Ethan Wong
3/4/22

CS M146 PS4

1. $(h_t^*(x), \beta_t^*) = \underset{(h_t(x), \beta_t)}{\operatorname{argmin}} \sum_n w_t(n) e^{-y_n \beta_t h_t(x_n)}$

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

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

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

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

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

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

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

b) • Training set is linearly separable
• hard-margin linear support vector machine (no slack)

With this setup, we would produce a linear separator that correctly classifies all of the data points. Thus, there would be no error → $\epsilon_1 = 0$. This means that as $\epsilon_t \to 0$, $\beta_t \to \infty$. Therefore, in the first boosting iteration $\beta_1 = \infty$.

2. a) $k=3 \quad x_1=1 \quad x_2=2 \quad x_3=5 \quad x_4=7$

We would get the optimal clustering by doing the following:

- $C_1 = \{x_1, x_2\}$ - $C_2 = \{x_3\}$ - $C_3 = \{x_4\}$

This results in the following centroids:

$\nu_1 = \frac{1+2}{2} = \frac{3}{2} \qquad \nu_2 = 5 \qquad \nu_3 = 7$

And the objective can be found by doing:

$$\text{Obj} = (1-\tfrac{3}{2})^2 + (2-\tfrac{3}{2})^2 + 0^2 + 0^2$$
$$= (\tfrac{1}{2})^2 + (\tfrac{1}{2})^2 = \tfrac{1}{2}$$
$$= 0.5$$

$\boxed{\nu_1 = 1.5 \quad \nu_2 = 5 \quad \nu_3 = 7 \quad \text{Objective} = 0.5}$

b) Lloyd's algorithm organizes data points into clusters, then reassigns ν_k according to the average of the data points in each of the clusters. Let's say that we initialize Lloyd's algorithm with the following:

- $C_1 = \{x_1\}$ - $C_2 = \{x_2\}$ - $C_3 = \{x_3, x_4\}$

Lloyd's algorithm would now compute the centroids using the average of the data points according to our initialization. The last cluster has a different set of 2 points. This means the algorithm should terminate at a local minima.

- $\nu_1 = 1$ - $\nu_2 = 2$ - $\nu_3 = \frac{5+7}{2} = 6$

$$\text{Obj} = 0^2 + 0^2 + (5-6)^2 + (7-6)^2 = 2$$

Based on these calculations, we see that the assignment of $A(x_1)=1$, $A(x_2)=2$, $A(x_3)=3$, $A(x_4)=3$ gives us a suboptimal solution when compared to the solution in part A. The objective in part A is 0.5 as compared to the objective of 2 in part B.

3. a) $\ell(\theta) = \sum_K \sum_n \gamma_{nK} \log w_K + \sum_K \{ \sum_n \gamma_{nK} \log N(x_n | \mu_K, \Sigma_K) \}$

$\nabla_{\mu_j} \ell(\theta) = \frac{\partial \ell(\theta)}{\partial \mu_K} \sum_K \sum_n \gamma_{nK} \log \left(\frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x_n - \mu_j)^2}{2\sigma^2}} \right)$

Note: $\sigma^2 = \Sigma_j$

$$= \frac{\partial \ell(\theta)}{\partial \mu_K} \sum_K \sum_n \gamma_{nK} \left[\left(\frac{-(x_n - \mu_j)^2}{2\sigma^2} \right) - \log \left(\frac{1}{\sigma\sqrt{2\pi}} \right) \right]$$

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

$$= \sum_n \gamma_{nj} \frac{(x_n - \mu_j)}{\sigma^2}$$

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

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

$$\boxed{\nabla_{\mu_j} \ell(\theta) = \sum_n \frac{\gamma_{nj}}{\Sigma_j} (x_n - \mu_j)}$$

b) Set the above solution (the gradient) to zero

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

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

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

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

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

$$\boxed{\mu_j = \frac{1}{\sum_n \gamma_{nj}} \sum_n \gamma_{nj} x_n \quad \checkmark}$$

c) $w_j = \frac{\sum_n \gamma_{nj}}{\sum_j \sum_n \gamma_{nj}}$ $\quad \nu_j = \frac{\sum_n \gamma_{nj} x_n}{\sum_n \gamma_{nj}}$

$\sum_j \sum_n \gamma_{nj} = 0.2 + 0.2 + 0.8 + 0.9 + 0.9 + 0.8 + 0.8 + 0.2 + 0.1 + 0.1 = 5$

$w_1 = \frac{\sum_n \gamma_{n1}}{5} = \frac{0.2 + 0.2 + 0.8 + 0.9 + 0.9}{5} = \frac{3}{5} = 0.6$

$w_2 = \frac{\sum_n \gamma_{n2}}{5} = \frac{0.8 + 0.8 + 0.2 + 0.1 + 0.1}{5} = \frac{2}{5} = 0.4$

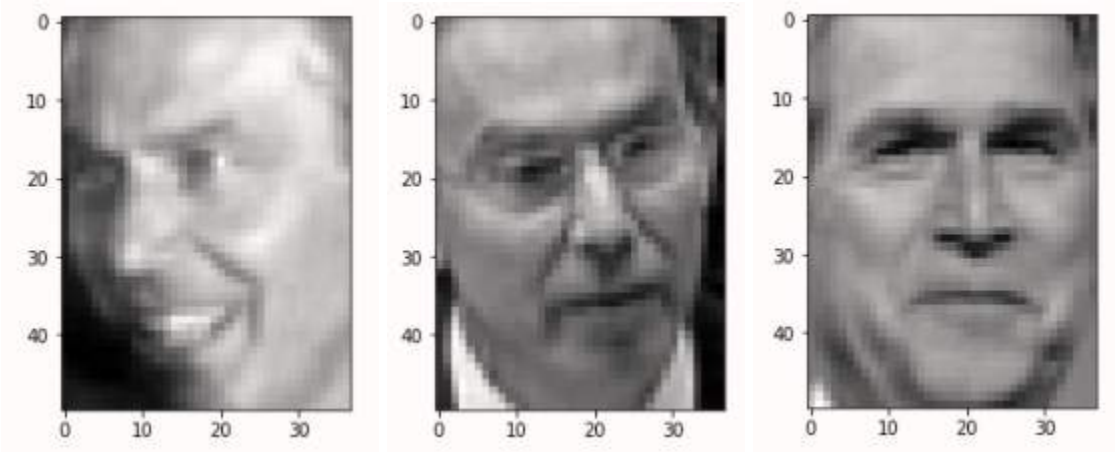
$\nu_1 = \frac{\sum_n \gamma_{n1} x_n}{\sum_n \gamma_{n1}} = \frac{(0.2 \cdot 5) + (0.2 \cdot 15) + (0.8 \cdot 25) + (0.9 \cdot 30) + (0.9 \cdot 40)}{0.2 + 0.2 + 0.8 + 0.9 + 0.9} = 29$

$\nu_2 = \frac{\sum_n \gamma_{n2} x_n}{\sum_n \gamma_{n2}} = \frac{(0.8 \cdot 5) + (0.8 \cdot 15) + (0.2 \cdot 25) + (0.1 \cdot 30) + (0.1 \cdot 40)}{0.8 + 0.8 + 0.2 + 0.1 + 0.1} = 14$

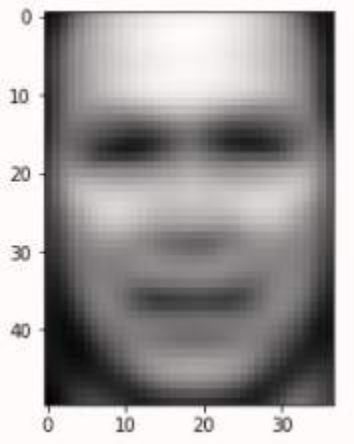
$w_1 = 0.6 \quad w_2 = 0.4$

$\nu_1 = 29 \quad \nu_2 = 14$

4.1 A)



Average Face:

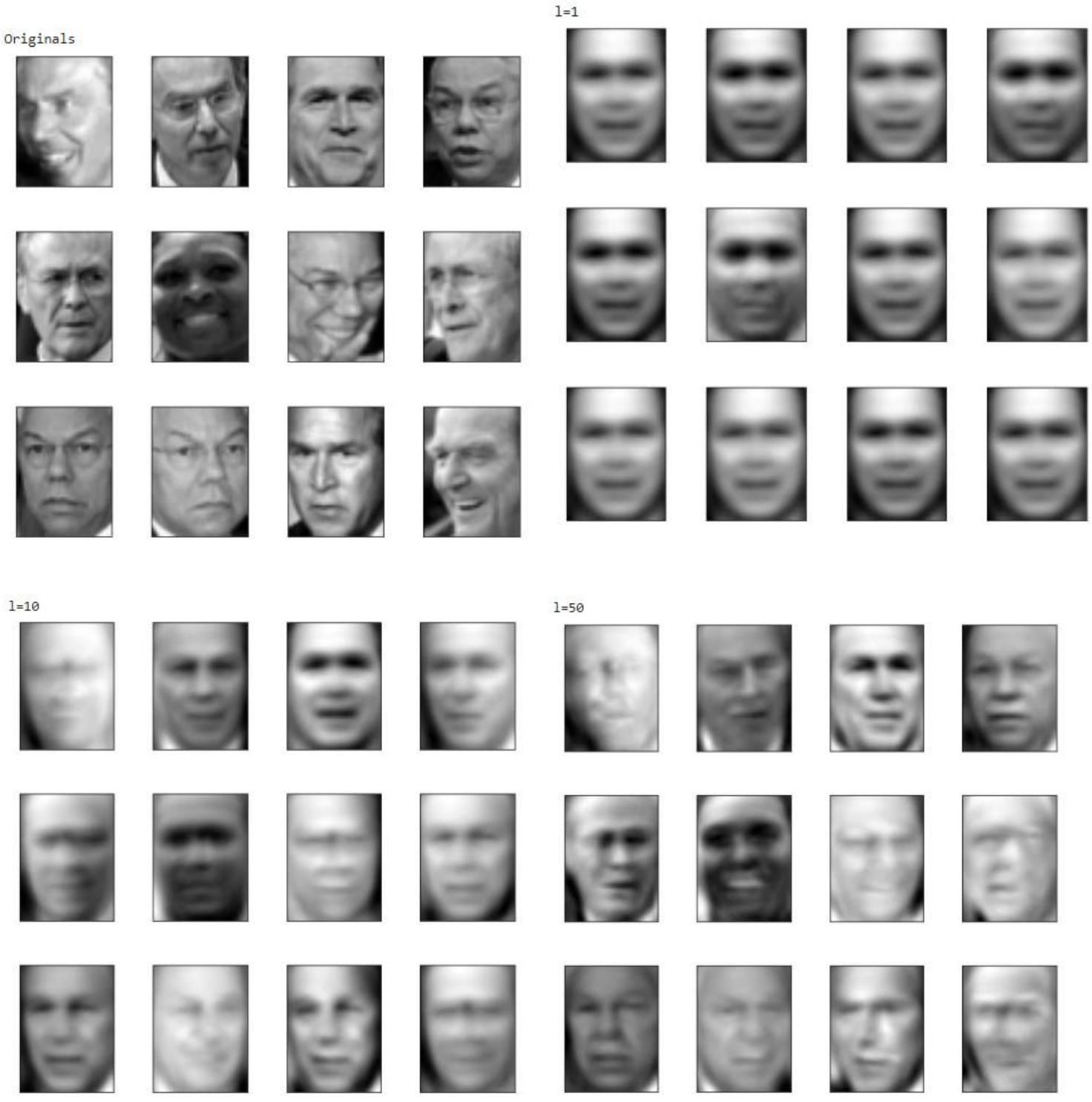


The average face is shown above. It still vaguely resembles a human face looking forward, but a lot of the features are quite blurred. We can see a clear image of the eyes, nose, and mouth but there are little to no details. Because the details all get blurred together, the eyes and mouth look very dark and eerie. Overall though, the average face looks quite good and representative of a human face.

4.1 B)

These images all have some distinct features that make them easy to recognize and differentiate from each other. Some of the features that are easy to discern include skin tone and facial hair. These images were probably chosen as the top eigenfaces because their distinct features make them stand out from the blur of the rest of the dataset.

4.1 C)



l=100

l=500

l=1288

Using all these different values for “l” show us a few different things. First, using smaller values of l (lower dimensions) is hard because most faces deduce to looking quite similar. As we increase the dimensions, we observe more distinct features that make it difficult to recognize the same people just because of a different facial expression. However, higher dimensions also show that faces start to look similar. For instance, we see there is more resemblance to the original as we increase the value for l. From this little experiment we see that choosing an appropriate dimension is vital as it allows us to use a correct amount of principal components for accurate classifications.

Code for 4.1 A B C)

```
### ========== TODO : START ========== ###
# part 1: explore LFW data set
# part 1a
X, y = get_lfw_data()
num = 12
show_image(X[0])
show_image(X[1])
show_image(X[2])
xAverage = np.mean(X, axis = 0)
show_image(xAverage)

# part 1b
U, mu = PCA(X)
plot_gallery([vec_to_image(U[:,i]) for i in range(num)])

# part 1c
num_components = [1, 10, 50, 100, 500, 1288]
print("Originals")
plot_gallery(X[:num], title = f'First 12 images (original-dimension)')
for l in num_components:
  print(f'l={l}')
  Z, Ul = apply_PCA_from_Eig(X, U, l, mu)
  new = reconstruct_from_PCA(Z, Ul, mu)
  plot_gallery(new[:num], title=f'First 12 images ({l}-dimension)')
### ========== TODO : END ========== ###
```

4.2 A)

It would be a bad idea to minimize over μ, c, and k. This is because we could set k=n such that n is the number of data points we have. This would cause $\mu_i = x_i$ as each point would just be assigned to its own cluster. We would then see that our objective (J) would become 0. As a result, $c^{(i)} = i$. After all of this, we can see that this defeats the purpose of computing the centers of each cluster to derive an approximate classification for the dataset. After all, our goal was to group data points based on their similar features in order to produce a predictive model.

4.2 B)

For Cluster Class:

```
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)
    l = []
    attr = np.array([p.attrs for p in self.points])
    numPoints = len(self.points)
    pos = np.sum(attr, axis = 0) / numPoints

    for p in self.points:
      l.append(p.label)

    clusterLabel, num = stats.mode(l)
    centroid = Point('C', clusterLabel[0], pos)
    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
    dist = {}

    for p in self.points:
      dist[p] = []
      for o in self.points:
        if o != p:
          dist[p].append(p.distance(o))

    for k, v in dist.items():
      avg = np.mean(v)
      dist[k] = avg

    medoid = min(dist, key = dist.get)
    return medoid
    ### ========== TODO : END ========== ###
```

For ClusterSet class:

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

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

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

    return centroids
    ### ========== TODO : END ========== ###


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

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

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

    return medoids
    ### ========== TODO : END ========== ###
```

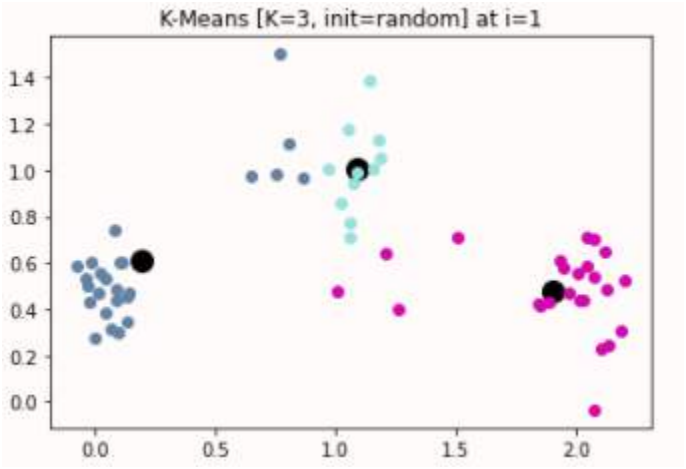
4.2 C)

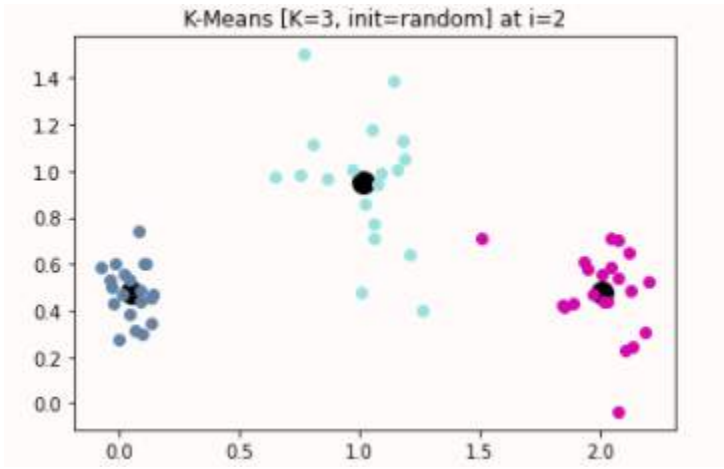
```
def random_init(points, k) :
    """
    Randomly select k unique elements from points to be initial cluster centers.

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

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

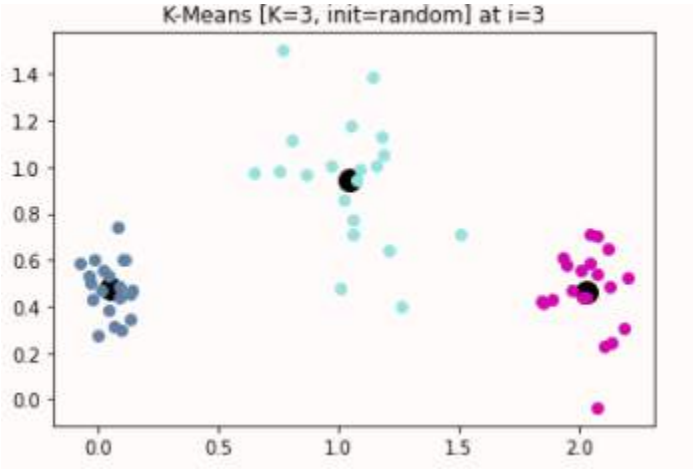
```
def kMeans(points, k, init='random', plot=False) :
    """
    Cluster points into k clusters using variations of k-means algorithm.

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

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

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

    return kAverages(points, k, ClusterSet.centroids, init = init, plot = plot)
    ### ========== TODO : END ========== ###
```

```python
def kAverages(points, k, average, init = 'random', plot = True):
    kClusters = ClusterSet()
    if init == 'random':
        MU = random_init(points, k)
    elif init == 'cheat':
        MU = cheat_init(points)

    x = 1
    while True:
      clusters = {}
      for m in MU:
        clusters[m] = Cluster([])

      for p in points:
        dist = [p.distance(m) for m in MU]
        index = np.argmin(dist)
        mu2 = MU[index]
        clusters[mu2].points.append(p)

      newCluster = ClusterSet()
      for m, c in clusters.items():
        newCluster.add(c)

      if (kClusters.equivalent(newCluster)) and len(kClusters.members) > 0:
        if average != ClusterSet.centroids:
          name = "K-Medoids"
        else:
          name = "K-Means"
        break

      MU = average(newCluster)
      kClusters = newCluster

      if plot:
        if average != ClusterSet.centroids:
          plot_clusters(newCluster, f"K-Medoids [K={k}, init={init}] at i={x}", average)
        else:
          plot_clusters(newCluster, f"K-Means [K={k}, init={init}] at i={x}", average)

      x += 1
    return kClusters
```

4.2 D)

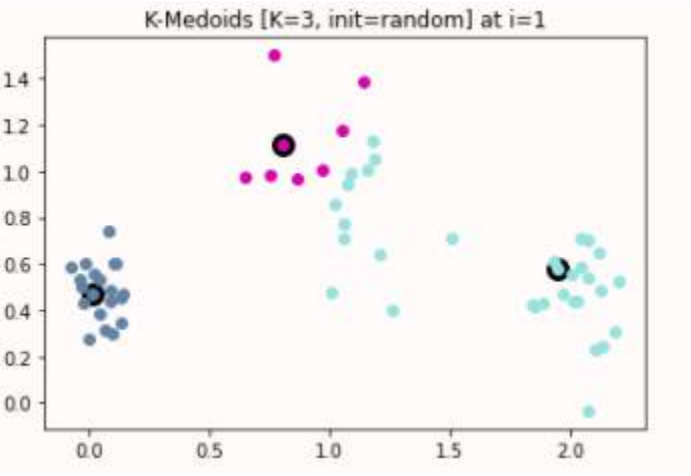


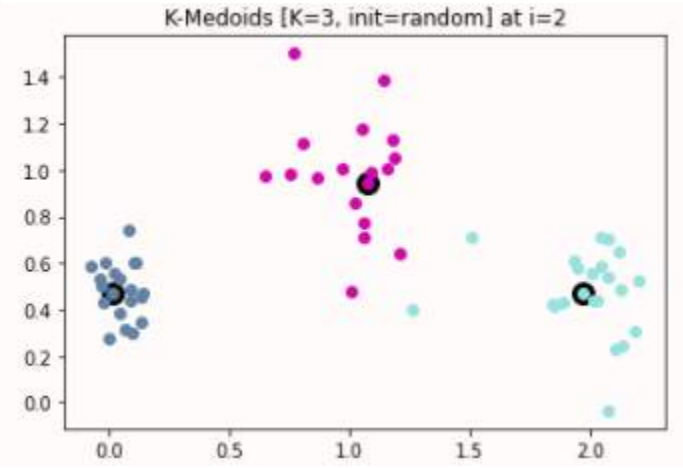


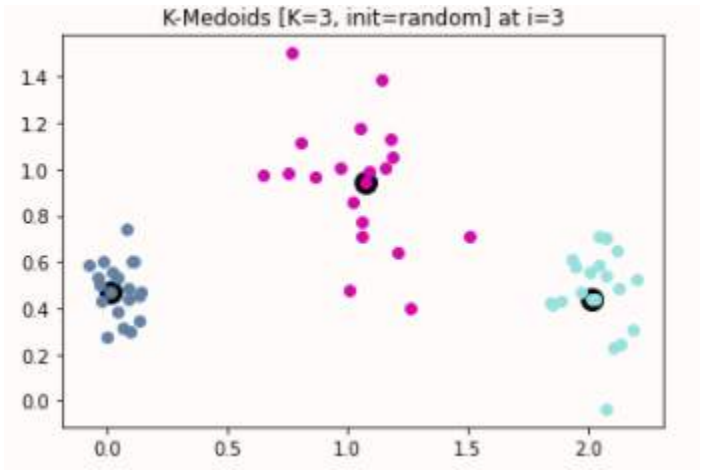


K-means was able to successfully classify the toy dataset a total of four iterations using random initialization. In reality it only needs three iterations, but an extra fourth iteration is run just to check if there are any changes to the clusters. There were no changes, so it stopped after the fourth iteration. As the iterations progress, we observe how the centroid for each cluster actually moves to the actual center of each cluster.

4.2 E)



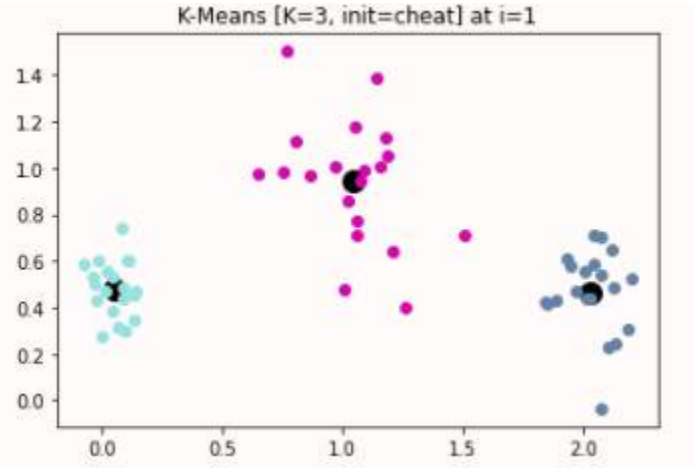


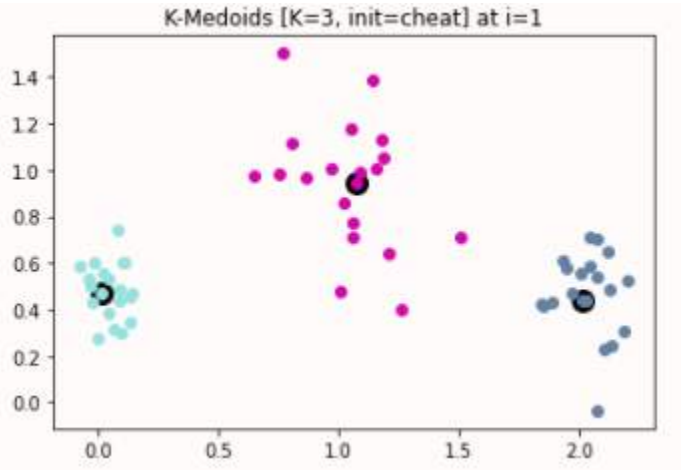


Just like in part D, we see that K-Medoids takes a total of four iterations to successfully classify the dataset. K-Medoids and K-Means happened to take the same number of iterations for this dataset. (There is still the same case with the extra fourth iteration for K-Medoids that was explained earlier). Although the same final results were achieved, the clusters were formed quite differently with the two approaches. For example, the centroids start at different locations before making their way to the actual center of the cluster.

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

4.2 F)





When we use cheat_init, both the K-Medoids and K-Means algorithm are able to get the same result with the centroids at the actual center of the clusters in just 2 iterations. This is considerably fewer iterations than without using cheat_init as we can cut the number of iterations in half.

Main Code for 4.2 D-F

```
### ========== TODO : START ========== ###
# part 2d-2f: cluster toy dataset
np.random.seed(1234)
N = 20
toy_dataset = generate_points_2d(N)
kMeans(toy_dataset, 3, init = 'random', plot = True)
kMedoids(toy_dataset, 3, init = 'random', plot = True)
kMeans(toy_dataset, 3, init = 'cheat', plot = True)
kMedoids(toy_dataset, 3, init = 'cheat', plot = True)
### ========== TODO : END ========== ###
```

cheat_init code

```
### ========== TODO : START ========== ###
# part 2f: implement
labels = set()
groups = {}
clusters = []

for p in points:
  labels.add(p.label)
for l in labels:
  groups[l] = []
for p in points:
  groups[p.label].append(p)
for k, v in groups.items():
    cluster = Cluster(v)
    clusters.append(cluster)

initial_points = [cluster.medoid() for cluster in clusters]
return initial_points
### ========== TODO : END ========== ###
```

4.3 A)

	average	min	max
kMeans	0.6175	0.55	0.775
kMedoids	0.6325	0.575	0.725

K-Means demonstrated a greater max than K-Medoids, but K-Means had a smaller average and min than K-Medoids. There are very slight differences between the values of the two approaches, likely due to the sensitivity that K-Means displays to outliers. In terms of runtime, it seems that K-Medoids is quicker as most of its runs finish within only two iterations and the slowest was 5 iterations. On the other hand, the number of iterations it took K-Means to finish ranged anywhere from less than 5 to over 12.

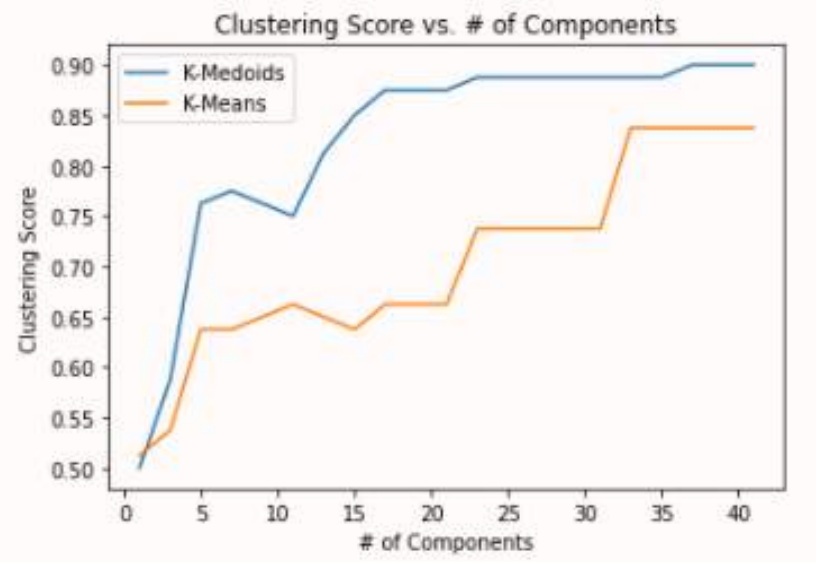
```
# part 3a: cluster faces
kMeanScores = []
kMedoidScores = []
np.random.seed(1234)
X1, y1 = util.limit_pics(X, y, [4, 6, 13, 16], 40)
points = build_face_image_points(X1, y1)

for i in range(10):
  mean = kMeans(points, 4, init = 'random')
  kMeanScores.append(mean.score())
  medoid = kMedoids(points, 4, init = 'random')
  kMedoidScores.append(medoid.score())

print('\t average min\t max')
print(f'kMeans\t {round(sum(kMeanScores) / len(kMeanScores), 4)}\t {round(min(kMeanScores), 4)}\t {round(max(kMeanScores), 4)}')
print(f'kMedoids {round(sum(kMedoidScores) / len(kMedoidScores), 4)}\t {round(min(kMedoidScores), 4)}\t {round(max(kMedoidScores), 4)}')
```

4.3 B)

Simply put, this graph (see below) shows us that K-Medoids consistently performs better than K-Means with more principal components used. The plot for K-Medoids shows that there is a dramatic increase in clustering score as the number of components increases from 0 to 5. There is also another significant increase in the clustering score as the number of components increases from 10 to 15. Once K-Medoids reached 15 components, the growth levels off as it has approached a clustering score of nearly 1. K-Medoids might have better performance than K-Means because it uses an actual data point from the cluster while K-Means does not.

As the number of components increases, the clustering score of K-Means doesn't improve as dramatically as we saw with K-Medoids. There's a more gradual increase in clustering score as more components are added, and there is eventually a plateau at a clustering score of 0.85 from 35-40 components. Overall, we learn that increasing the number of components can increase the clustering score for both K-Medoids and K-Means as we can generalize similarities between data points better.

```
# part 3b: explore effect of lower-dimensional representations on clustering performance
kMeanScores = []
kMedoidScores = []
np.random.seed(1234)
U, mu = PCA(X)
K = 2
X1, y1 = util.limit_pics(X, y, [4, 13], 40)
components = np.arange(1, 42, K)

for i in components:
  Z, Ul = apply_PCA_from_Eig(X1, U, i, mu)
  X_rec = reconstruct_from_PCA(Z, Ul, mu)
  points = build_face_image_points(X_rec, y1)
  meanResult = kMeans(points, K, init = 'cheat')
  medoidResult = kMedoids(points, K, init = 'cheat')
  kMeanScores.append(meanResult.score())
  kMedoidScores.append(medoidResult.score())

plt.figure()
plt.plot(components, kMedoidScores, label = 'K-Medoids')
plt.plot(components, kMeanScores, label = 'K-Means')
plt.ylabel("Clustering Score")
plt.xlabel("# of Components")
plt.title("Clustering Score vs. # of Components")
plt.legend()
plt.show()
```



4.3 C)

In order to find a pair that could discriminate very well and a pair that it finds very difficult, I used two for-loops to iterate through all the combinations of the pairings. To do this, I used K-Medoids with cheat initialization enabled (we saw previously that this yields the best results in the fewest iterations). In every iteration, I build a subset of 40 pictures for each individual and then build a new set of points. Finally, I compute the score of K-Medoid for that given pair and keep track of the pairs that are doing well and the ones that are not.

```
# part 3c: determine ``most discriminative'' and ``least discriminative'' pairs of images
np.random.seed(1234)
classes = set(y)
currMax = 0
maxPair = [0, 0]
currMin = 999999999999999
minPair = [0, 0]

for i in range(len(classes)):
  for j in range(len(classes)):
    if i != j:
      X1, y1 = util.limit_pics(X, y, [i, j], 40)
      points = build_face_image_points(X1, y1)
      cluster = kMedoids(points, 2, init = 'cheat')
      s = cluster.score()
      if s < currMin:
        minPair = [i,j]
        currMin = s
      elif s > currMax:
        maxPair = [i,j]
        currMax = s

print(minPair)
print(maxPair)
plot_representative_images(X, y, minPair)
plot_representative_images(X, y, maxPair)
```

Pair that it can discriminate well:

This pair is probably easier for the algorithm to discriminate between because these two individuals look very different. They have different skin tones, different nose shapes, different-looking mouths, and different-shaped eyes. Visually, almost everything about them is different.

Pair that it can’t discriminate well:

This pair is probably difficult for the algorithm to discriminate between because these two individuals look very similar. They have the same skin tone, similar noses, and similar eye shapes. They share a lot of the same features, so it is understandable that the algorithm would have a hard time with these two.

PCA and k-means

Setting up

```
"""
Author      : Yi-Chieh Wu, Sriram Sankararaman
"""

# numpy and scipy libraries
import numpy as np
from scipy import stats

# matplotlib libraries
import matplotlib.pyplot as plt
import collections


# To add your own Drive Run this cell.
from google.colab import drive
drive.mount('/content/gdrive')
```

```
     Mounted at /content/gdrive
```

```
import sys
# Change the path below to the path where your folder locates
# where you have util.py
### ========== TODO : START ========== ###
sys.path.append('/content/gdrive/My Drive/Colab Notebooks/pset4')
### ========== TODO : START ========== ###



import util
from util import *
```

Point, Cluster and Set of Clusters classes

```
################################################################################
# classes
################################################################################

class Point(object) :
```

```
        def __init__(self, name, label, attrs) :
            """
            A data point.

            Attributes
            --------------------
                name  -- string, name
                label -- string, label
                attrs -- string, features
            """

            self.name = name
            self.label = label
            self.attrs = attrs


        #============================================================
        # utilities
        #============================================================

        def distance(self, other) :
            """
            Return Euclidean distance of this point with other point.

            Parameters
            --------------------
                other -- Point, point to which we are measuring distance

            Returns
            --------------------
                dist  -- float, Euclidean distance
            """
            # Euclidean distance metric
            return np.linalg.norm(self.attrs-other.attrs)


        def __str__(self) :
            """
            Return string representation.
            """
            return "%s : (%s, %s)" % (self.name, str(self.attrs), self.label)


    class Cluster(object) :

        def __init__(self, points) :
            """
            A cluster (set of points).

            Attributes
            --------------------
```

```
            points -- list of Points, cluster elements
        """
        self.points = points


    def __str__(self) :
        """
        Return string representation.
        """
        s = ""
        for point in self.points :
            s += str(point)
        return s

    #============================================================
    # utilities
    #============================================================

    def purity(self) :
        """
        Compute cluster purity.

        Returns
        --------------------
            n           -- int, number of points in this cluster
            num_correct -- int, number of points in this cluster
                                with label equal to most common label in cluster
        """
        labels = []
        for p in self.points :
            labels.append(p.label)

        cluster_label, count = stats.mode(labels)
        return len(labels), np.float64(count)


    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)
        l = []
        attr = np.array([p.attrs for p in self.points])
        numPoints = len(self.points)
```

```
        pos = np.sum(attr, axis = 0) / numPoints

        for p in self.points:
          l.append(p.label)

        clusterLabel, num = stats.mode(l)
        centroid = Point('C', clusterLabel[0], pos)
        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
        dist = {}

        for p in self.points:
          dist[p] = []
          for o in self.points:
            if o != p:
              dist[p].append(p.distance(o))

        for k, v in dist.items():
          avg = np.mean(v)
          dist[k] = avg

        medoid = min(dist, key = dist.get)
        return medoid
        ### ========== TODO : END ========== ###


    def equivalent(self, other) :
        """
        Determine whether this cluster is equivalent to other cluster.
        Two clusters are equivalent if they contain the same set of points
        (not the same actual Point objects but the same geometric locations).

        Parameters
        --------------------
            other -- Cluster, cluster to which we are comparing this cluster

        Returns
```

```
            --------------------
                flag  -- bool, True if both clusters are equivalent or False otherwise
            """

            if len(self.points) != len(other.points) :
                return False

            matched = []
            for point1 in self.points :
                for point2 in other.points :
                    if point1.distance(point2) == 0 and point2 not in matched :
                        matched.append(point2)
            return len(matched) == len(self.points)


    class ClusterSet(object):

        def __init__(self) :
            """
            A cluster set (set of clusters).

            Parameters
            --------------------
                members -- list of Clusters, clusters that make up this set
            """
            self.members = []


        #============================================================
        # utilities
        #============================================================

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

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

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

            return centroids
            ### ========== TODO : END ========== ###
```

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

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

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

        return medoids
        ### ========== TODO : END ========== ###


    def score(self) :
        """
        Compute average purity across clusters in this cluster set.

        Returns
        --------------------
            score -- float, average purity
        """

        total_correct = 0
        total = 0
        for c in self.members :
            n, n_correct = c.purity()
            total += n
            total_correct += n_correct
        return total_correct / float(total)


    def equivalent(self, other) :
        """
        Determine whether this cluster set is equivalent to other cluster set.
        Two cluster sets are equivalent if they contain the same set of clusters
        (as computed by Cluster.equivalent(...)).

        Parameters
        --------------------
            other -- ClusterSet, cluster set to which we are comparing this cluster set

        Returns
        --------------------
            flag  -- bool, True if both cluster sets are equivalent or False otherwise
        """
```

```
        if len(self.members) != len(other.members):
            return False

        matched = []
        for cluster1 in self.members :
            for cluster2 in other.members :
                if cluster1.equivalent(cluster2) and cluster2 not in matched:
                    matched.append(cluster2)
        return len(matched) == len(self.members)


    #============================================================
    # manipulation
    #============================================================

    def add(self, cluster):
        """
        Add cluster to this cluster set (only if it does not already exist).

        If the cluster is already in this cluster set, raise a ValueError.

        Parameters
        --------------------
            cluster -- Cluster, cluster to add
        """

        if cluster in self.members :
            raise ValueError

        self.members.append(cluster)
```

k-means and k-medoids algorithms

```
##########################################################################
# k-means and k-medoids
##########################################################################

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

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

    Returns
```

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


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

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

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

        Returns
        --------------------
            initial_points -- list of k Points, initial cluster centers
        """
        ### ========== TODO : START ========== ###
        # part 2f: implement
        labels = set()
        groups = {}
        clusters = []

        for p in points:
          labels.add(p.label)
        for l in labels:
          groups[l] = []
        for p in points:
          groups[p.label].append(p)
        for k, v in groups.items():
            cluster = Cluster(v)
            clusters.append(cluster)

        initial_points = [cluster.medoid() for cluster in clusters]
        return initial_points
        ### ========== TODO : END ========== ###

    def kAverages(points, k, average, init = 'random', plot = True):
        kClusters = ClusterSet()
        if init == 'random':
            MU = random_init(points, k)
        elif init == 'cheat':
            MU = cheat_init(points)
```

```
        x = 1
        while True:
          clusters = {}
          for m in MU:
            clusters[m] = Cluster([])

          for p in points:
            dist = [p.distance(m) for m in MU]
            index = np.argmin(dist)
            mu2 = MU[index]
            clusters[mu2].points.append(p)

          newCluster = ClusterSet()
          for m, c in clusters.items():
            newCluster.add(c)

          if (kClusters.equivalent(newCluster)) and len(kClusters.members) > 0:
            if average != ClusterSet.centroids:
              name = "K-Medoids"
            else:
              name = "K-Means"
            break

          MU = average(newCluster)
          kClusters = newCluster

          if plot:
            if average != ClusterSet.centroids:
              plot_clusters(newCluster, f"K-Medoids [K={k}, init={init}] at i={x}", average)
            else:
              plot_clusters(newCluster, f"K-Means [K={k}, init={init}] at i={x}", average)

          x += 1
        return kClusters

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

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

```
            plot       bool, true to plot clusters with corresponding averages
                       for each iteration of algorithm

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

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

        return kAverages(points, k, ClusterSet.centroids, init = init, plot = plot)
        ### ========== TODO : END ========== ###


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

Utilities

```
    ################################################################################
    # helper functions
    ################################################################################

    def build_face_image_points(X, y) :
        """
        Translate images to (labeled) points.

        Parameters
        --------------------
            X     -- numpy array of shape (n,d), features (each row is one image)
            y     -- numpy array of shape (n,), targets

        Returns
        --------------------
```

```
            point -- list of Points, dataset (one point for each image)
        """

        n,d = X.shape

        images = collections.defaultdict(list) # key = class, val = list of images with this clas
        for i in range(n) :
            images[y[i]].append(X[i,:])

        points = []
        for face in images :
            count = 0
            for im in images[face] :
                points.append(Point(str(face) + '_' + str(count), face, im))
                count += 1

        return points


    def plot_clusters(clusters, title, average) :
        """
        Plot clusters along with average points of each cluster.

        Parameters
        --------------------
            clusters -- ClusterSet, clusters to plot
            title    -- string, plot title
            average  -- method of ClusterSet
                        determines how to calculate average of points in cluster
                        allowable: ClusterSet.centroids, ClusterSet.medoids
        """

        plt.figure()
        np.random.seed(20)
        label = 0
        colors = {}
        centroids = average(clusters)
        for c in centroids :
            coord = c.attrs
            plt.plot(coord[0],coord[1], 'ok', markersize=12)
        for cluster in clusters.members :
            label += 1
            colors[label] = np.random.rand(3,)
            for point in cluster.points :
                coord = point.attrs
                plt.plot(coord[0], coord[1], 'o', color=colors[label])
        plt.title(title)
        plt.show()


    def generate_points_2d(N, seed=1234) :
```

```
    """
    Generate toy dataset of 3 clusters each with N points.

    Parameters
    ---------------------
        N      -- int, number of points to generate per cluster
        seed   -- random seed

    Returns
    --------------------
        points -- list of Points, dataset
    """
    np.random.seed(seed)

    mu = [[0,0.5], [1,1], [2,0.5]]
    sigma = [[0.1,0.1], [0.25,0.25], [0.15,0.15]]

    label = 0
    points = []
    for m,s in zip(mu, sigma) :
        label += 1
        for i in range(N) :
            x = random_sample_2d(m, s)
            points.append(Point(str(label)+'_'+str(i), label, x))

    return points
```

Main function

```
##########################################################################
# main
##########################################################################

def main() :
    ### ========== TODO : START ========== ###
    # part 1: explore LFW data set
    # part 1a
    X, y = get_lfw_data()
    num = 12
    show_image(X[0])
    show_image(X[1])
    show_image(X[2])
    xAverage = np.mean(X, axis = 0)
    show_image(xAverage)

    # part 1b
    U, mu = PCA(X)
    plot_gallery([vec_to_image(U[:,i]) for i in range(num)])
```

```
# part 1c
num_components = [1, 10, 50, 100, 500, 1288]
print("Originals")
plot_gallery(X[:num], title = f'First 12 images (original-dimension)')
for l in num_components:
  print(f'l={l}')
  Z, Ul = apply_PCA_from_Eig(X, U, l, mu)
  new = reconstruct_from_PCA(Z, Ul, mu)
  plot_gallery(new[:num], title=f'First 12 images ({l}-dimension)')
### ========== TODO : END ========== ###



### ========== TODO : START ========== ###
# part 2d-2f: cluster toy dataset
np.random.seed(1234)
N = 20
toy_dataset = generate_points_2d(N)
kMeans(toy_dataset, 3, init = 'random', plot = True)
kMedoids(toy_dataset, 3, init = 'random', plot = True)
kMeans(toy_dataset, 3, init = 'cheat', plot = True)
kMedoids(toy_dataset, 3, init = 'cheat', plot = True)
### ========== TODO : END ========== ###



### ========== TODO : START ========== ###
# part 3a: cluster faces
kMeanScores = []
kMedoidScores = []
np.random.seed(1234)
X1, y1 = util.limit_pics(X, y, [4, 6, 13, 16], 40)
points = build_face_image_points(X1, y1)

for i in range(10):
  mean = kMeans(points, 4, init = 'random')
  kMeanScores.append(mean.score())
  medoid = kMedoids(points, 4, init = 'random')
  kMedoidScores.append(medoid.score())

print('\t average min\t max')
print(f'kMeans\t {round(sum(kMeanScores) / len(kMeanScores), 4)}\t {round(min(kMeanScores
print(f'kMedoids {round(sum(kMedoidScores) / len(kMedoidScores), 4)}\t {round(min(kMedoid



# part 3b: explore effect of lower-dimensional representations on clustering performance
kMeanScores = []
kMedoidScores = []
np.random.seed(1234)
U, mu = PCA(X)
K = 2
```

```
X1, y1 = util.limit_pics(X, y, [4, 13], 40)
components = np.arange(1, 42, K)

for i in components:
  Z, Ul = apply_PCA_from_Eig(X1, U, i, mu)
  X_rec = reconstruct_from_PCA(Z, Ul, mu)
  points = build_face_image_points(X_rec, y1)
  meanResult = kMeans(points, K, init = 'cheat')
  medoidResult = kMedoids(points, K, init = 'cheat')
  kMeanScores.append(meanResult.score())
  kMedoidScores.append(medoidResult.score())

plt.figure()
plt.plot(components, kMedoidScores, label = 'K-Medoids')
plt.plot(components, kMeanScores, label = 'K-Means')
plt.ylabel("Clustering Score")
plt.xlabel("# of Components")
plt.title("Clustering Score vs. # of Components")
plt.legend()
plt.show()


# part 3c: determine ``most discriminative'' and ``least discriminative'' pairs of images
np.random.seed(1234)
classes = set(y)
currMax = 0
maxPair = [0, 0]
currMin = 999999999999999
minPair = [0, 0]

for i in range(len(classes)):
  for j in range(len(classes)):
    if i != j:
      X1, y1 = util.limit_pics(X, y, [i, j], 40)
      points = build_face_image_points(X1, y1)
      cluster = kMedoids(points, 2, init = 'cheat')
      s = cluster.score()
      if s < currMin:
        minPair = [i,j]
        currMin = s
      elif s > currMax:
        maxPair = [i,j]
        currMax = s

print(minPair)
print(maxPair)
plot_representative_images(X, y, minPair)
plot_representative_images(X, y, maxPair)

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

```
if __name__ == "__main__" :
    main()
```