()]
```



(a) Iteration 1 (b) Iteration 2

Figure 6: Clustering by k-means with cheat initialization for each iteration



(a) Iteration 1 (b) Iteration 2

Figure 7: Clustering by k-medoids with cheat initialization for each iteration

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) **(2 pts)** 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

[3] There is a bug in fetch_lfw version 0.18.1 where the results of the loaded images are not always in the same order. This is not a problem for the previous parts but can affect the subset selected in this part. Thus, you may see varying results. Results that show the correct qualitative behavior will get full credit.

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?

Solution:

	average purity	min purity	max purity	average time	min time	max time
k-means	0.622	0.525	0.7	0.067	0.034	0.142
k-medoids	0.621	0.55	0.669	0.106	0.082	0.174

The two clustering methods have similar average purity, but k-means is faster than k-medoids.

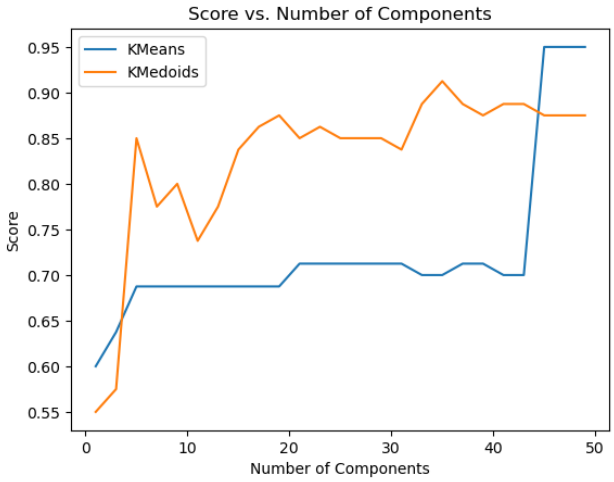
Now construct another dataset by selecting 40 images each from two individuals 4 and 13.

(b) **(5 pts)** 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.

Solution: K-Medoids performs significantly better than k-means for smaller numbers of principal components, but k-means jumps ahead around 45 principal components. This is likely because k-medoids is more robust to noise and outliers, but k-means is better at capturing the overall structure of the data. Performance generally increases with the number of principal components.



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

(c) **(5 pts)** 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.
Solution: To find the most similar individuals, I found the image pairs with the lowest purity (their clusters are not well-separated), and to find the most discriminative individuals, I found the image pairs with the highest purity (their clusters are very well-separated). I tried both K-means and K-medoids. K-means was able to perfectly distinguish between the most different individuals, but had more trouble with the most similar individuals.

(a) 0, 2 (b) 5, 17 (c) 8, 12 (d) 8, 17

Figure 8: Most Similar Pairs (purity = 0.525) with K-Medoids

(a) 9, 16 (b) 0, 13 (c) 0, 6

Figure 9: Most Discriminative Pairs (purity = 0.9875) with K-Medoids

(a) 4, 12

Figure 10: Most Similar Pair (purity = 0.5) with K-Means

(a) 3, 13 (b) 0, 13

Figure 11: Most Discriminative Pairs (purity = 1.0) with K-Means