Problem Set 4: Boosting, Unsupervised Learning

Bonnie Liu
UID: 005300989
Discussion 1A (Ulzee An)

Due Date: March 12, 2022

For this problem set, I collaborated with Henry Li, Hannah Zhong, David Xiong, Justin He, Yunqiu (Rachel) Han, and Nicholas Dean.

1 AdaBoost

1.1 derivative with respect to β_t

Our objective function is

$$J(h_t(\boldsymbol{x_n}), \beta_t) = (e^{\beta_t} - e^{-\beta_t}) \sum_n w_t(n)\mathbb{I}[y_n \neq h_t(\boldsymbol{x_n})] + e^{-\beta_t} \sum_n w_t(n)$$

Recall that $\sum_n w_t(n) = 1$. Additionally, recall that in class, we defined the weighted classification error $\epsilon_t = \sum_n w_t(n)\mathbb{I}[y_n \neq h_t(\boldsymbol{x_n})]$. Then we can rewrite our objective function as

$$J(h_t(\boldsymbol{x_n}), \beta_t) = (e^{\beta_t} - e^{-\beta_t})\epsilon_t + e^{-\beta_t}$$

Taking the derivative, we get

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

If we set this expression to 0 and solve for β_t, we get the following:

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

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

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

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

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

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

1.2 β_1 of linearly separable training set

When t = 1, $\beta_1 = \frac{1}{2}ln(\frac{1-\epsilon_1}{\epsilon_1})$. Since the dataset is linearly separable, the hard-margin linear support vector machine (no slack) will classify all points correctly. This means that our weighted classification error ϵ_1 equals 0, which means $\beta_1 = \infty$. This makes sense because if our classifier $h_1(\boldsymbol{x_n})$ classifies all the data points correctly, that means we've already found a perfect classifier, and we don't need any other classifiers. Thus, we can set the weight of classifier $h_1(\boldsymbol{x_n})$ to infinity.

2 K-means for Single Dimensional Data

2.1 optimal clustering

The optimal clustering for this data is $(x_1, x_2), (x_3), (x_4)$ with prototypes 1.5, 5, and 7 respectively. The corresponding value of the objective function is

$$\begin{aligned} J(\{r_{nk}\}, \{\boldsymbol{\mu_k}\}) &= \sum_n \sum_k r_{nk} ||\boldsymbol{x_n} - \boldsymbol{\mu_k}||_2^2 \\ &= ||1-1.5||_2^2 + ||2-1.5||_2^2 + ||5-5||_2^2 + ||7-7||_2^2 \\ &= 0.5 \end{aligned}$$

2.2 Lloyd's algorithm

Our goal is to find an initialization of the cluster prototypes such that Lloyd's algorithm does not converge to the optimal solution found in part a. Suppose our cluster prototypes are initialized like so: $\boldsymbol{\mu_1} = 1$, $\boldsymbol{\mu_2} = 2$, and $\boldsymbol{\mu_3} = 6$. Then x_1 is assigned to cluster 1; x_2 is assigned to cluster 2; and x_3 and x_4 are assigned to cluster 3. Using Lloyd's algorithm, we update the prototypes using the following formula:

$$\boldsymbol{\mu_k} = \frac{\sum_n r_{nk}\boldsymbol{x_n}}{\sum_n r_{nk}}$$

We obtain the following new cluster prototypes:

$$\begin{aligned} \boldsymbol{\mu_1} &= 1 \\ \boldsymbol{\mu_2} &= 2 \\ \boldsymbol{\mu_3} &= \frac{5+7}{2} = 6 \end{aligned}$$

As we can see, our new cluster prototypes did not change from what they were initialized to. No matter how many iterations of Lloyd's algorithm we go through, our cluster prototypes will not change for this problem because the data points will not change and the algorithm has converged upon a local minimum.

Furthermore, we can show that this assignment is suboptimal by analyzing the value of the objective function when we use these cluster prototypes:

$$\begin{aligned} J(\{r_{nk}\}, \{\boldsymbol{\mu_k}\}) &= \sum_n \sum_k r_{nk} ||\boldsymbol{x_n} - \boldsymbol{\mu_k}||_2^2 \\ &= ||1-1||_2^2 + ||2-2||_2^2 + ||5-6||_2^2 + ||7-6||_2^2 \\ &= 2 \end{aligned}$$

This is greater than the value of the objective function we calculated in part a using cluster prototypes at 1.5, 5, and 7. Thus, this assignment must be suboptimal.

3 Gaussian Mixture Models

3.1 gradient of MLE

$$\begin{aligned} \nabla_{\boldsymbol{\mu_j}} \ell(\boldsymbol{\theta}) &= \frac{\partial}{\partial \boldsymbol{\mu_j}} \ell(\boldsymbol{\theta}) \\ &= \frac{\partial}{\partial \boldsymbol{\mu_j}} \sum_n ln\ p(\boldsymbol{x_n}, z_n) \\ &= \frac{\partial}{\partial \boldsymbol{\mu_j}} [\sum_k \sum_n \gamma_{nk} ln(\omega_k) + \sum_k \{\sum_n \gamma_{nk} ln(\mathcal{N}(\boldsymbol{x_n} | \boldsymbol{\mu_k}, \boldsymbol{\Sigma_k}))\}] && (1) \\ &= \frac{\partial}{\partial \boldsymbol{\mu_j}} [\sum_k \{\sum_n \gamma_{nk} ln(\frac{1}{(2\pi)^{\frac{n}{2}} |\boldsymbol{\Sigma}|^{\frac{1}{2}}} e^{-\frac{1}{2}(\boldsymbol{x_n} - \boldsymbol{\mu_k})' \boldsymbol{\Sigma}^{-1} (\boldsymbol{x_n} - \boldsymbol{\mu_k})})\}] && (2) \\ &= \frac{\partial}{\partial \boldsymbol{\mu_j}} [\sum_k \{\sum_n \gamma_{nk} (ln(\frac{1}{(2\pi)^{\frac{n}{2}} |\boldsymbol{\Sigma}|^{\frac{1}{2}}}) - \frac{1}{2} (\boldsymbol{x_n} - \boldsymbol{\mu_k})' \boldsymbol{\Sigma}^{-1} (\boldsymbol{x_n} - \boldsymbol{\mu_k}))\}] && (3) \\ &= \frac{\partial}{\partial \boldsymbol{\mu_j}} [\sum_k \{-\frac{1}{2} \sum_n \gamma_{nk} (\boldsymbol{x_n} - \boldsymbol{\mu_k})' \boldsymbol{\Sigma}^{-1} (\boldsymbol{x_n} - \boldsymbol{\mu_k})\}] && (4) \\ &= -\frac{1}{2} \sum_n \gamma_{nj} 2 \boldsymbol{\Sigma}^{-1} (\boldsymbol{x_n} - \boldsymbol{\mu_k})(-1) && (5) \\ &= \sum_n \gamma_{nj} \boldsymbol{\Sigma}^{-1} (\boldsymbol{x_n} - \boldsymbol{\mu_j}) && (6) \end{aligned}$$

From step 1 to step 2, we removed the first term because it does not contain $\boldsymbol{\mu_j}$ and thus will become zero when we take derivative with respect to $\boldsymbol{\mu_j}$. We also plugged in the probability density function of a normal distribution of $\boldsymbol{x_n}$ given parameters $\boldsymbol{\mu_k}$ and $\boldsymbol{\Sigma_k}$.

From step 2 to step 3, I used the product property of logarithms to split up the expression into two terms. In step 4, the first term obtained from the split goes away since it does not contain $\boldsymbol{\mu_j}$ and will thus zero out when we take the derivative with respect to $\boldsymbol{\mu_j}$.

In step 5, I used Ulzee's hint from Campuswire post 398. We know that if $\boldsymbol{y} = f(\boldsymbol{x}) = \boldsymbol{x}'\boldsymbol{A}\boldsymbol{x}$, then $\frac{\partial f}{\partial \boldsymbol{x}} = 2\boldsymbol{A}\boldsymbol{x}$. After cancelling out some constant factors in step 6, we get the final answer.

3.2 solving for $\boldsymbol{\mu_j}$

$$\textstyle\sum_n \gamma_{nj} \boldsymbol{\Sigma}^{-1}(\boldsymbol{x_n} - \boldsymbol{\mu_j}) = 0$$

$$\sum_n \gamma_{nj} \boldsymbol{\Sigma}^{-1}\boldsymbol{x_n} - \sum_n \gamma_{nj} \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu_j} = 0$$

$$\boldsymbol{\Sigma}^{-1} \sum_n \gamma_{nj}\boldsymbol{x_n} = \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu_j} \sum_n \gamma_{nj}$$

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

3.3 EM algorithm

We know from lecture that we can calculate the new weights and means using the following equations:

$$\omega_k = \frac{\sum_n \gamma_{nk}}{\sum_k \sum_n \gamma_{nk}}$$

$$\mu_k = \frac{\sum_n \gamma_{nk} x_n}{\sum_n \gamma_{nk}}$$

Let's calculate the weights first.

$$\begin{aligned}
\omega_1 &= \frac{\sum_n \gamma_{n1}}{\sum_k \sum_n \gamma_{nk}} \\
&= \frac{0.2 + 0.2 + 0.8 + 0.9 + 0.9}{5} \\
&= \frac{3}{5} \\
&= 0.6 \\
\omega_2 &= \frac{\sum_n \gamma_{n2}}{\sum_k \sum_n \gamma_{nk}} \\
&= \frac{0.8 + 0.8 + 0.2 + 0.1 + 0.1}{5} \\
&= \frac{2}{5} \\
&= 0.4
\end{aligned}$$

To check, $\omega_1 = 0.6 > 0$ and $\omega_2 = 0.4 > 0$ and $\omega_1 + \omega_2 = 0.6 + 0.4 = 1$, which satisfies our constraints.

Now let's calculate the means.

$$\begin{aligned}
\mu_1 &= \frac{\sum_n \gamma_{n1} x_n}{\sum_n \gamma_{n1}} \\
&= \frac{0.2(5) + 0.2(15) + 0.8(25) + 0.9(30) + 0.9(40)}{0.2 + 0.2 + 0.8 + 0.9 + 0.9} \\
&= 29 \\
\mu_2 &= \frac{\sum_n \gamma_{n2} x_n}{\sum_n \gamma_{n2}} \\
&= \frac{0.8(5) + 0.8(15) + 0.2(25) + 0.1(30) + 0.1(40)}{0.8 + 0.8 + 0.2 + 0.1 + 0.1} \\
&= 14
\end{aligned}$$

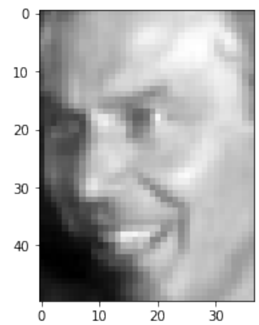
4 Implementation: Clustering and PCA

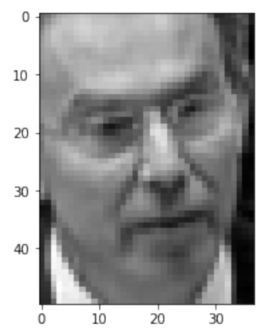
4.1 PCA and Image Reconstruction: get_lfw_data() and average face

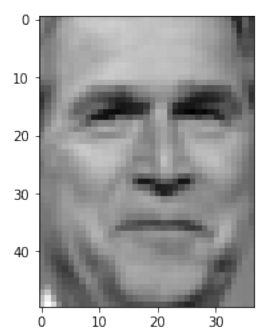
```
# part 1: explore LFW data set
X, y = util.get_lfw_data()

part 1a: plot some input images, compute mean, and plot average face
util.show_image(X[0:])
util.show_image(X[1:])
util.show_image(X[2:])

avg_face = X.mean(axis=0)
util.show_image(avg_face)
```

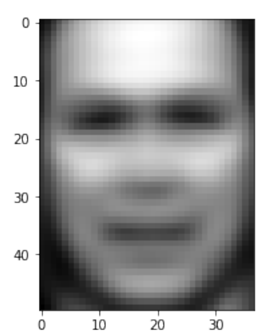
The above code outputted the following results. We see that although it is the compilation of many different images, the average face still has the features of a human face. We can somewhat distinguish the eyes, nose, and mouth.

```
Total dataset size:
        num_samples: 1867
        num_features: 1850
        num_classes: 19
```


0
10
20
30
40
0
10
20
30


0
10
20
30
40
0
10
20
30


0
10
20
30
40
0
10
20
30


0
10
20
30
40
0
10
20
30

4.2 PCA and Image Reconstruction: top 12 eigenfaces

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

These are selected as the top 12 eigenfaces because they capture the most variance out of our data. Notice how each of these faces look pretty different from one another.

4.3 PCA and Image Reconstruction: effect of using more or fewer dimensions to represent images

```
# part 1c: effect of using more or fewer dimensions to represent images
num_components = [1, 10, 50, 100, 500, 1288]
for l in num_components:
```

```
Z, Ul = util.apply_PCA_from_Eig(X, U, l, mu)
X_rec = util.reconstruct_from_PCA(Z, Ul, mu)
print("gallery for l = " + str(l))
util.plot_gallery(X_rec)
```

For l = 1,

For l = 10,

For l = 50,

For l = 100,

For l = 500,

For l = 1288,

As we decrease the number of components l, we are able to capture less variance in the data since we reduce the dimensions by so much. When l = 1, we can barely distinguish the faces from one another.

4.4 K-Means and K-Medoids: minimizing objective function over $\boldsymbol{\mu}$, $\boldsymbol{c}$, and k

The minimum possible value of $J(\boldsymbol{\mu}, \boldsymbol{c}, k)$ is 0. This occurs if $k = n$, $\boldsymbol{\mu}_c^{(i)} = \boldsymbol{x}^{(i)}$ for all $i \in \{1, ..., n\}$, and $c^{(i)} = i$ for all $i \in \{1, ..., n\}$.

4.5 K-Means and K-Medoids: Cluster and ClusterSet implementation

```
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)
        attrs = 0
        label_to_freq = {}
        for point in self.points:
          attrs += point.attrs
          if point.label in label_to_freq:
            label_to_freq[point.label] += 1
          else:
            label_to_freq[point.label] = 0
        attrs /= len(self.points)
        label_based_on_asc_freq = sorted(label_to_freq, key=label_to_freq.get)
        centroid = Point('centroid', label_based_on_asc_freq[-1], attrs)
        return centroid
        ### ========== TODO : END ========== ###

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

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

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

```python
        """

        ### ========== TODO : START ========== ###
        # part 2b: implement
        medoid = None
        minDist = sys.float_info.max
        for i in self.points:
          dist = 0
          for j in self.points:
            dist += i.distance(j)
          if dist < minDist:
            minDist = dist
            medoid = i
        return medoid
        ### ========== 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 = [cluster.medoid() for cluster in self.members]
        return medoids
        ### ========== TODO : END ========== ###
```

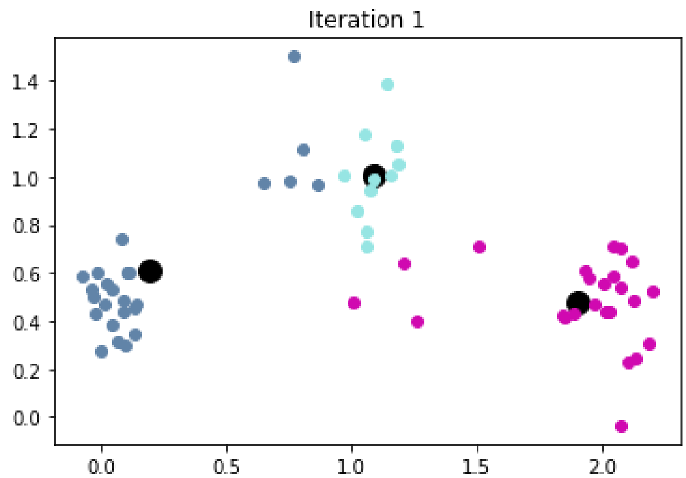
4.6 K-Means and K-Medoids: random_init() and kMeans()

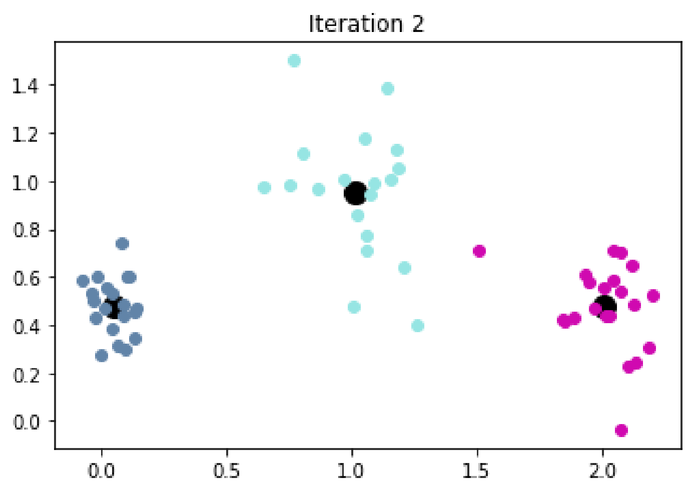
```python
def random_init(points, k) :
    """
    Randomly select k unique elements from points to be initial cluster centers.

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

    Returns
    --------------------
        initial_points -- list of k Points, initial cluster centers
    """
```

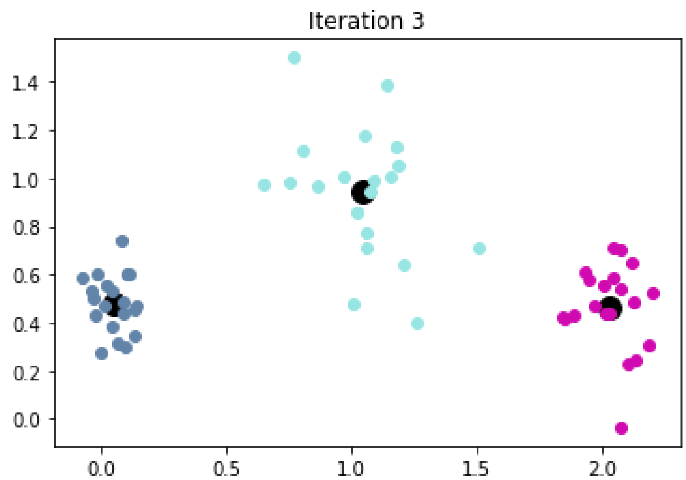
```
    ### ========== TODO : START ========== ###
    # part 2c: implement (hint: use np.random.choice)
    initial_points = np.random.choice(points, size=k, replace=False)
    return initial_points
    ### ========== TODO : END ========== ###

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(...).


    prev_k_clusters = None
    centers = random_init(points, k) if init == 'random' else cheat_init(points)
    k_clusters = None
    iter = 1
    while prev_k_clusters is None or not k_clusters.equivalent(prev_k_clusters):
      print("iter = " + str(iter))
      prev_k_clusters = k_clusters
```

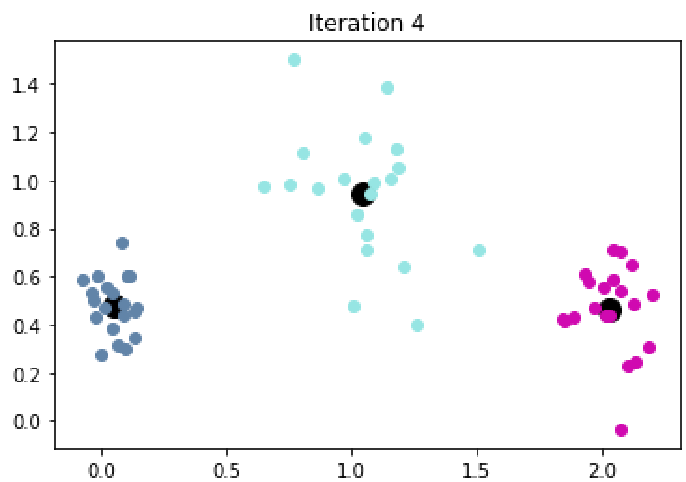
```
      # assign each point to a cluster
      k_clusters = assign(points, centers)
      # plot clusters for current iteration
      if plot:
        plot_clusters(k_clusters, 'Iteration ' + str(iter), ClusterSet.centroids)
      # update prototype of clusters
      centers = k_clusters.centroids()
      iter += 1
    return k_clusters
    ### ========== TODO : END ========== ###

def assign(points, centers) :
    """
    Assigns each point to a cluster center.

    Parameters
    --------------------
        points   -- list of Points, dataset
        centers  -- list of k cemters

    Returns
    --------------------
        k_clusters  -- ClusterSet, k clusters
    """
    # maps center index to list of points
    idx_to_pts = {idx : [] for idx in range(len(centers))}
    # find closest center for each point
    for point in points:
        closestCenterIdx = -1
        minDist = sys.float_info.max
        for i in range(len(centers)):
          if point.distance(centers[i]) <= minDist:
            minDist = point.distance(centers[i])
            closestCenterIdx = i
        idx_to_pts[closestCenterIdx].append(point)
    k_clusters = ClusterSet()
    for idx in idx_to_pts:
      cluster = Cluster(idx_to_pts[idx])
      k_clusters.add(cluster)
    return k_clusters
```

4.7 K-Means and K-Medoids: Test Performance





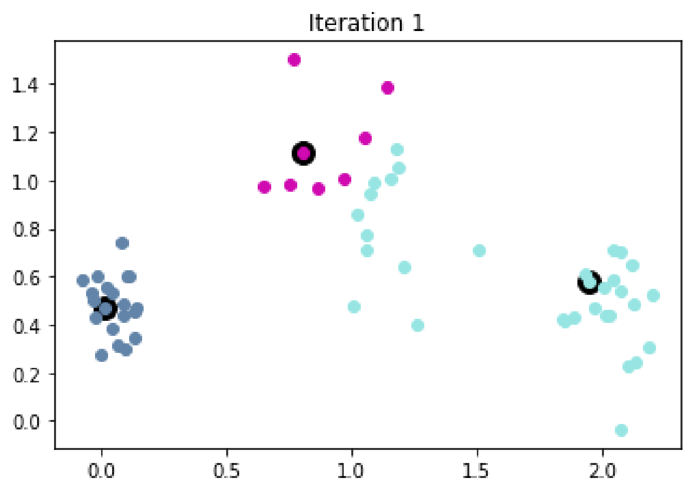


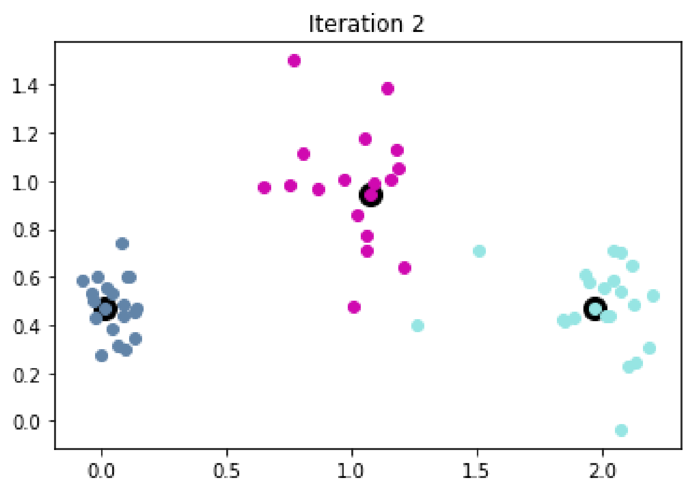


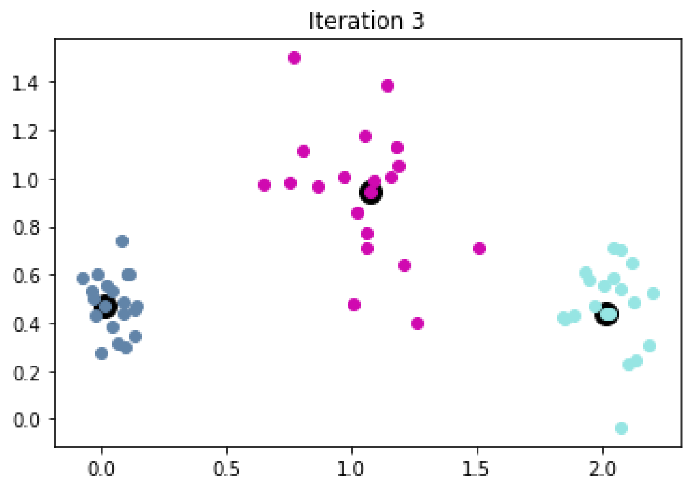
4.8 K-Means and K-Medoids: K-Medoids

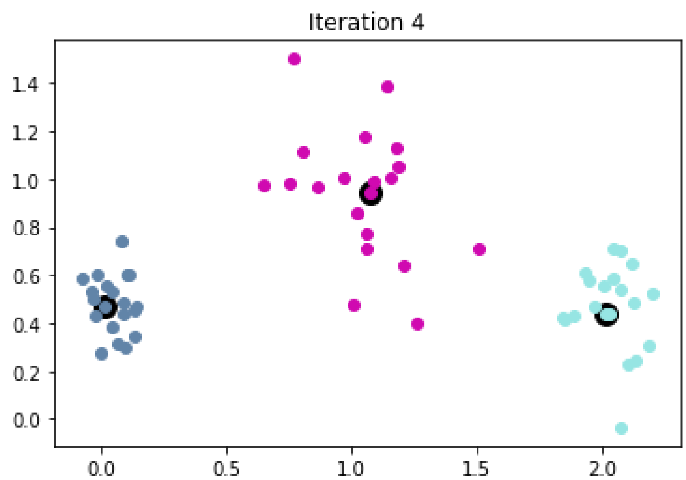
```
def kMedoids(points, k, init='random', plot=False) :
    """
    Cluster points in k clusters using k-medoids clustering.
    See kMeans(...).
    """
    ### ========== TODO : START ========== ###
    # part 2e: implement
    prev_k_clusters = None
    centers = random_init(points, k) if init == 'random' else cheat_init(points)
    k_clusters = None
    iter = 1
```

```
while prev_k_clusters is None or not k_clusters.equivalent(prev_k_clusters):
  print("iter = " + str(iter))
  prev_k_clusters = k_clusters
  # assign each point to a cluster
  k_clusters = assign(points, centers)
  # plot clusters for current iteration
  if plot:
    plot_clusters(k_clusters, 'Iteration ' + str(iter), ClusterSet.medoids)
  # update prototype of clusters
  centers = k_clusters.medoids()
  iter += 1
return k_clusters
```









4.9 K-Means and K-Medoids: cheat_init()

5 Code

```
#####################################################################
# 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)
        attrs = 0
        label_to_freq = {}
        for point in self.points:
          attrs += point.attrs
          if point.label in label_to_freq:
            label_to_freq[point.label] += 1
          else:
            label_to_freq[point.label] = 0
        attrs /= len(self.points)
        label_based_on_asc_freq = sorted(label_to_freq, key=label_to_freq.get)
        centroid = Point('centroid', label_based_on_asc_freq[-1], attrs)
        return centroid
        ### ========== TODO : END ========== ###


    def medoid(self) :
        """
        Compute medoid of this cluster, that is, the point in this cluster
        that is closest to all other points in this cluster.

        Returns
        --------------------
            medoid -- Point, medoid of this cluster
        """

        ### ========== TODO : START ========== ###
        # part 2b: implement
        medoid = None
        minDist = sys.float_info.max
        for i in self.points:
          dist = 0
          for j in self.points:
            dist += i.distance(j)
          if dist < minDist:
            minDist = dist
            medoid = i
        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 = [cluster.centroid() for cluster in self.members]
        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 = [cluster.medoid() for cluster in self.members]
        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
######################################################################
```

```python
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)
    initial_points = np.random.choice(points, size=k, replace=False)
    return initial_points
    ### ========== 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
    initial_points = []
    return initial_points
    ### ========== 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(...).


prev_k_clusters = None
centers = random_init(points, k) if init == 'random' else cheat_init(points)
k_clusters = None
iter = 1
while prev_k_clusters is None or not k_clusters.equivalent(prev_k_clusters):
  print("iter = " + str(iter))
  prev_k_clusters = k_clusters
  # assign each point to a cluster
  k_clusters = assign(points, centers)
  # plot clusters for current iteration
  if plot:
    plot_clusters(k_clusters, 'Iteration ' + str(iter), ClusterSet.centroids)
  # update prototype of clusters
  centers = k_clusters.centroids()
```

```
      iter += 1
    return k_clusters
    ### ========== 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
    prev_k_clusters = None
    centers = random_init(points, k) if init == 'random' else cheat_init(points)
    k_clusters = None
    iter = 1
    while prev_k_clusters is None or not k_clusters.equivalent(prev_k_clusters):
      print("iter = " + str(iter))
      prev_k_clusters = k_clusters
      # assign each point to a cluster
      k_clusters = assign(points, centers)
      # plot clusters for current iteration
      if plot:
        plot_clusters(k_clusters, 'Iteration ' + str(iter), ClusterSet.medoids)
      # update prototype of clusters
      centers = k_clusters.medoids()
      iter += 1
    return k_clusters






    k_clusters = ClusterSet()
    return k_clusters
    ### ========== TODO : END ========== ###

def assign(points, centers) :
    """
    Assigns each point to a cluster center.

    Parameters
    --------------------
        points   -- list of Points, dataset
        centers  -- list of k cemters
```

```
    Returns
    --------------------
        k_clusters  -- ClusterSet, k clusters
    """
    # maps center index to list of points
    idx_to_pts = {idx : [] for idx in range(len(centers))}
    # find closest center for each point
    for point in points:
        closestCenterIdx = -1
        minDist = sys.float_info.max
        for i in range(len(centers)):
          if point.distance(centers[i]) <= minDist:
            minDist = point.distance(centers[i])
            closestCenterIdx = i
        idx_to_pts[closestCenterIdx].append(point)
    k_clusters = ClusterSet()
    for idx in idx_to_pts:
      cluster = Cluster(idx_to_pts[idx])
      k_clusters.add(cluster)
    return k_clusters

######################################################################
# 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 cla
    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
######################################################################

def main() :
    ### ========== TODO : START ========== ###
    # part 1: explore LFW data set
    X, y = util.get_lfw_data()

    # part 1a: plot some input images, compute mean, and plot average face
    # util.show_image(X[0:])
    # util.show_image(X[1:])
    # util.show_image(X[2:])

    avg_face = X.mean(axis=0)
    # util.show_image(avg_face)

    # part 1b: perform PCA
    U, mu = util.PCA(X)
    # util.plot_gallery([util.vec_to_image(U[:,i]) for i in range(12)])

    # part 1c: effect of using more or fewer dimensions to represent images
    num_components = [1, 10, 50, 100, 500, 1288]
```

```
    for l in num_components:
      Z, Ul = util.apply_PCA_from_Eig(X, U, l, mu)
      X_rec = util.reconstruct_from_PCA(Z, Ul, mu)
      # print("gallery for l = " + str(l))
      # util.plot_gallery(X_rec)
    ### ========== TODO : END ========== ###



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



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


    # part 3b: explore effect of lower-dimensional representations on clustering performanc
    np.random.seed(1234)


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


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


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