CM146, Winter 2021
Problem Set 4: Boosting, Unsupervised learning
Due March 12, 2021, 11:59pm

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:

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

$$\begin{aligned}\frac{\partial J(\beta_t)}{\partial \beta_t} &= (e^{\beta_t} + e^{-\beta_t})\epsilon_t - e^{-\beta_t} \\ &= 0\end{aligned}$$

Solving gives:

$$\begin{aligned}e^{2\beta_t} + 1 &= \frac{1}{\epsilon_t} \\ e^{2\beta_t} &= \frac{1-\epsilon_t}{\epsilon_t} \\ \beta_t &= \frac{1}{2}\log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)\end{aligned}$$

(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: The value of β_1 is infinite. Increasing β_1 will decrease all the training losses since $y_n h(\boldsymbol{x}_n) > 0$ for all n.

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

That is, if the training set is linearly separable and we use a hard-margin SVM with no slack as a base classifier, then a single classifier is sufficient for the ensemble classifier. Thus, in stage $t = 1$, $h(\boldsymbol{x})$ will correspond to a linear decision boundary that correctly classifies all points ($\epsilon_1 = 0$) so that $\beta_1 = \infty$.

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 objective ?

Solution:

$$c_1 = 1.5, c_2 = 5, c_3 = 7, objective = 0.5^2 + 0.5^2 + 0 + 0 = 0.5$$

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

Solution:

$$c_1 = 1, c_2 = 2, c_3 = 6, objective = 0^2 + 0^2 + 1^2 + 1^2 = 2$$

When running Lloyd's on this configuration, the new cluster assignment will be $z_1 = 1, z_2 = 2, z_3 = 3, z_4 = 3$.

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) **(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:** Gaussian/Normal distribution:

$$\mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) = \frac{1}{\sqrt{(2\pi)^d |\boldsymbol{\Sigma}_k|}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_k^{-1} (\boldsymbol{x} - \boldsymbol{\mu}_k)\right)$$

The log-likelihood function:

$$l(\boldsymbol{\theta}) = \text{Const} + \sum_k \left\{ \sum_n \gamma_{nk} \log N(\boldsymbol{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right\}$$

which can be simplified as

$$l(\boldsymbol{\theta}) = \sum_k \left\{ \sum_n \gamma_{nk} \left[-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_j^{-1} (\boldsymbol{x} - \boldsymbol{\mu}_k) \right] \right\}$$

The gradient is computed as,

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

Note that we've applied,

$$\frac{\partial \left(\boldsymbol{x}^T A \boldsymbol{x}\right)}{\partial \boldsymbol{x}} = \boldsymbol{x}^T \left(A + A^T\right)$$

(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:** Set the results in (1) as zero and we can directly get

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

(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 **??**.

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?

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}

Solution: The one-dimensional GMM is defined as

$$p(x) = \sum_{i=1}^{K} \omega_i \mathcal{N}\left(x \mid \mu_i, \sigma_i\right)$$

$$\mathcal{N}\left(x \mid \mu_i, \sigma_i\right) = \frac{1}{\sigma_i \sqrt{2\pi}} \exp\left(-\frac{(x-\mu_i)^2}{2\sigma_i^2}\right)$$

$$\sum_{i=1}^{K} \omega_i = 1$$

Update ω_1, ω_2 in the M-Step:

$$\hat{\omega}_k = \sum_{i=1}^{N} \frac{\hat{\gamma}_{ik}}{N}$$

we have

$$\omega_1 = \frac{0.2+0.2+0.8+0.9+0.9}{5} = \frac{3}{5}, \ \omega_2 = \frac{0.8+0.8+0.2+0.2+0.1}{5} = \frac{2}{5}$$

Update μ_1, μ_2 in the M-Step:

$$\hat{\mu}_k = \frac{\sum_{i=1}^{N} \hat{\gamma}_{ik} x_i}{\sum_{i=1}^{N} \hat{\gamma}_{ik}}$$

we have

$$\mu_1 = \frac{0.2\times 5 + 0.2\times 15 + 0.8\times 25 + 0.9\times 30 + 0.9\times 40}{0.2+0.2+0.8+0.9+0.9} = \frac{87}{3} = 29$$

$$\mu_2 = \frac{0.8\times 5 + 0.8\times 15 + 0.2\times 25 + 0.1\times 30 + 0.1\times 40}{0.8+0.8+0.2+0.1+0.1} = \frac{28}{2} = 14$$

3.1 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) **(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. **Solution:**

Figure 1: (*Left*, *Middle*) Sample Images, (*Right*) 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 xrange(12)])
```

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

Figure 2: Top 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.
Solution:

Figure 3: $l = 1$

Figure 4: $l = 10$

Figure 5: $l = 50$

Figure 6: $l = 100$

Figure 7: $l = 500$

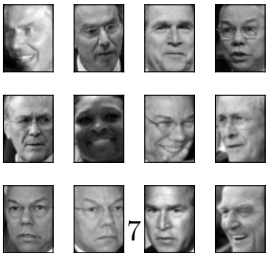

Figure 8: $l = 1288$

Figure 9: PCA Image Reconstruction

Note that the exact dataset returned by `fetch_lfw_people` seems to vary between machines, so results may differ.

For low-dimensional projections (e.g. $l = 1, 10$), it is difficult to distinguish many of the faces. For mid-dimensional values (e.g. $l = 50$), a little more detail becomes clear. For higher-dimensional projections (e.g. $l = 100$), we can distinguish the people fairly well. (Details like glasses seem to be missing, but the reconstructed images for $l = 500$ are almost indistinguishable from the original dimension data.) If we were to use this as a compressed version of the images, we would use $500/1850 \approx 27\%$ of the original storage while retaining most the original detail.

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

3.2 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 objective value is 0. It is achieved when we have n clusters such that $c_i = i$ and $\boldsymbol{\mu}_j = \boldsymbol{x}^{(i)}$.

(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` (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. **Solution:**

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

(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. **Solution:**

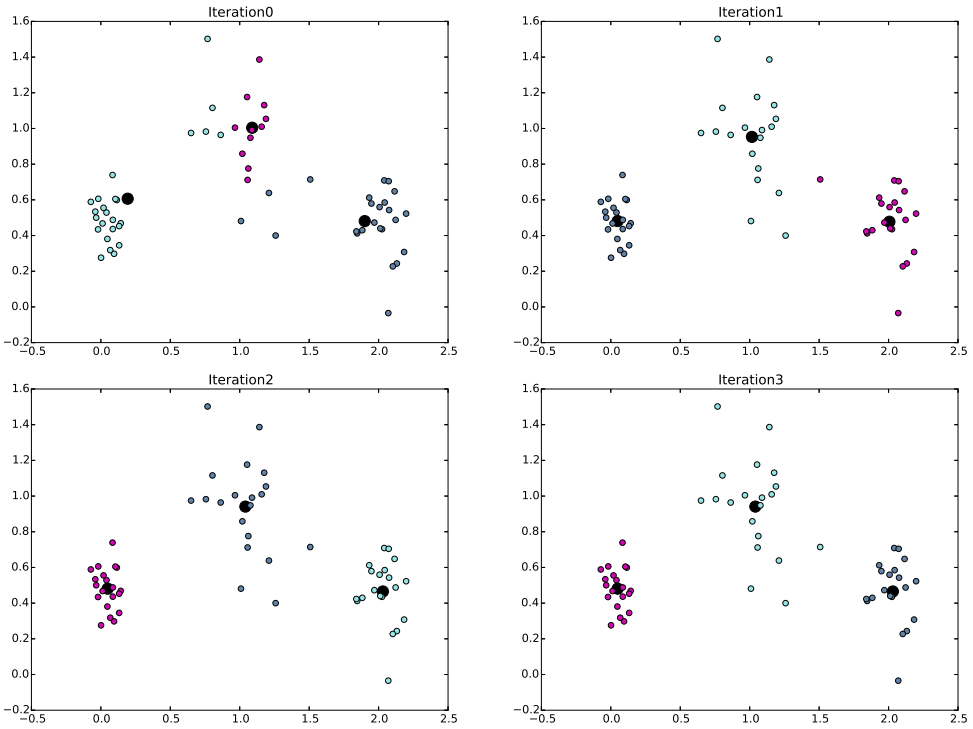

Figure 10: k-means with random initialization

Note that you may have different results based on the randomly generated toy dataset and different random initialization.

(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=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`).[1]

As before, include plots for k-medoids clustering *for each iteration* when using random initialization. **Solution:**

[1] 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.

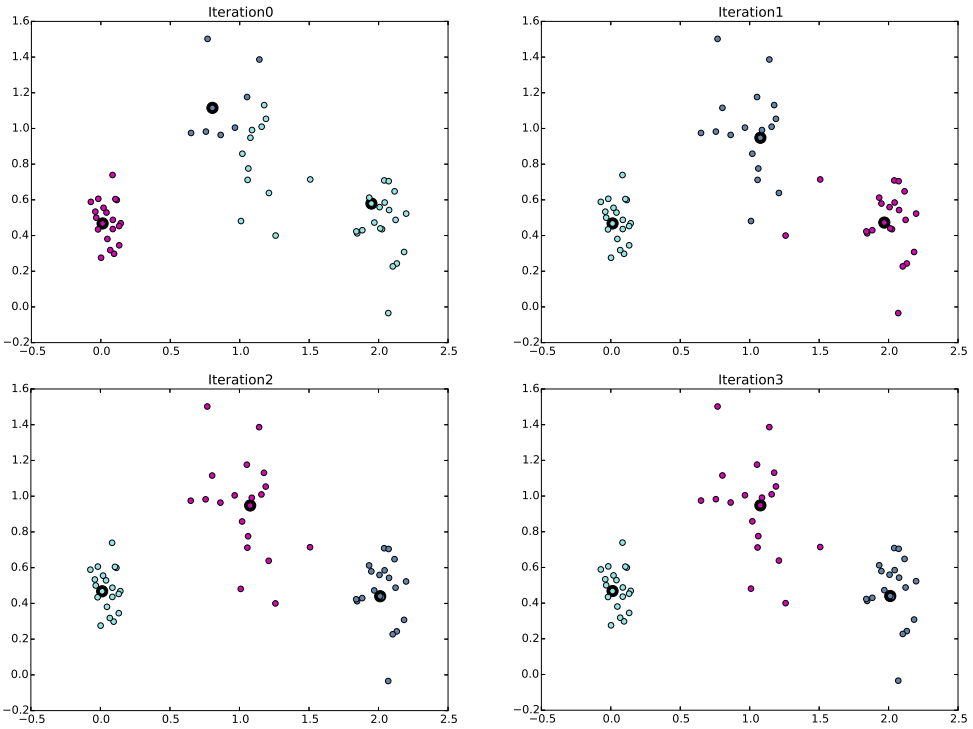

Figure 11: k-medoids with random initialization

(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 *for each iteration*. **Solution:**

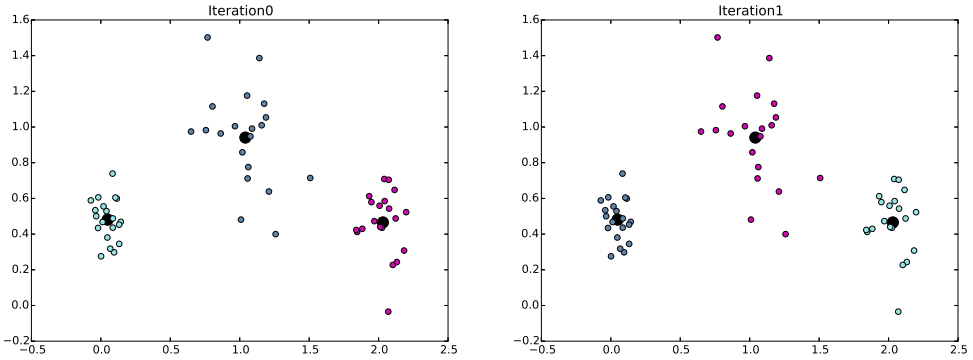

Figure 12: k-means with cheat initialization

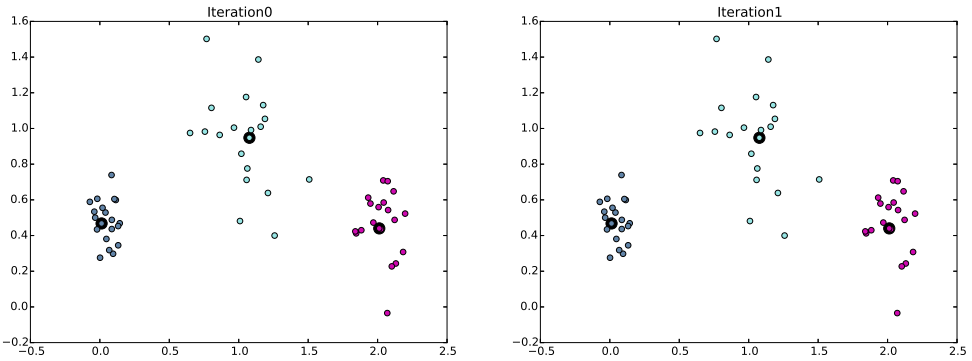

Figure 13: k-medoids with cheat initialization

3.3 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: [2]

```
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 can vary widely depending upon the initialization, run both clustering methods 10 times and report the average, minimum, and maximum performance. How do the clustering methods compare in terms of clustering performance and runtime? **Solution:**

	average	min	max
k-means	0.671875	0.5875	0.8
k-medoids	0.618125	0.56875	0.725

Note that the performance may vary depending upon images returned by `limit_pics(...)` and random initialization.

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.

[2] 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.

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. **Solution:**

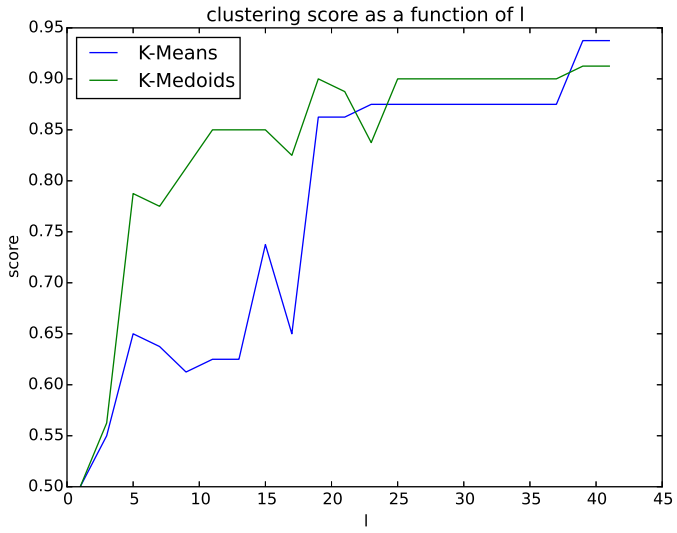

Figure 14: Clustering Score vs Number of PCs

In general, for k-means and k-medoids, as the number of principal components increases from 1 to 40, the score increases. This makes sense because having more principal components gives us more flexibility in how we try to distinguish the two image classes.

You may have observed that score seemed decrease to something slightly lower for larger values of l. This likely is a result of there being noise in the higher-dimensional representations of the images. Indeed, one application of PCA is to remove noise from a dataset.

k-medoids perform better than k-means. This is likely because k-medoids uses cluster "medians" rather than cluster "means" and is thus more robust to noise.

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 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. **Solution:** We applied k-medoids on the reconstructed data using 500 principal components, for all pairs of classes. The minimum clustering score was obtained with individuals 4 versus 5 (score = 0.500), while the maximum clustering was obtained with individuals 6 versus 16 (score = 1.00). (As noted

in previous parts, the exact pairs that you obtain might be different depending on differences in the fetched LFW dataset, different clustering algorithms, different initializations, and different number of principal components chosen.)

For our specific dataset, the easiest pair to discriminate is of two women with very different facial features and complexions while the hardest pair is of two men with similar facial features. The results from k-medoids clustering aligns with our intuition.