Assignment4

---

1. Spectral Clustering

Before we do the following tasks, we need to implement the spectral clustering algorithm before. The implementing procedure for the algorithm is very simple, the algorithm can be done by these steps:

1. From distance matrix `W` form matrix `D`, which is a diag matrix and each value in diagonal is the sum of each row of `W`.
2. Then solve the general eigen problem for matrix `D - W` and `W`.
3. Select the second smallest eigen value and coresponding eigen vector `y`.
4. Perform kmeans algorithm on vector `y`.

   So the main code is below, you can also see in `spectral_clustering/spectral.m`:

```
sum_w_matrix = sum(W, 2);
D = zeros(size(W));

% form matrix D
for i=1:size(D, 1)
    D(i, i) = sum_w_matrix(i);
end

[y_matrix, value_matrix] = eig(D - W, D);
y = y_matrix(:, 2); % second smallest eigen vector
idx = litekmeans(y, k); % perform kmeans
```

- **(a)**

  In this part we will experiment spectral clustering on synthesis data. First we need to construct the knn-graph. The thoughts also very direct, we first calculate the distance by `EuDist2.m` in matrix `distance_matrix`. Then for every data point we select the k closest points out, this we can sort and find the first k min distances for every point and corresponding index.

  Also we need to filter the very far points since we only need to focus the local connectivity, so we also need to consider the threshold for point's distance. So one simple way to consider these two parts is that we just let the `distance_matrix` to find points which less than the `min(vector, threshold)`.

  This problem I select one proper threshold is `0.6` and the neighbor I select for knn-graph is `300`. They work fine in the result so I choose these.

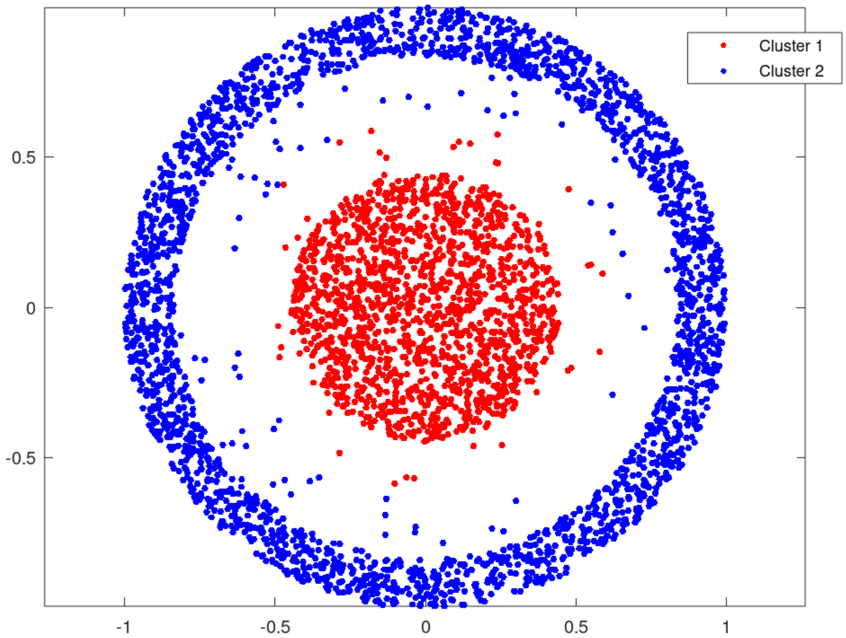
  One last thing is try to make the graph matrix `W symmetric` in order to calculate the eigen decomposition faster.

The implementing is below, you can also find in `spectral_clustering/knn_graph.m and kmin_vector.m`:

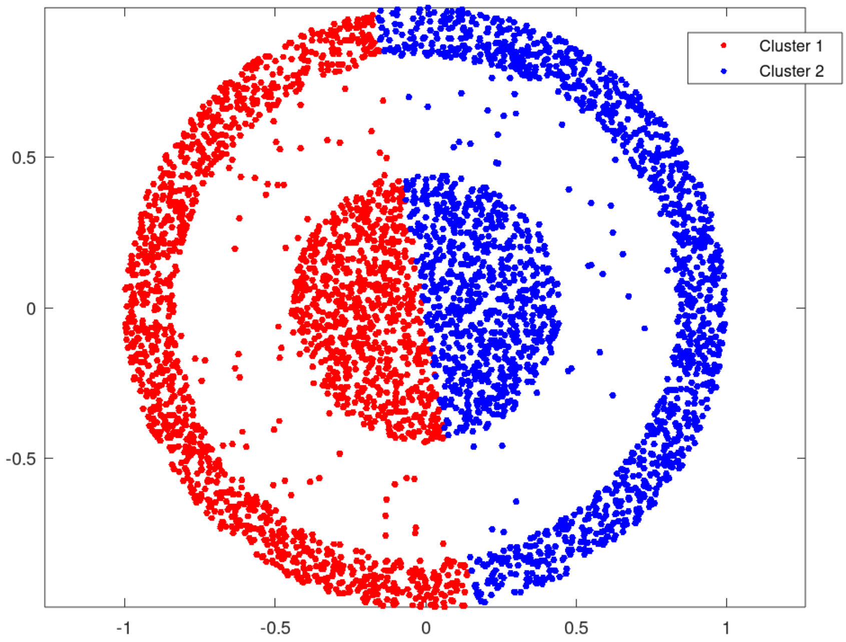
```
function [vector, idx] = kmin_vector(D, k)
    [Y, idx] = sort(D);
    vector = Y(k, :);
    idx = idx(k, :);
end

function W = knn_graph(X, k, threshold)
    distance_matrix = EuDist2(X);
    [vector, idx] = kmin_vector(distance_matrix, k);

    W = zeros(size(distance_matrix));
    % consider threshold
    W = distance_matrix <
        min([vector;threshold * ones(size(vector))], [], 1);
    W = W | W'; % make symmetric matrix
end
```

Results shown: You will clearly to find that spectral clustering performs better than kmeans in this situation!

- Spectral clustering:



- Kmeans:



- **(b)**

  This part we try spectral clustering on real-world data. The main task we need to do is to compare the performance of kmeans and spectral clustering on real world data. There are two evaluation indicators: accuracy and normalized mutual information. **The higher accuracy and normalized mutual information means that the clustering algorithm performs better.**

  So we gonna to implement all these. I follow Matlab codes for clustering ways to calculate the accuracy and normalized mutual information. The main ways is below:

```
res = bestMap(gnd,res);
%==  evaluate AC: accuracy ==
AC = length(find(gnd == res))/length(gnd);
%==  evaluate MIhat: nomalized mutual information ==
MIhat = MutualInfo(gnd,res);
```

For kmeans, I use the professor Dengcai's `litekeans.m` to do kmeans algorithm. For spectral I use the `constructW.m` and set `options.NeighborMode = 'KNN'; options.k = 7; options.WeightMode = 'Binary';` I try `20` times for the loop, and get the final results, you can find the code in `spectral_exp2.m`:

- kmeans: accuracy = **0.5059**, mutual information = **0.3852**
- spectual: accuracy = **0.8332**, mutual information = **0.5261**

```
Accuracy for spectral is 0.833207, mutual information is 0.526050
Accuracy for kmeans is 0.505876, mutual information is 0.385204
```

---

2. Principal Component Analysis

Also before all tasks, we need to implement the PCA algorithm. The implementing procedure is below:

1. Calculate the covariance matrix of data, just simply by matlab builtin function `S = cov(data)`.
2. Calculate the eigen values and eigen vectors of matrix `S`.
3. Sort the eigen vectors by the descending order of eigen values. Every eigen vector is called one principal component.

```
function [eigvector, eigvalue] = pca(data)
    % calculate cov matrix of data
    S = cov(data);
    [vectors, value] = eig(S);
    % make value vectors
    value = sum(value, 2);
    % sorting by descend
    [value, idx] = sort(value, 'descend');
    eigvector = vectors(:, idx);
    eigvalue = value;
end
```

- **(a)**

  In this part we gonna to use the pca algorithm to hack some rotated images for us. The main thoughts as for me is below:

1. Find out the non background color pixel by `[x, y] = find(img_r < 255);`
2. Then use the coordinate x, y as the feature of one data point.
3. Perform pca on data points and get the eigen vectors.
4. Then I use the first eigen vector which represents the direction of most variance information contains. Calculate the angle with x axis using builtin functions `atan2` and `rad2deg` to calculate.
5. Then simply rotate the input image with the angle calculated.
6. Filter the background points piexl value to 255(white).

```
function img = hack_pca(filename)
    img_o = imread(filename);
    img_r = double(img_o);

    % here use position in image to represent features
    % filter non background piexls
    [x, y] = find(img_r < 255);
    data = [x, y];
    [eigvectors, eigvalues] = pca(data);

    eigenvector = eigvectors(:, 1)';
    rotate_angle = rad2deg(atan2(eigenvector(1), eigenvector(2)));
    new_image_r = imrotate(img_o, rotate_angle);
    new_image_r(find(double(new_image_r) < 1)) = 255;

    figure;
    imshow(uint8(new_image_r));
end
```

Then you can run the `show_rotate.m` written by me show the final results:

- 1.gif:

- 2.gif

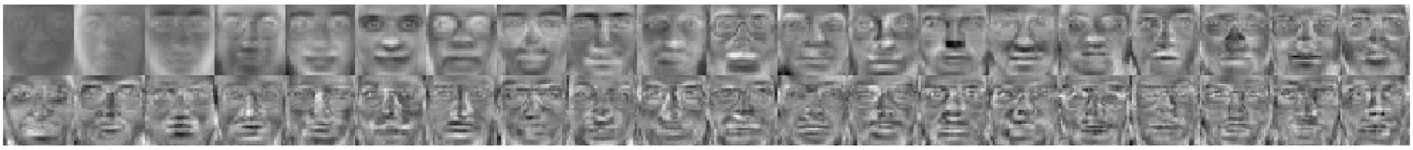
- 3.gif

- 4.gif

- 5.gif

- **(b)**
- **(i)**

  The eigen faces shown below:



- **(ii)**

  The procedures for this task is:

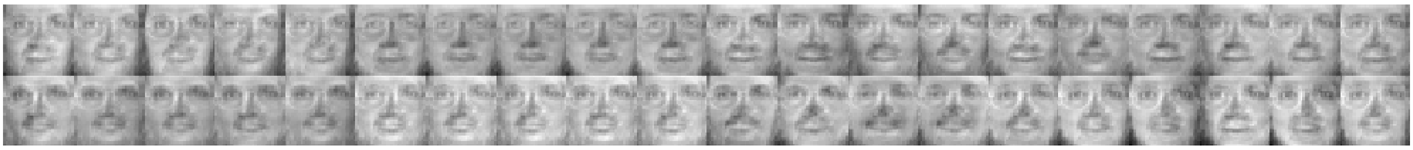
1. First perform PCA on `fea_train` to get the eigen vectors and eigen values.
2. According to the choosen dimension `k` to select the first `k` eigen vectors to form transformation matrix.
3. Multiply the transformation matrix with `fea_train` and `fea_test` to get `low_dim_train` and `low_dim_test`.
4. Perform knn with `low_dim_test`, `low_dim_train`, `gnd_train` and `k = 1` to get the `predict_test`.
5. Using the `predict_test` and `gnd_test` to calculate the accuracy.
6. Then using $\check{X} = A(A^T X)$ to reconstruct the image and show_face. This part code you can see in `pca_exp1.m`.

```
load('ORL_data', 'fea_Train', 'gnd_Train', 'fea_Test', 'gnd_Test');
% 1. Feature preprocessing
[num_train, num_featrues] = size(fea_Train);
[num_test, num_featrues] = size(fea_Test);
% 2. Run PCA
[eigenvectors, eigenvalues] = pca(fea_Train);
% 3. Visualize eigenface
show_face(eigenvectors');
show_face(fea_Train);
% 4. Project data on to low dimensional space
reduced_dim = [8, 16, 32, 64, 128];
% 5. Run KNN in low dimensional space
for i=1:length(reduced_dim)
    current_dim = reduced_dim(i);
    transform_matrix = eigenvectors(:, 1:current_dim);
    low_dim_train = fea_Train * transform_matrix;
    low_dim_test = fea_Test * transform_matrix;

    predict_test = knn(low_dim_test', low_dim_train', gnd_Train', 1);
    predict_test = predict_test';
    correct_num = sum(predict_test == gnd_Test);

    fprintf("For dim %d,the testing error rate is %f\n",
        current_dim,
        1 - (correct_num / num_test));

    figure;
    % reconstruction
    show_face(low_dim_train * transform_matrix');
end
```

- origin image:

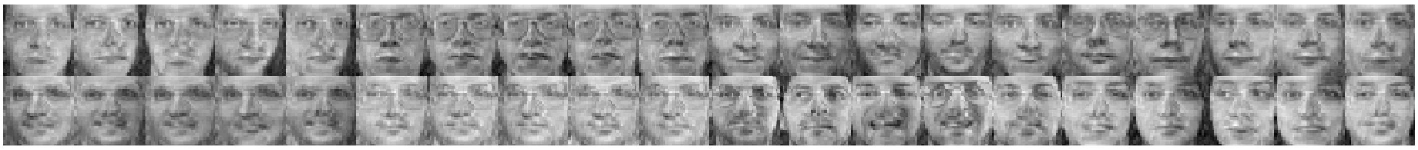
- dimension 8:

- dimension 16:



- dimension 32:

- dimension 64:



- dimension 128:

And the results for testing error is shown below:

Dimension	Testing error
8	0.245
16	0.2
32	0.18
64	0.15
128	0.15

```
octave:10> pca_exp1
For dim 8, the testing error rate is 0.245000
For dim 16, the testing error rate is 0.200000
For dim 32, the testing error rate is 0.180000
For dim 64, the testing error rate is 0.150000
For dim 128, the testing error rate is 0.150000
```

Additional, I also tried LDA in this task by professor Dengcai's code `LDA.m` to implement this algorithm. The main procedure is same as above, only change from PCA to LDA to get transformation matrix. These are the results for LDA performance, you will see the significant performance for LDA algorithm (you can see the code in `pca_exp2.m`):

Dimension	Testing error
8	0.215
16	0.115
32	0.065
39	0.055

```
octave:14> pca_exp2
For dim 8, the testing error rate is 0.215000
For dim 16, the testing error rate is 0.115000
For dim 32, the testing error rate is 0.065000
For dim 39, the testing error rate is 0.055000
```