CM146, Winter 2021
Problem Set 4: Boosting, Unsupervised Learning
Due March 12, 2021

Name: Joseph Picchi
UID: 605-124-511

Problem 1: AdaBoost

(a) Problem 1a

Solution:

$$\frac{\partial}{\partial \beta_t}\left[\left(e^{\beta_t}-e^{-\beta_t}\right)\sum_n w_t(n)\mathbb{I}\left[y_n \neq h_t(x_n)\right]+e^{-\beta_t}\sum_n w_t(n)\right]=0$$

$$\left(e^{\beta_t}+e^{-\beta_t}\right)\sum_n w_t(n)\mathbb{I}\left[y_n \neq h_t(x_n)\right]-e^{-\beta_t}\sum_n w_t(n)=0$$

$$\left(e^{\beta_t}+e^{-\beta_t}\right)\epsilon_t-e^{-\beta_t}=0$$

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

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

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

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

$$-2\beta_t=\ln\frac{\epsilon_t}{1-\epsilon_t}$$

$$\beta_t=-\frac{1}{2}\ln\frac{\epsilon_t}{1-\epsilon_t}$$

$$\beta_t=\frac{1}{2}\ln\frac{1-\epsilon_t}{\epsilon_t}$$

(b) Problem 1b

Solution:

1. Since the training set is linearly separable, the hard-margin linear SVM will correctly classify all data points.
2. By (1), there exist no data points for which $y_n \neq h_1(x_n)$
3. By definition, $\epsilon_1 = \sum_n \frac{1}{N} \cdot \mathbb{I}\left[y_n \neq h_1(x_n)\right]$
4. By (2) and (3), $\epsilon_1 = 0$
5. $\lim_{\epsilon_1 \to 0^+} \frac{1-\epsilon_1}{\epsilon_1} = \infty$
6. By (5), $\lim_{\epsilon_1 \to 0^+} \beta_1 = \lim_{\epsilon_1 \to 0^+} \frac{1}{2} \ln \frac{1-\epsilon_1}{\epsilon_1} = \infty$
7. $\therefore$ In theory, $\beta_1 = \infty$

Problem 2: K-means for single dimensional data

(a) Problem 2a

Solution:

Since $K = 3$ and $N = 4$, there must be 2 clusters with exactly 1 point and 1 cluster with exactly 2 points. Thus, there are $\binom{4}{2} = 6$ possible clusterings.

We can enumerate over all 6 clusterings and find the corresponding objective value. Notationally, points that belong to the same set in the clusterings below are in the same cluster (e.g. the clustering $\{x_1, x_2\}, \{x_3\}, \{x_4\}$ means that cluster 1 contains x_1 and x_2, a cluster 2 contains x_3, and cluster 3 contains x_4).

(1) for the clustering $\{x_1, x_2\}, \{x_3\}, \{x_4\}$...

$$\mu_1 = (x_1 + x_2)/2 = (1+2)/2 = 1.5$$
$$\mu_2 = x_3 = 5$$
$$\mu_3 = x_4 = 7$$

$$J = (x_1 - \mu_1)^2 + (x_2 - \mu_1)^2 + (x_3 - \mu_2)^2 + (x_4 - \mu_3)^2$$
$$= (1 - 1.5)^2 + (2 - 1.5)^2 + (5 - 5)^2 + (7 - 7)^2$$
$$= 0.5$$

(2) for the clustering $\{x_1, x_3\}, \{x_2\}, \{x_4\}$...

$$\mu_1 = (x_1 + x_3)/2 = (1 + 5)/2 = 3$$
$$\mu_2 = x_2 = 2$$
$$\mu_3 = x_4 = 7$$

$$J = (x_1 - \mu_1)^2 + (x_2 - \mu_2)^2 + (x_3 - \mu_1)^2 + (x_4 - \mu_3)^2$$
$$= (1 - 3)^2 + 0 + (5 - 3)^2 + 0$$
$$= 8$$

(3) for the clustering $\{x_1, x_4\}, \{x_2\}, \{x_3\}$...

$$\mu_1 = (x_1 + x_4)/2 = (1 + 7)/2 = 4$$
$$\mu_2 = x_2 = 2$$
$$\mu_3 = x_3 = 5$$

$$J = (x_1 - \mu_1)^2 + (x_2 - \mu_2)^2 + (x_3 - \mu_3)^2 + (x_4 - \mu_1)^2$$
$$= (1 - 4)^2 + 0 + 0 + (7 - 4)^2$$
$$= 18$$

(4) for the clustering $\{x_1\}, \{x_2, x_3\}, \{x_4\}$...

$$\mu_1 = x_1 = 1$$
$$\mu_2 = (x_2 + x_3)/2 = (2 + 5)/2 = 3.5$$
$$\mu_3 = x_4 = 7$$

$$J = (x_1 - \mu_1)^2 + (x_2 - \mu_2)^2 + (x_3 - \mu_2)^2 + (x_4 - \mu_3)^2$$
$$= 0 + (2 - 3.5)^2 + (5 - 3.5)^2 + 0$$
$$= 4.5$$

(5) for the clustering $\{x_1\}, \{x_2, x_4\}, \{x_3\}$...

$$\mu_1 = x_1 = 1$$
$$\mu_2 = (x_2 + x_4)/2 = (2 + 7)/2 = 4.5$$
$$\mu_3 = x_3 = 5$$

$$J = (x_1 - \mu_1)^2 + (x_2 - \mu_2)^2 + (x_3 - \mu_3)^2 + (x_4 - \mu_2)^2$$
$$= 0 + (2 - 4.5)^2 + 0 + (7 - 4.5)^2$$
$$= 12.5$$

(6) for the clustering $\{x_1\}, \{x_2\}, \{x_3, x_4\}$...

$$\mu_1 = x_1 = 1$$
$$\mu_2 = x_2 = 2$$
$$\mu_3 = (x_3 + x_4)/2 = (5 + 7)/2 = 6$$

$$J = (x_1 - \mu_1)^2 + (x_2 - \mu_2)^2 + (x_3 - \mu_3)^2 + (x_4 - \mu_3)^2$$
$$= 0 + 0 + (5 - 6)^2 + (7 - 6)^2$$
$$= 2$$

∴ By enumerating all possible clusterings, we found…

- Optimal clustering: $\{x_1, x_2\}, \{x_3\}, \{x_4\}$
  - (i.e. x_1 & x_2 are in one cluster, x_3 is in another cluster, and x_4 is in the last cluster.)
- Corresponding value of objective: $J = 0.5$

(b) Problem 2b

Solution:

"Show the assignment"…

The suboptimal assignment is $\{x_1\}, \{x_2\}, \{x_3, x_4\}$. (i.e. x_3, x_4 in one cluster and the other 2 points in their own clusters).

"Show why it is suboptimal"…

1. From problem 2a, the optimal objective value is $J = 0.5$.
2. The objective value for this clustering is…

$$\mu_1 = x_1 = 1$$
$$\mu_2 = x_2 = 2$$
$$\mu_3 = (x_3 + x_4)/2 = (5 + 7)/2 = 6$$

$$J = (x_1 - \mu_1)^2 + (x_2 - \mu_2)^2 + (x_3 - \mu_3)^2 + (x_4 - \mu_3)^2$$
$$= 0 + 0 + (5 - 6)^2 + (7 - 6)^2$$
$$= 2$$

3. Objective for this clustering = 2 > 0.5 = optimal objective value
4. ∴ this clustering is suboptimal.

"Explain why it will not be improved"…

1. We try running another iteration of Lloyd's algorithm with this suboptimal assignment as the current assignment.
2. By (1), our current centroid values are…

   $\mu_1 = x_1 = 1$

   $\mu_2 = x_2 = 2$

   $\mu_3 = (x_3 + x_4)/2 = (5 + 7)/2 = 6$
3. In step 1 of Lloyd's algorithm…
   1. x_1 and x_2 keep their current assignments because $\|x_1 - \mu_1\|^2 = \|x_2 - \mu_2\|^2 = 0$
   2. x_3 keeps its current assignment because it is closer in value to μ_3 than it is to any other centroid.
   3. x_4 keeps its current assignment because it is closer in value to μ_3 than it is to any other centroid.
4. In step 2 of Lloyd's algorithm, all 3 centroids remain the same because none of the assignments changed in step 1 of Lloyd's.
5. By (3) and (4), nothing changed in the last iteration of Lloyd's algorithm, so the objective function J retains the same value as the prior iteration.
6. By (5), the algorithm terminates in step 3 of Lloyd's algorithm.
7. By (6), Lloyd's algorithm terminated without improving the assignment.
8. ∴ the assignment will not be improved.

Problem 3: Gaussian Mixture Models

(a) Problem 3a

Solution:

$$
\begin{aligned}
\nabla_{\mu_j} l(\theta) &= \nabla_{\mu_j} \left[\sum_k \sum_n \gamma_{nk} \log w_k + \sum_k \sum_n \gamma_{nk} \log N(x_n \mid \mu_k, \Sigma_k) \right] \\
&= \nabla_{\mu_j} \left[\sum_n \gamma_{nj} \log N(x_n \mid \mu_j, \Sigma_j) \right] \\
&= \nabla_{\mu_j} \left[\sum_n \gamma_{nj} \log \left(\frac{1}{|2\pi\Sigma_j|^{1/2}} \exp \left[-\frac{1}{2}(x_n - \mu_j)^T \Sigma_j^{-1} (x_n - \mu_j) \right] \right) \right] \\
&= \sum_n \gamma_{nj} \left[\nabla_{\mu_j} \log \left(\frac{1}{|2\pi\Sigma_j|^{1/2}} \right) - \nabla_{\mu_j} \left(\frac{1}{2}(x_n - \mu_j)^T \Sigma_j^{-1} (x_n - \mu_j) \right) \right] \\
&= \sum_n \gamma_{nj} \cdot \nabla_{\mu_j} \left(-\frac{1}{2}(x_n - \mu_j)^T \Sigma_j^{-1} (x_n - \mu_j) \right) \\
&= \sum_n \gamma_{nj} \left(-\frac{1}{2} \right) \cdot \nabla_{\mu_j} \left[(x_n - \mu_j)^T (\Sigma_J^{-1} x_n - \Sigma_J^{-1} \mu_j) \right] \\
&= \sum_n \gamma_{nj} \left(-\frac{1}{2} \right) \cdot \nabla_{\mu_j} \left[x_n^T \Sigma_j^{-1} x_n - x_n^T \Sigma_j^{-1} \mu_j - \mu_j^T \Sigma_j^{-1} x_n + \mu_j^T \Sigma_j^{-1} \mu_j \right] \\
&= \sum_n \gamma_{nj} \left(-\frac{1}{2} \right) \cdot \nabla_{\mu_j} \left[x_n^T \Sigma_j^{-1} x_n - 2\mu_j^T \Sigma_j^{-1} x_n + \mu_j^T \Sigma_j^{-1} \mu_j \right] \\
&= \sum_n \gamma_{nj} \left(-\frac{1}{2} \right) \cdot \left(0 - 2\Sigma_j^{-1} x_n + 2\Sigma_j^{-1} \mu_j \right) \\
&= \sum_n \gamma_{nj} \Sigma_j^{-1} (x_n - \mu_j)
\end{aligned}
$$

(b) Problem 3b

Solution:

$$
\begin{aligned}
\nabla_{\mu_j} l(\theta) = \sum_n \gamma_{nj} \Sigma_j^{-1}(x_n - \mu_j) &= 0 \\
\Sigma_j^{-1} \sum_n \gamma_{nj}(x_n - \mu_j) &= 0 \\
\sum_n \gamma_{nj}(x_n - \mu_j) &= 0 \\
\sum_n \gamma_{nj} x_n - \sum_n \gamma_{nj} \mu_j &= 0 \\
\mu_j \sum_n \gamma_{nj} &= \sum_n \gamma_{nj} x_n \\
\mu_j &= \frac{1}{\sum_n \gamma_{nj}} \sum_n \gamma_{nj} x_n
\end{aligned}
$$

(c) Problem 3c

Solution:

$$
\begin{aligned}
w_1 &= \frac{\sum_n \gamma_{n1}}{\sum_k \sum_n \gamma_{nk}} \\
&= \frac{\gamma_{11} + \ldots + \gamma_{51}}{\gamma_{11} + \ldots + \gamma_{51} + \gamma_{12} + \ldots + \gamma_{52}} \\
&= \frac{0.2 + 0.2 + 0.8 + 0.9 + 0.9}{0.2 + 0.2 + 0.8 + 0.9 + 0.9 + 0.8 + 0.8 + 0.2 + 0.1 + 0.1} \\
&= \frac{3}{5} \\
&= 0.6
\end{aligned}
$$

$$
\begin{aligned}
w_2 &= \frac{\sum_n \gamma_{n2}}{\sum_k \sum_n \gamma_{nk}} \\
&= \frac{\gamma_{12} + \ldots + \gamma_{52}}{\gamma_{11} + \ldots + \gamma_{51} + \gamma_{12} + \ldots + \gamma_{52}} \\
&= \frac{0.8 + 0.8 + 0.2 + 0.1 + 0.1}{0.2 + 0.2 + 0.8 + 0.9 + 0.9 + 0.8 + 0.8 + 0.2 + 0.1 + 0.1} \\
&= \frac{2}{5} \\
&= 0.4
\end{aligned}
$$

$$
\begin{aligned}
\mu_1 &= \frac{\sum_n \gamma_{n1} x_n}{\sum_n \gamma_{n1}} \\
&= \frac{\gamma_{11} x_1 + \gamma_{21} x_2 + \ldots + \gamma_{51} x_5}{\gamma_{11} + \gamma_{21} + \ldots + \gamma_{51}} \\
&= \frac{(0.2)(5) + (0.2)(15) + (0.8)(25) + (0.9)(30) + (0.9)(40)}{0.2 + 0.2 + 0.8 + 0.9 + 0.9} \\
&= \frac{87}{3} \\
&= 29
\end{aligned}
$$

$$
\begin{aligned}
\mu_2 &= \frac{\sum_n \gamma_{n2} x_n}{\sum_n \gamma_{n2}} \\
&= \frac{\gamma_{12} x_1 + \gamma_{22} x_2 + \ldots + \gamma_{52} x_5}{\gamma_{12} + \gamma_{22} + \ldots + \gamma_{52}} \\
&= \frac{(0.8)(5) + (0.8)(15) + (0.2)(25) + (0.1)(30) + (0.1)(40)}{0.8 + 0.8 + 0.2 + 0.1 + 0.1} \\
&= \frac{28}{2} \\
&= 14
\end{aligned}
$$

Problem 4: Implementation: Clustering and PCA

Problem 4.1: PCA and Image Reconstruction

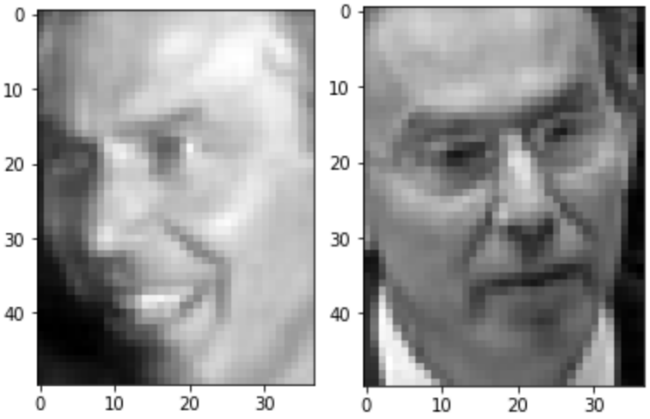
(a) Problem 4.1(a)

Solution:

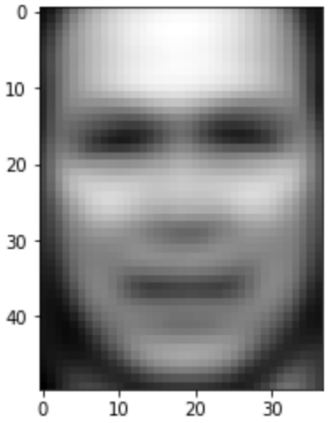
Code in "`main(…)`":

```
# PART 1A
print("running part 1A...")

# get LFW dataset
X, y = get_lfw_data()

# plot a couple of the input images
show_image(X[0])
show_image(X[1])

# compute the mean of all the images, and plot it
mean_image = np.mean(X, axis=0)
show_image(mean_image)
# END PART 1A
```

Result of plotting a couple of input images from the code in "`main(…)`":



Result of plotting the mean of all the images in "`main(…)`":



The “average” face has the general components of a human face in the correct positions (e.g. eyes, nose, and mouth are all in the visually correct locations for a human face). Furthermore, the “average” face seems to be oriented as looking straight ahead because, although each of the faces in the individual pictures has a slightly different tilt and orientation, these orientations average out to one that is looking straight out of the page. The “average” face is blurry because each of the images has their facial components (e.g. eyes, nose) in a slightly different location in the frame, thus contributing to the noise of the averaged out image. Lastly, the eyes, nose, and mouth of the “average” face seem larger than a typical human face because the different orientations in the individual picture add variation to the location of the eyes, nose, and mouth in the “average” image.

(b) Problem 4.1(b)

Solution:

Code in “`main(…)`”:

```
# PART 1B
print("running part 1B...")

# perform PCA on the data
U, mu = PCA(X)

# show the top 12 eigenfaces
plot_gallery([vec_to_image(U[:,i]) for i in range(12)])
#END PART 1B
```

Plots of the top 12 eigenfaces from the code in “`main(…)`”:

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

Each of the eigenfaces has its own characteristic lighting and facial orientation, with the top eigenfaces mainly looking out of the page and latter eigenfaces displaying a slight tilt to the side.

The top eigenfaces resemble human faces more closely than do the latter eigenfaces, which makes sense because the top eigenfaces have a smaller reconstruction error compared to the latter eigenfaces.

These were selected as the top eigenfaces because they capture the most variation out of the original dataset and because they more closely resemble human faces (i.e. capture trends in the feature values) compared to eigenfaces with lesser eigenvalues.

(c) Problem 4.1(c)

Solution:

Code in "`main(...)`":

```
# PART 1C
print("running part 1C...")

# select the principle components of the data
U, mu = PCA(X)

# select a number l of components to use
for l in [1, 10, 50, 100, 500, 1288]:
  print("for l=%d" % l)

  # project images to lower dimensional space
  Z, Ul = apply_PCA_from_Eig(X, U, l, mu)

  # reconstruct high-dimensional images from low-dimensional ones
  x_rec = reconstruct_from_PCA(Z, Ul, mu)

  # plot the first 12 images of the dataset
  plot_gallery(x_rec[0:12])

#END PART 1C
```

"Submit a gallery of the first 12 images in the dataset:

For $l = 1$:

For $l = 10$:

For $l = 50$:

For $l = 100$:

For $l = 500$:

For $l = 1288$:

"Comment briefly on the effectiveness of differing values of l with respect to facial recognition"…

Low values of l create images that do not closely resemble the raw data and that look very similar to one another, since we capture only a small fraction of the variance present in the original dataset and have a high reconstruction error.

As l increases, the images increasingly resemble the raw data because the amount of variance present in the projected dataset relative to the original dataset is nondecreasing and the reconstruction error is nonincreasing.

The more variance we have, the more effective the facial recognition algorithms will be at recognizing and distinguishing faces. Therefore, higher values of l lead to more effective facial recognition, though they also result in higher computational costs.

4.2: K-Means and K-Medoids

(a) Problem 4.2(a)

Solution:

"What is the minimum possible value of $J(c, \mu, k)$?"…

The minimum value is $J(c, \mu, k) = 0$.

"What values of c, μ, and k result in this value?"…

$J(c, \mu, k)$ achieves its minimum value when each point $x^{(i)}$ is assigned to its own cluster, resulting in the following values:

- $k = n$
- $c = \begin{bmatrix} c^{(1)} & c^{(2)} & \dots & c^{(n)} \end{bmatrix}^T = [1 \quad 2 \quad \dots \quad n]^T$
- $\mu = [\mu_1 \quad \mu_2 \quad \cdots \quad \mu_k]^T = \begin{bmatrix} x^{(1)} & x^{(2)} & \dots & x^{(n)} \end{bmatrix}^T$

:"Show that this is a bad idea"…

It is a bad idea to minimize the objective function over c, μ, and k because we can always achieve the minimum possible value of the objective by creating k clusters and assigning each point $x^{(i)}$ to its own cluster, since the mean of that cluster would consequently be $x^{(i)}$ and the distance from $x^{(i)}$ to itself is always 0.

Thus, minimizing over k is trivial because $k = n$ with a proper initialization will always result in a minimum objective function value of 0. Thus, $k = n$ with always be the selected value of k, and the resulting model will likely overfit to the training data.

(b) Problem 4.2(b)

Solution:

Code in "`Cluster.centroid(…)`":

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

Code in "`Cluster.medoid(…)`":

```
### ========== TODO : START ========== ###
# part 2b: implement

# for each point, find the sum of the distances to all other points
dist_to_others = []
for p1 in self.points:
  p1_dist = 0

  # sum up distance from p1 to all other points
  for p2 in self.points:
    p1_dist += p1.distance(p2)

  dist_to_others.append(p1_dist)

# set medoid to the point with the minimum cumulative distance to
# all other points
index_of_medoid = dist_to_others.index(min(dist_to_others))
medoid = self.points[index_of_medoid]
return medoid
### ========== TODO : END ========== ###
```

Code in "`ClusterSet.centroids(…)`":

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

Code in "`ClusterSet.medoids(…)`":

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

(c) Problem 4.2(c)

Solution:

Code in "`random_init(…)`":

```
### ========== TODO : START ========== ###
# part 2c: implement (hint: use np.random.choice)
return np.random.choice(points, size=k, replace=False)
### ========== TODO : END ========== ###
```

Code in “kMeans(…)”:

—> I added the parameter “average” to the parameter list…

```
def kMeans(points, k, average=ClusterSet.centroids, init='random', plot=False):
```

—> And I implemented the “TODO” section as follows…

```
### ========== 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(...).

k_clusters = ClusterSet()

# initialize center for each cluster
centers = None
if (init == 'cheat'):
  centers = cheat_init(points)
else:
  centers = random_init(points, k)

iteration = 0
while (1):
  # create new cluster assignments
  clusters = [[] for i in range(k)]
  for p in points:
    dist_to_centers = [p.distance(p2) for p2 in centers]
    cluster_assignment = dist_to_centers.index(min(dist_to_centers))
    clusters[cluster_assignment].append(p)

  # use assignments to create new ClusterSet object
  new_clusters = ClusterSet()
  for c in clusters:
    new_clusters.add(Cluster(c))

  # update the cluster centers
  centers = average(new_clusters)

  # plot clusters with corresponding averages, if desired
  iteration += 1
  if (plot):
    plot_clusters(new_clusters, "clusters after iteration " + str(iteration), \
                  average)

  # stop if clustering no longer changes
  if k_clusters.equivalent(new_clusters):
    break
  else:
    k_clusters = new_clusters

return k_clusters, iteration
### ========== TODO : END ========== ###
```

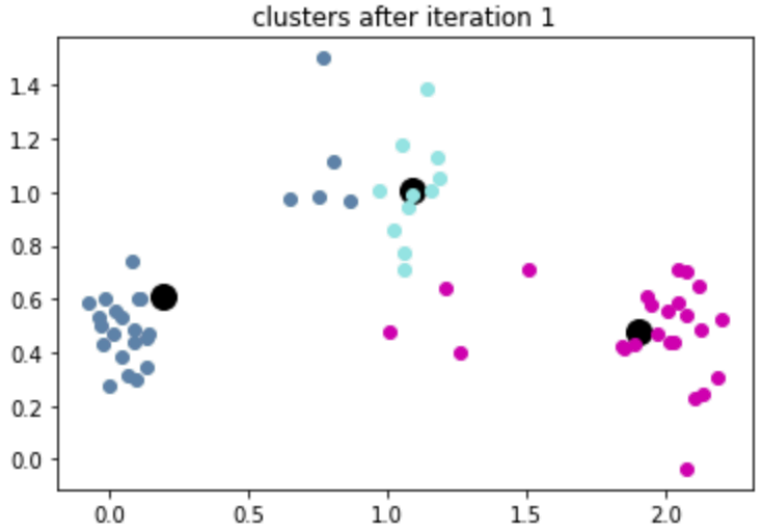
(d) Problem 4.2(d)

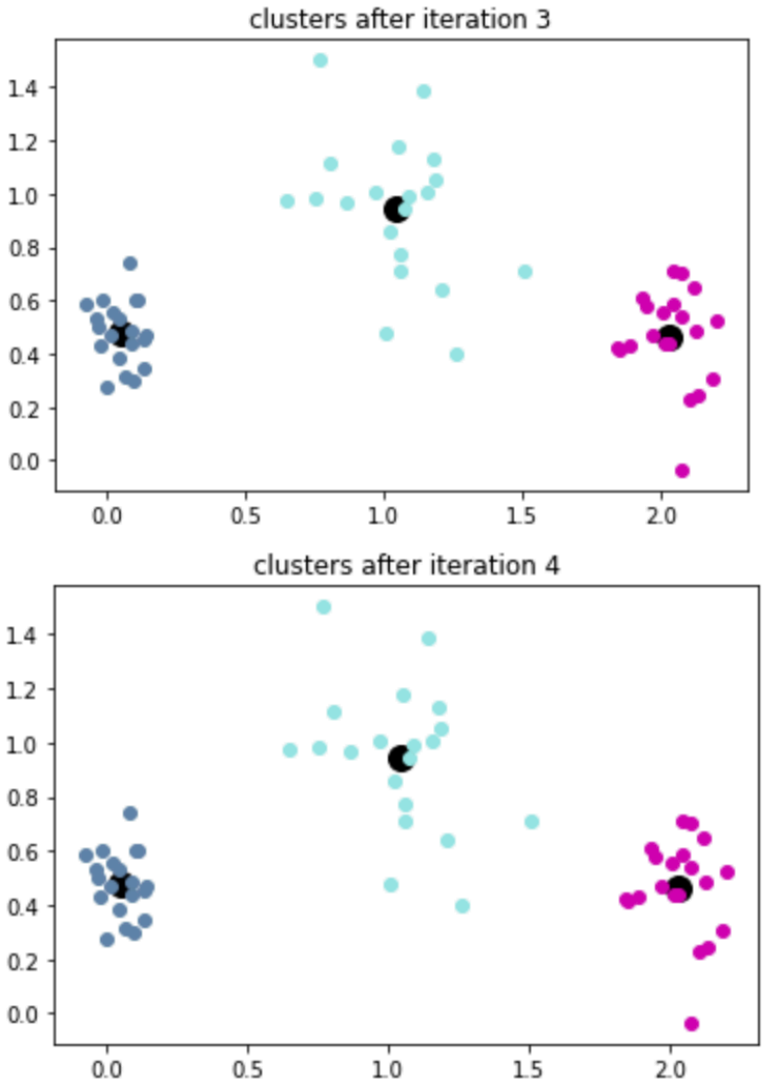
Solution:

Code in “main(…)”:

```
# PART 2D
print("running part 2d...")
np.random.seed(1234)

# generate three clusters, each containing 20 points
points = generate_points_2d(20)

# run kMeans with random initialization and generate plots
kMeans(points, 3, plot=True)
# END PART 2D
```

"Include plots for kMeans cluster assignments and cluster centers for each iteration"…





(e) Problem 4.2(e)

Solution:

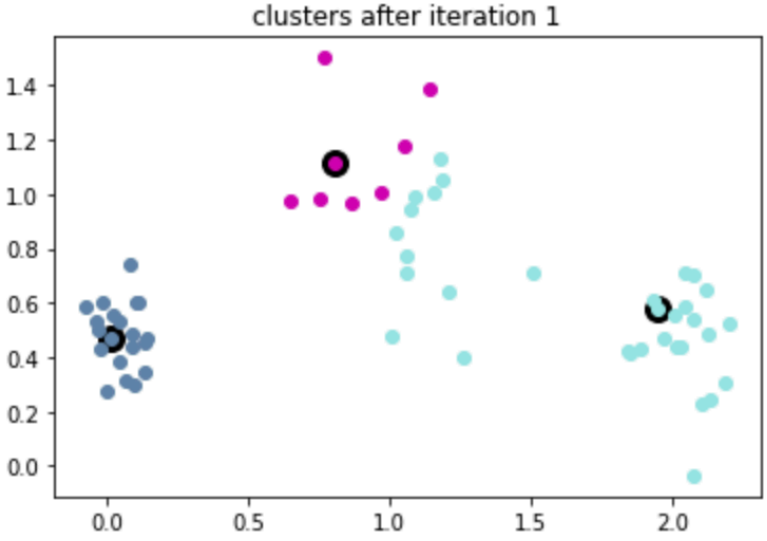
Code in “kMedoids(…)”:

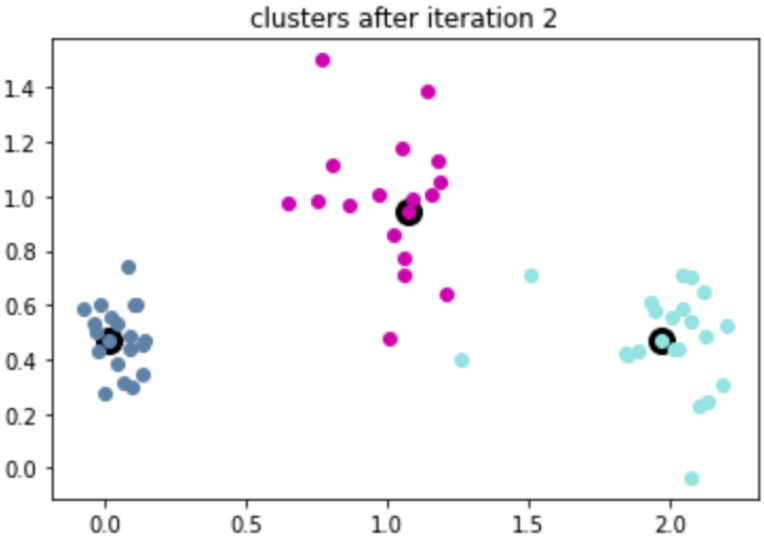
```
### ========== TODO : START ========== ###
# part 2e: implement
k_clusters, iterations = kMeans(points, k, average=ClusterSet.medoids, init=init, plot=plot)
return k_clusters, iterations
### ========== TODO : END ========== ###
```

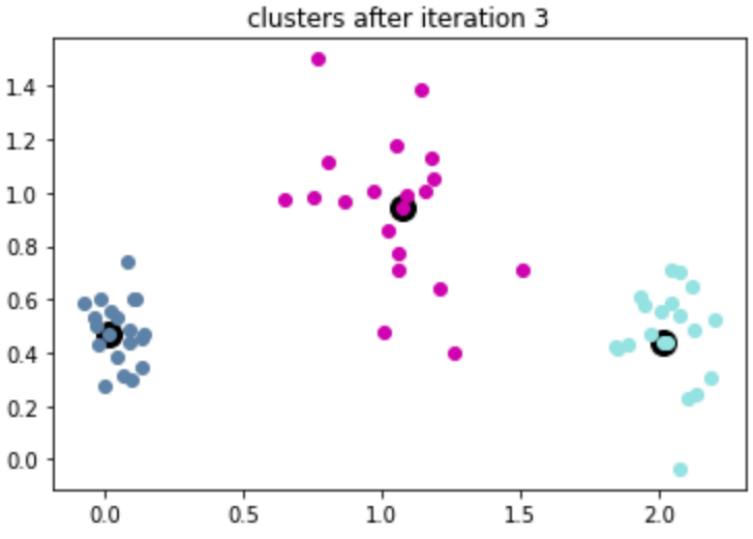
Code in “main(…)”:

```
# PART 2E
print("running part 2e...")
kMedoids(points, 3, plot=True)
# END PART 2E
```

“Include plots for k-medoids clustering for each iteration”:







(f) Problem 4.2(f)

Solution:

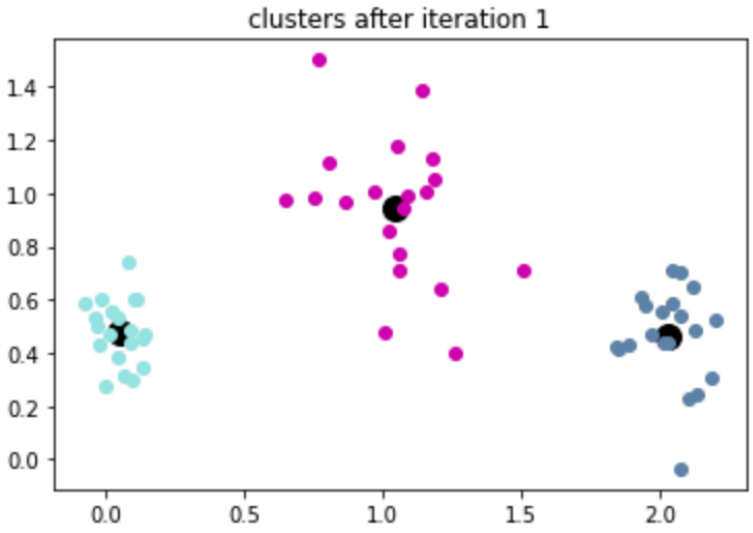
Code in “cheat_init(…)”:

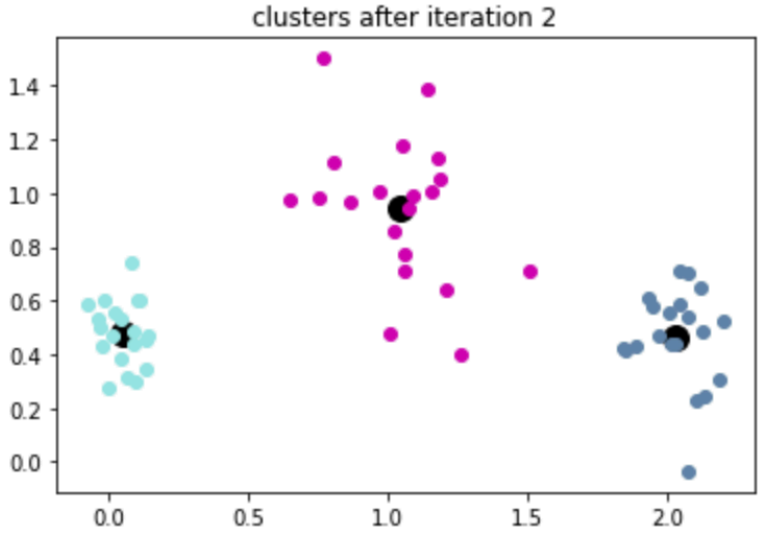
```
### ========== TODO : START ========== ###
# part 2f: implement

# find the value of k
all_labels = [p.label for p in points]
k = len(np.unique(all_labels))

# group points according to their labels
clusters = [[] for i in range(k)]
for p in points:
  clusters[p.label - 1].append(p)

# make a ClusterSet with our new clusters
set_of_clusters = ClusterSet()
for c in clusters:
  set_of_clusters.add(Cluster(c))

# return a list containing the medoids of each cluster
initial_points = set_of_clusters.medoids()
return initial_points
### ========== TODO : END ========== ###
```

Code in “main(…)”:

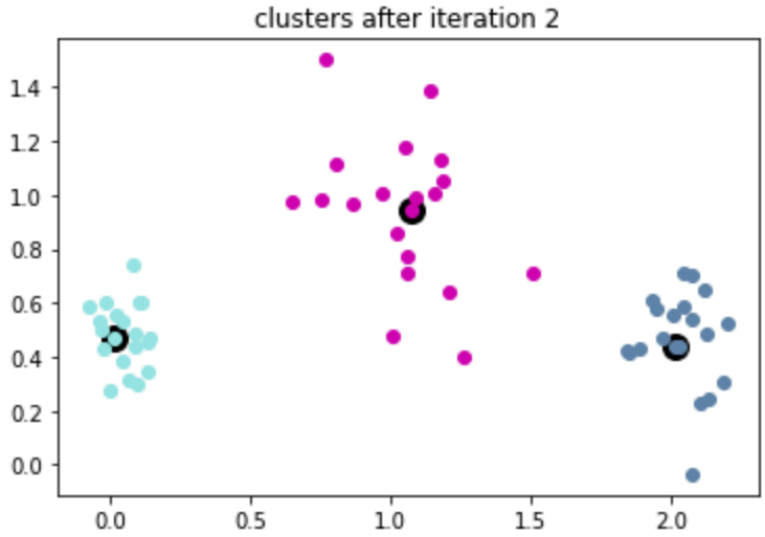
```
# PART 2F
print("running part 2f...")
print("'cheat_init' with KMeans...")
kMeans(points, 3, init='cheat', plot=True)
print("'cheat_init' with KMedoids...")
kMedoids(points, 3, init='cheat', plot=True)
# END PART 2F
```

"Include plots for k-means and k-medoids for each iteration"…
—> for k-means:





—> for k-medoids:



"Compare clustering by initializing using cheat_init(…)"…

Using "cheat_init(…)" leads to fewer iterations to achieve the same final clustering result as "random_init(…)", with the number of iterations dropping from 4 to 2 for both k-means and k-medoids when we use "cheat_init(…)" instead of "random_init(…)".

This makes sense because the cluster centers are initialized extremely close to (if not exactly the same as) the cluster centers in the final iteration. Thus, the cluster assignments change extremely minimally (if at all) before the final clustering is achieved, reducing the need for further iterations before reaching the final cluster state.

4.3: Clustering Faces

(a) Problem 4.3(a)

Solution:

Code in “main(…)”:

```
### ========== TODO : START ========== ###
# part 3a: cluster faces
print("running part 3a...")
np.random.seed(1234)

# make new image dataset and translate these images to labeled points
X1, y1 = util.limit_pics(X, y, [4, 6, 13, 16], 40)
points = build_face_image_points(X1, y1)

# run KMeans 10 times and report average, min, and max score
kmeans_scores = []
kmeans_iters = []
kmed_scores = []
kmed_iters = []
for i in range(10):
  # run kmeans and record results
  k_clusters, iters = kMeans(points, 4)
  kmeans_scores.append(k_clusters.score())
  kmeans_iters.append(iters)

  # run kmedoids and record results
  k_clusters, iters = kMedoids(points, 4)
  kmed_scores.append(k_clusters.score())
  kmed_iters.append(iters)

# report average, min, max score for kmeans, and its average iteration count
print("KMeans scores on image dataset:")
print("average = %0.3f\nmin = %0.3f\nmax = %0.3f\niterations = %0.3f"\
      % (np.mean(kmeans_scores), min(kmeans_scores), max(kmeans_scores), \
         np.mean(kmeans_iters)))

# report average, min, max score for kmedoids, and its average iteration count
print("KMedoids scores on image dataset:")
print("average = %0.3f\nmin = %0.3f\nmax = %f\niterations = %0.3f"\
      % (np.mean(kmed_scores), min(kmed_scores), max(kmed_scores), \
         np.mean(kmed_iters)))
```

Output of the code in “main(…)”:

```
running part 3a...
KMeans scores on image dataset:
average = 0.617
min = 0.550
max = 0.775
iterations = 7.700
KMedoids scores on image dataset:
average = 0.632
min = 0.575
max = 0.725000
iterations = 3.900
```

From the above output, the average, min, and max performance are:

	average	min	max
k-means	0.617	0.550	0.775
k-medoids	0.632	0.575	0.725

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

On average, k-medoids performs better in the sense that it has a higher average cluster purity score across all of its clusters. K-medoids also has a higher minimum score, as shown in the table above, which is a further indication that it performs better than k-means in most cases.

Interestingly, k-medoids has a lower maximum score, which indicates that k-means can perform better than k-means in some cases, where the relative performance depends on the initialization.

From the output from the code in "main(…)", we see that k-medoids converges to a solution in fewer iterations than k-medoids in the average case. K-means took an average of 7.7 iterations to converge, while k-medoids took an average of 3.9 iterations across the 10 trials.

Therefore, in the average case, k-medoids had a faster runtime.

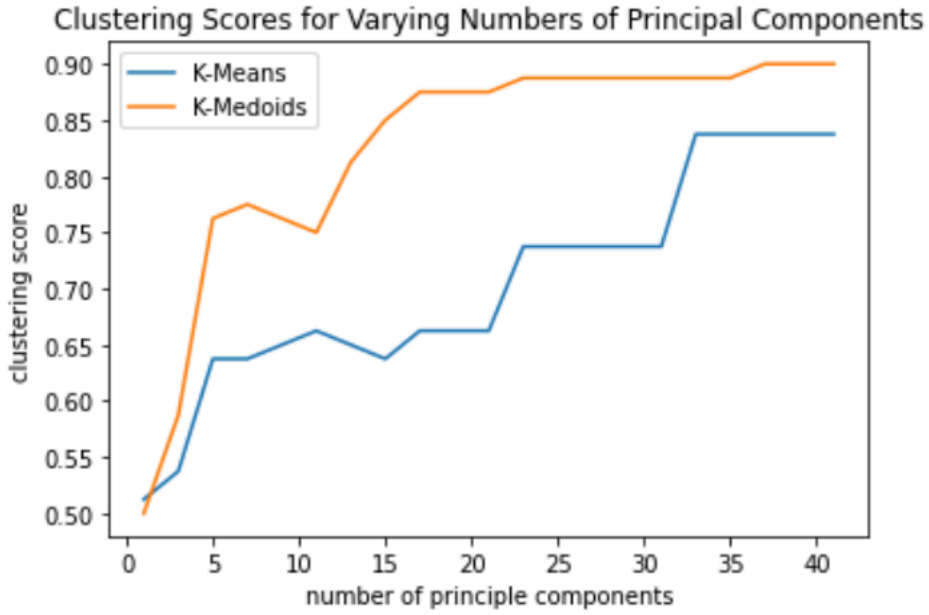
(b) Problem 4.3(b)

Solution:

Code in "main(…)":

```
# part 3b: explore effect of lower-dimensional representations on clustering performance
print("running part 3b...")
np.random.seed(1234)

# create another dataset
X1, y1 = util.limit_pics(X, y, [4, 13], 40)

# compute principle components for entire dataset
U, mu = PCA(X)

n_components = [i for i in range(1,42,2)]
kmeans_scores = []
kmed_scores = []
for l in n_components:
  # project newly generated dataset to lower dimension and generate points
  projected_dataset, Ul = apply_PCA_from_Eig(X1, U, l, mu)
  points = build_face_image_points(projected_dataset, y1)

  # compute scores for each clustering algorithm
  k_clusters, iters = kMeans(points, 2, init='cheat')
  kmeans_scores.append(k_clusters.score())
  k_clusters, iters = kMedoids(points, 2, init='cheat')
  kmed_scores.append(k_clusters.score())

# plot clustering score versus number of components for each algorithm
plt.figure()
plt.plot(n_components, kmeans_scores, label="K-Means")
plt.plot(n_components, kmed_scores, label="K-Medoids")
plt.title('Clustering Scores for Varying Numbers of Principal Components')
plt.xlabel('number of principle components')
plt.ylabel('clustering score')
plt.legend()
plt.show()
```

"plot the clustering score versus the number of components for each clustering algorithm"…



"Discuss the results in a few sentences":

For the same initialization, k-medoids achieved a higher final score for every number of principle components (except for $l = 1$), thus indicating that k-medoids likely exhibits better performance on this dataset.

In general, the clustering score increases as the number of principle components increases, which makes sense because including more principle components captures a higher percentage of the variance from the initial dataset, thus making it easier to distinguish points that belong to different classes.

The curve increases rapidly for low values of l, then increases more slowly as l increases. This makes sense because the eigenvectors are sorted by eigenvalues, so the initial eigenvectors that we include capture the most variance in the dataset and make the biggest impact on the ability to cluster effectively. The eigenvalues we add at higher values of l capture lower variance and thus do not improve the clustering scores as much.

(c) Problem 4.3(c)

Solution:

Code in “main(…)”:

```
# part 3c: determine ``most discriminative'' and ``least discriminative'' pairs of images
print("running part 3c...")
np.random.seed(1234)

pair_scores = []
im_pairs = []
for im1 in range(19):
  for im2 in range(im1+1,19):
    # record the pairing
    im_pairs.append((im1, im2))

    # get dataset with only these 2 images
    X1, y1 = util.limit_pics(X, y, [im1, im2], 40)
    points = build_face_image_points(X1, y1)

    # get kmedoids score
    k_clusters, iters = kMedoids(points, 2, init='cheat')
    pair_scores.append(k_clusters.score())

# find pair that clustering can discriminate well
index = pair_scores.index(max(pair_scores))
most = im_pairs[index]
print("most discriminative pair: ", most)
plot_representative_images(X, y, [most[0], most[1]], \
                           title='Most Discriminative Pair')

# find difficult pair
index = pair_scores.index(min(pair_scores))
least = im_pairs[index]
print("least discriminative pair: ", least)
plot_representative_images(X, y, [least[0], least[1]], \
                           title='Least Discriminative Pair')
```

"Describe your methodology":

My implementation loops over all $\binom{19}{2} = 171$ pairs of images and creates a new dataset containing 40 samples of each of the 2 images. It then performs k-medoids on the pair and records the resulting clustering score.

After considering all pairs, it selects the pair with the highest score as the most discriminative pair and the pair with the lowest score as the least discriminative pair.

I used k-medoids because it achieved better performance metrics in the tests performed in parts 4.3(a) and 4.3(b). I used "cheat_init(…)" because it indicates how well the clustering algorithms could perform if they have a decent initialization state.

"Report the two pairs in your writeup":
—> most discriminative pair: 9 & 16


—> least discriminative pair: 4 & 5


“Comment briefly on the results”:

The most discriminative pair had vastly different skin tones, which means that there was likely a large difference between the images for each pixel value (i.e. feature value), which allowed the clustering algorithm to separate them into different clusters more easily and thus discriminate them easily.

The least discriminative pair had very similar skin tones and similar facial structures, which meaning that there was likely a lesser difference in each pixel value (i.e. feature value), which made them more difficult to separate into different clusters.