# Report of Project-4

Tianran Zhang 15300180104

December 30, 2017

## Contents

**1 Dimensionality Reduction** **1**
1.1 freyface.mat . . . 2
1.2 PCA . . . 2
1.2.1 Find the num of k . . . 3
1.2.2 Details in Eigenvectors . . . 4
1.2.3 Top Two Eigenvectors . . . 5
1.2.4 Reconstruct Frey's Face . . . 6
1.2.5 Adding Noise . . . 6
1.3 swissroll.m . . . 7

**2 Gaussian Mixture Model** **8**
2.1 EM for Mixture of Gaussians . . . 8
2.2 Mixtures of Gaussians . . . 9
2.3 Training . . . 10
2.4 Initializing a mixture of Gaussians with k-means . . . 14
2.5 Classification using MoGs . . . 15

**3 Matlab Codes** **17**
3.1 find_eigenvector.m . . . 17
3.2 ReconstructFace . . . 18
3.3 distmat . . . 18
3.4 swissroll . . . 19
3.5 run_q3 . . . 20

# 1 Dimensionality Reduction

In this section, I experienced PCA and use dimension reduction to explore the data. Dimension reduction is a very important method in PCA, through which we could explore the data in a lower dimension, which will make the analysis convient and intuitive.

## 1.1 freyface.mat

First, we explore the dataset we got, freyface.mat, the images of Brendan Frey's face in a variety of expressions. X in freyface.mat contains 1965 images of Brendan Frey's face, and we could see the first 100 Frey's faces by :

```
>> load freyface.mat
>> X = double(X);
>> showfreyface(X(:, 1:100))
```

The images are:

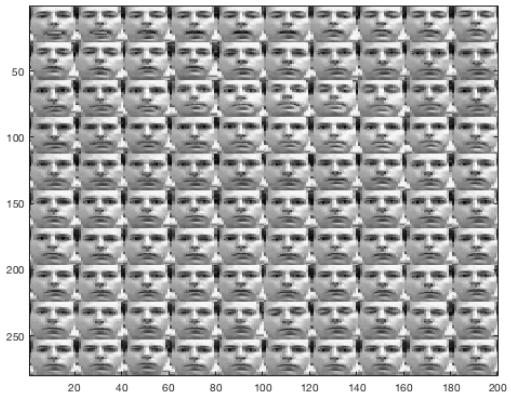


Figure 1: The first 100 images of Brendan Frey's faces

## 1.2 PCA

Find the eigenvectors of $X * X^T/N$, both with and without first removing the mean(I put the $\lambda$ in a descending order instead the original one with a increasing order):

```
load freyface.mat
X = double(X);

[m, N] = size(X);
% lambda_un is the eigenvalues without removing the mean
% with a descending order.
% Vun is the eigenvectors correspond to lambda_un.
[Vun, Dun] = eig(X*X'/N);
[lambda_un, order] = sort(diag(Dun), 'descend');
Vun = Vun(:, order);

% lambda_ctr is the eigenvalues with removing the mean
% with a descending order.
% Vctr is the eigenvectors correspond to lambda_ctr.
Xctr = X - repmat(mean(X, 2), 1, N);
```

```
[Vctr, Dctr] = eig(Xctr*Xctr' /N);
[lambda_ctr, order] = sort(diag(Dctr), 'descend');
Vctr = Vctr(:, order);
```

### 1.2.1 Find the num of k

Look at the eigenspectra (I only plot the $\lambda$ first remove the mean):

```
figure;
plot(1:m, lambda_ctr(1:m, 1));
xlabel('number of k');
ylabel('lambda\_un');
```

The plot of $\lambda$ is :

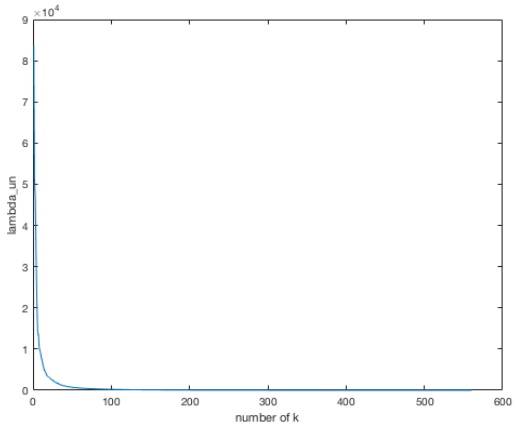


Figure 2: $\lambda$ against k ranged from 0 to 600

From the plot we could find that the method works better when k is in range 0 to 100. So I draw another plot of $\lambda$ and fix the number of k between 0 to 100.

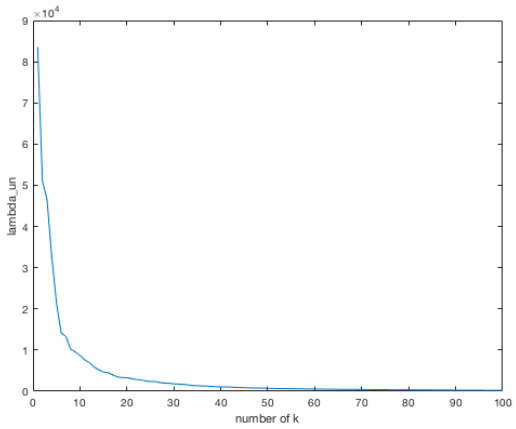


Figure 3: $\lambda$ against k ranged from 0 to 100

It's still hard to tell what nember of k we should choose. So now define a rate, the sum of 1th to kth lambda weights over the whole lambda:

$$rate = \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{N} \lambda_i}$$

And we want to find out the best k to make the rate over 95%. I wrote the script to find out the best number of k:

```
% p  : the number of k
% p1 : the percentage of the first p lambdas take  in all lambdas
% S0 : the weights of the first p lambdas
% S1 : the weights of all lambdas
p = 0;        p1 = 0;
s0 = 0;       s1 = sum(lambda_ctr);
while (p1 <0.95)
   p = p + 1;
   s0 = s0 + lambda_ctr(p);
   p1 = s0/s1;
end
```

and the result shows that we should make k = 80.
The Matlab codes in this part named*find_eigenvector.m* is shown in section 3.1 below.

### 1.2.2 Details in Eigenvectors

Look at the top 16 eigenvectors in each case:

```
showfreyface(Vctr(:, 1:16));
```

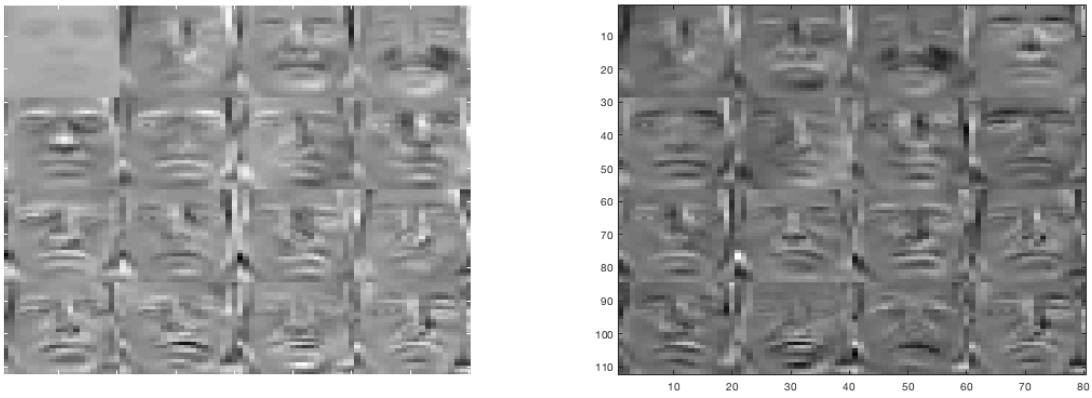


Figure 4: The top 16 eigenvectors in each case(The left one is the top 16 eigenvectors without removing means while the right one is removed)

From the plots we could find that each figure seems basicly a human face. They each present a most important features of Frey's face. They are different in different features of Frey's

face. For example, the first figure shows most clearly Frey's chin, and the second shows Frey's mouth more clearly, so on. We could say that the top 16 eigenvectors each represent a principal component of Frey's face.
Compared the one without removing the mean, the one have removed means seems more clearly and works better in details od the images. So I think that PCA requires to remove the mean.

### 1.2.3 Top Two Eigenvectors

Plot the data on the top two eigenvectors, and plot the resulting 2D points:

```
Yun = Vun(:, 1:2)' * X;
Yctr = Vctr(:, 1:2)' * X;

figure;
plot(Yctr(1, :), Yctr(2, :), '.');
figure;
plot(Yun(1, :), Yun(2, :), '.');
```

The plots are shown below:

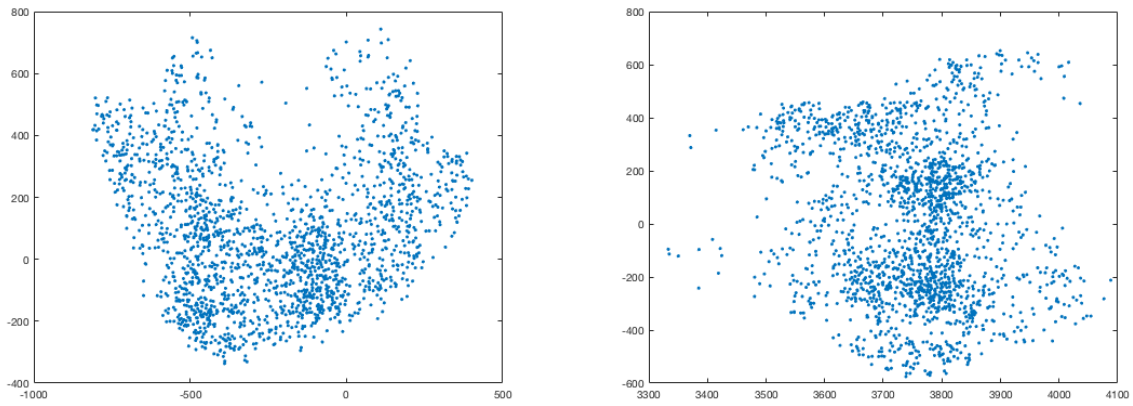


Figure 5: The resulting 2D points in a 2-D coordinate system ( The left one is the plot of Vctr and the right one is the plot of Vun)

We can see that the plot with removing the mean concentrated mostly around 0, while the plot without removing the mean are more decentralized.
Use the function *explorefreymanifold* to explore the space (We only concentrate on the plots with removing the mean):

```
explorefreymanifold(Yctr, X);
```

The function *explorefreymanifold* has an interactive mode to allow users to click on a point and see the corresponding face image.
When I click the points of a small cluster, the Frey's faces are basicly with the same expressions. Because they are in the same principal component.

### 1.2.4 Reconstruct Frey's Face

Now, try reconstructing a face from an arbitrary point in the space. We do this by the following steps:
a. Choose a point y within the space and show the Frey's face it represent.
b. Compose the corresponding projeted vector $\hat{x}$, by reconstruct the figure.
To reconstruct Frey's face,I wrote a function:

```
function [X_re] = reconstruct(Y,V)
  X_re = V * Y;
end
```

So after running the Codes:

```
y1 = randi([1,255], 560, 1);
figure;
showfreyface(y1);
figure;
showfreyface(reconstruct(V'*(y1-mean(X,2)),V)+mean(X,2));
```

We get the original and the reonstructed Frey's face :

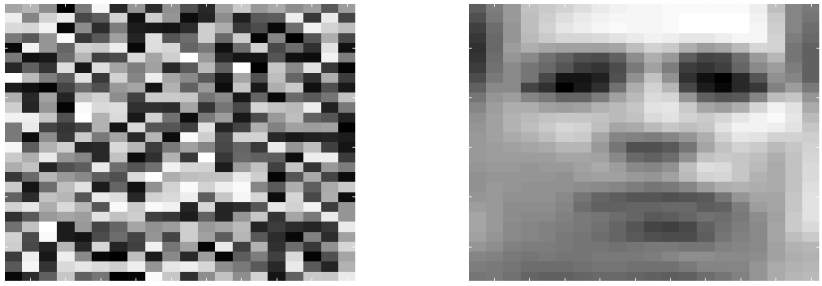

Figure 6: The original and reconstructed Frey's face

The reconstructed one looks much better than the original one.

### 1.2.5 Adding Noise

Try adding noise to a face and then reconstructing Frey's face(We choose the 100th Frey's face to construct the noise):

```
% Adding noise (choosing the 100 th Fray's face)
X_noise = X(:, 100) + 10*randn(m,1);
figure;
showfreyface(X_noise);
figure;
showfreyface(reconstruct(V'*(X_noise-mean(X,2)),V)+mean(X,2));
```

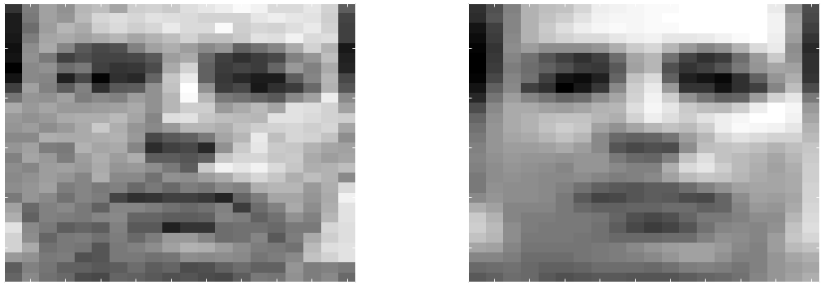

Figure 7: The original and reconstructed Frey's face after adding the noise

This time, the figure looks better in both the original one and the reconstructed one. The Matlab Code in this section is named *ReconstructFace* is shown in Sec.3.2

## 1.3 swissroll.m

In this section, I extract the dataset generation part of *swissroll.m*, and try running Isomap method on it:

```
% RUN ISOMAP ALGORITHM
options.dims = 1:2;
Y = isomap(distmat(X'), 'k', K, options);
```

And the images I visualized in the report is :

And the way the swiss roll develop is :

Figure 8: The images I visualized

The Matlab Codes named *distmat* and *swissroll2* in this section is shown in Sec.3.3, Sec3.4.

# 2 Gaussian Mixture Model

In this section, I experience with the mixture Gaussians model, a classification model. Using EM algorithm to maximize the log likelihood function.
Besides, I also use k-means method to change the initialization and optimize both convergence speed and log likelihood function.

## 2.1 EM for Mixture of Gaussians

In this section, we consider a special case of a Gaussian mixture model, in which the covariance matrices $\Sigma_k$ of the omponents are all constrained to have a common value $\Sigma$, i.e. $\Sigma_k = \Sigma$, for all k. Now we derive the EM equations for maximizing the likelihood function under such a model.

$$\begin{aligned} P(x) &= \sum_{k=1}^{K} \pi_k N(x|\mu_k, \Sigma_k) \\ &= \sum_{k=1}^{K} \pi_k N(x|\mu_k, \Sigma) \\ &= \sum_{k=1}^{K} \pi_k \frac{1}{\sqrt{(2\pi)^k |\Sigma|}} exp\{-\frac{1}{2}(x-\mu_k)^T \Sigma^{-1}(x-\mu_k)\} \end{aligned}$$

The GMM log likelihood function is :

$$\begin{aligned} P(X|\Phi) &= \sum_{i=1}^{N} log \sum_{k=1}^{K} \pi_k N(X_i|\mu_k, \Sigma) \\ &= \sum_{i=1}^{N} log \sum_{k=1}^{K} \pi_k \frac{1}{\sqrt{(2\pi)^k |\Sigma|}} exp\{-\frac{1}{2}(x-\mu_k)^T \Sigma^{-1}(x-\mu_k)\} \end{aligned}$$

And what we should do now using EM algorithm is to find the best $\mu_k, \Sigma, \pi_k$ for the maximum of GMM log likelihood function. They could be easily achieved.

The E step(Density Estimation):

$$\gamma(i,k) = \frac{\pi_k N(X_i|\mu_k, \Sigma)}{\sum_{j=1}^{k} \pi_j N(X_i|\mu_j, \Sigma)}$$

i = 1, 2, ..., N and k = 1, 2, ..., K
where $\gamma(i,k)$ stands for the probability of i th occurs when it beloings to the k th cluster

The M step(Maximum the GMM log likelihood funcyion):

$$\begin{aligned} \mu_k &= \frac{\sum_{i=1}^{N} \gamma(i,k) X_i}{\sum_{i=1}^{N} \gamma(i,k)} \\ \Sigma &= \frac{\sum_{i=1}^{N} \gamma(i,k)()X_i - \mu_k)(X_i - \mu_k)^T}{\sum_{i=1}^{N} \gamma(i,k)} \\ \pi_k &= \frac{\sum_{i=1}^{N} \gamma(i,k)}{N} \end{aligned}$$

## 2.2 Mixtures of Gaussians

Read and understand the codes of GMM:

File *mogEM.m*: Performs EM for a mixture of K axis-aligned Gaussians. The program does a little change in E step to make the algorithm simplified.

File *mogLogProb.m*: Computes the log-probability of data under a MoG model.

File *kmeans.m*: Computes the means of clusters using k-means algorithm for initialization of a MoG model.

File *distmat.m*: Contains a function that efficiently computes pairwise distances between sets of vectors, which is also be used in swissroll in the previous section, Sec.1.3.

## 2.3 Training

First take a look at some of the examples of handwritten 2's and 3's to make sure that I have transferred the data properly. I wrote a function to do this:

```
function [] = plot_digit(x, ncol, nrow)
%   plot the digits as images
%   By tianran Zhang

if nargin < 3
  [~, k] = size(x);
  ncol = 2;
  nrow = k/ncol;
end
figure;
imagesc( cell2mat(squeeze(num2cell(reshape(x, [16, 16, ncol, nrow]),...
                          [1,2]))));
colormap gray;
end
```

And After running:

```
>> plot_digit(train2, 20, 15)
>> plot_digit(train3, 20, 15)
```

We got the images of handwritten 2's and 3's:

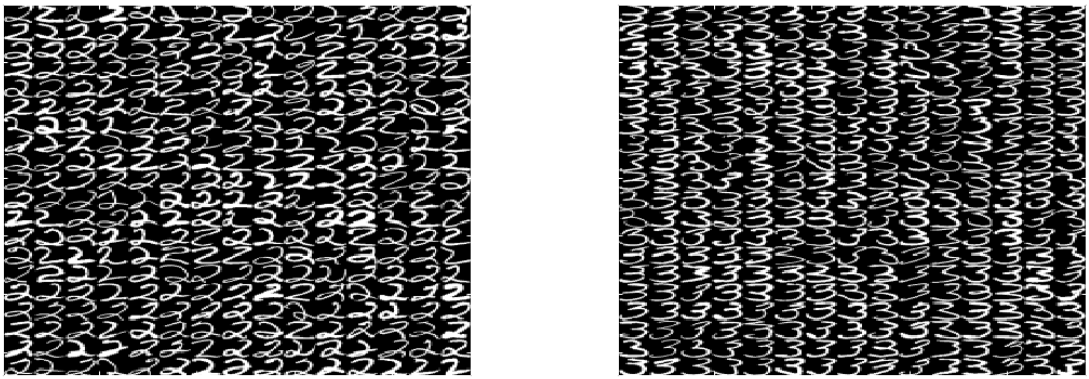

Figure 9: The handwritten 2's and 3's

Now, for each training set separately, we train a mixture of Gaussians using the code in mogEM.m ( Set k = 2 , minimum variance = 0.01, training iteration = 20):

```
load digits;

[p2,mu2,vary2,logProbX2] = mogEM(train2, 2, 20, 0.01, 0);
[p3,mu3,vary3,logProbX3] = mogEM(train3, 2, 20, 0.01, 0);
```

And the plots we got is shown bellow:

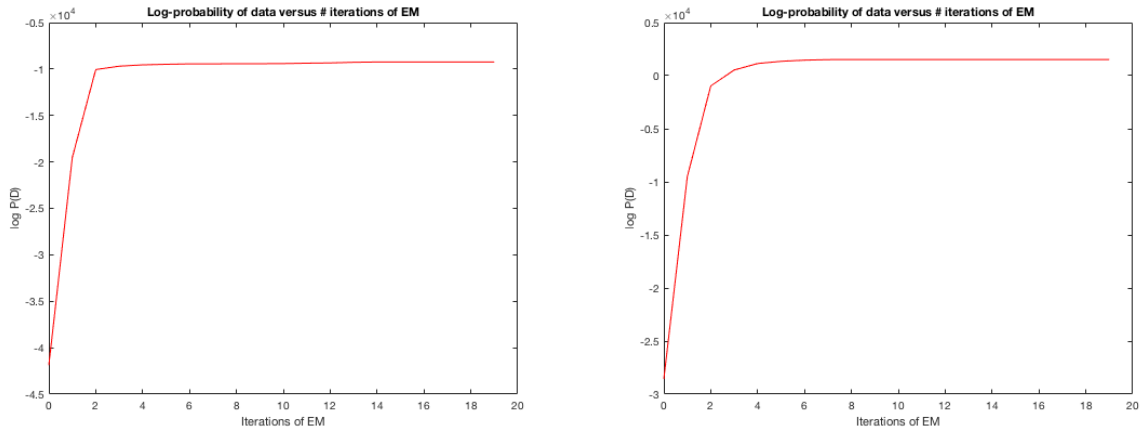


Figure 10: The log probability of handwritten 2's and 3's during the training.

Now let's experiment with the parameter settings. I wrote a script to find out the suitable randConst in MoG-EM:

```
load digits;

logProb2 = zeros(10, 1);
logProb3 = zeros(10, 1);
t = 0;
for k = 1:10
[p2,mu2,vary2,logProbX2] = mogEM(train2, 2, 20, 0.01, 0, k);
[p3,mu3,vary3,logProbX3] = mogEM(train3, 2, 20, 0.01, 0, k);

t = t+1;
logProb2(t) = logProbX2(20);
logProb3(t) = logProbX3(20);
end

% print the log-prob against k
figure;
plot(1:10, logProb2);
xlabel('k');
ylabel('log-prob of X');

figure;
plot(1:10, logProb3);
xlabel('k');
ylabel('log-prob of X');
```

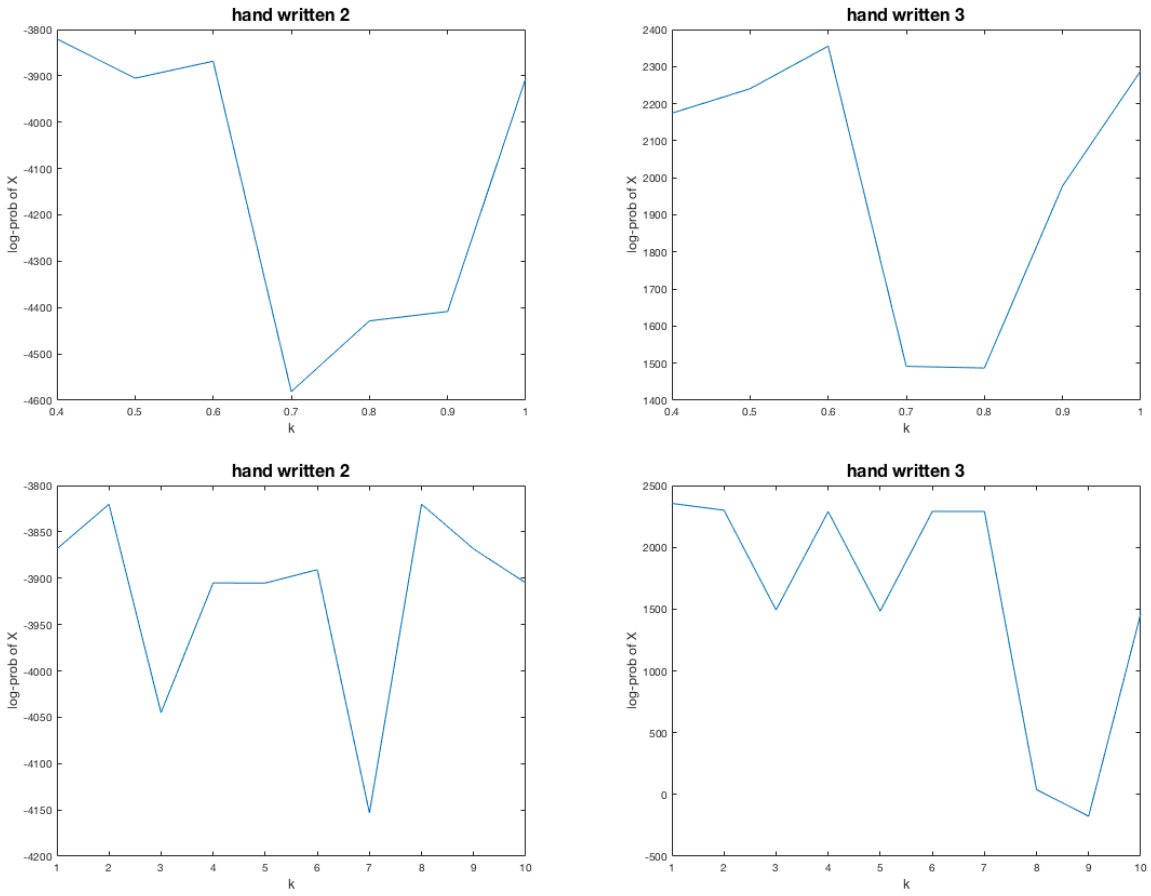


Figure 11: the log-prob against k

From the plot we could see that randConst = 1 may be the best choice. So from now on, I set randConst = 1 as a const.

For each model, I use the function *plot_digit* I have written before to show both the mean vector and variance vector as images.:

```
plot_digit(mu2, 1, 2);
plot_digit(vary2, 1, 2);
plot_digit(mu3, 1, 2);
plot_digit(vary3, 1, 2);
```

And the images are:

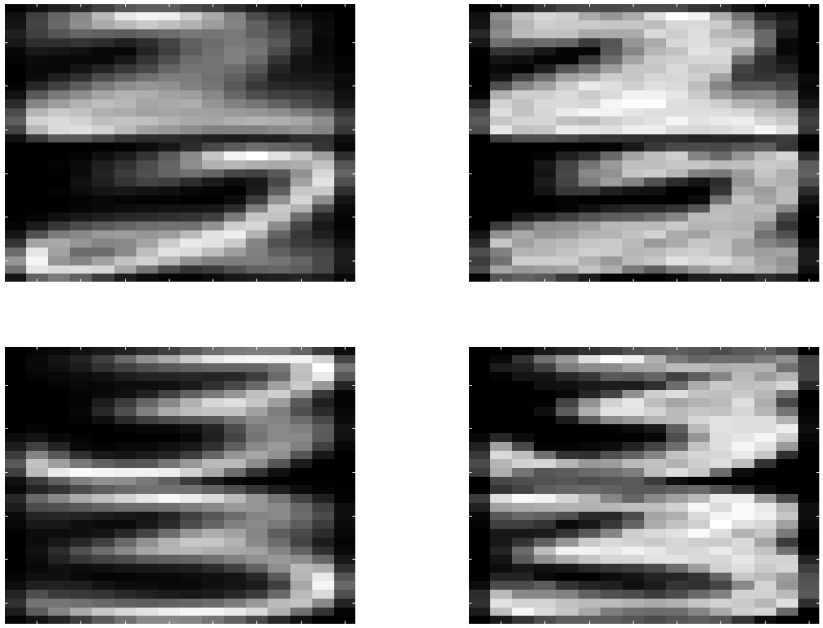

Figure 12: The mean and variance vector of 2 and 3(The left ones are mean vector images while the right ones are variance vector images)

The mixing proportions for the clusters within each model are :

```
% print the proportion for the clusters within each model
fprintf('proportion of train2:\n');
disp(p2);
fprintf('proportion of train3:\n');
disp(p3);
```

The outcomes are:

```
        proportion of train2:
        0.5000
        0.5000

        proportion of train3:
        0.5333
        0.4667
```

And the $logP(TrainingData)$ for each model is :

```
% Provide logP(TrainingData) for each model
fprintf('logP(Train2): %f\n', logProbX2(end));
fprintf('logP(Train3): %f\n', logProbX3(end));
```

The outcomes are:

```
logP(Train2): -3857.659492
logP(Train3): 2355.063823
```

The Matlab code of this section named *run_q3* is shown in Sec.3.5

## 2.4 Initializing a mixture of Gaussians with k-means

To make the MOG model become quicker, I change the initializtion of the means in *mogEM.m* with k-means algorithm (Use 5 iterations of k-means):

```
% Change the random initialization with k-means algorithm, use 5
% iterations.
mu = kmeans(x, K, 5);
```

Next train a MoG model with 20 components on all 600 training vectors(both 2's and 3's) with both the original initialization and the one based on k-means. The speed of the convergence is shown bellow:

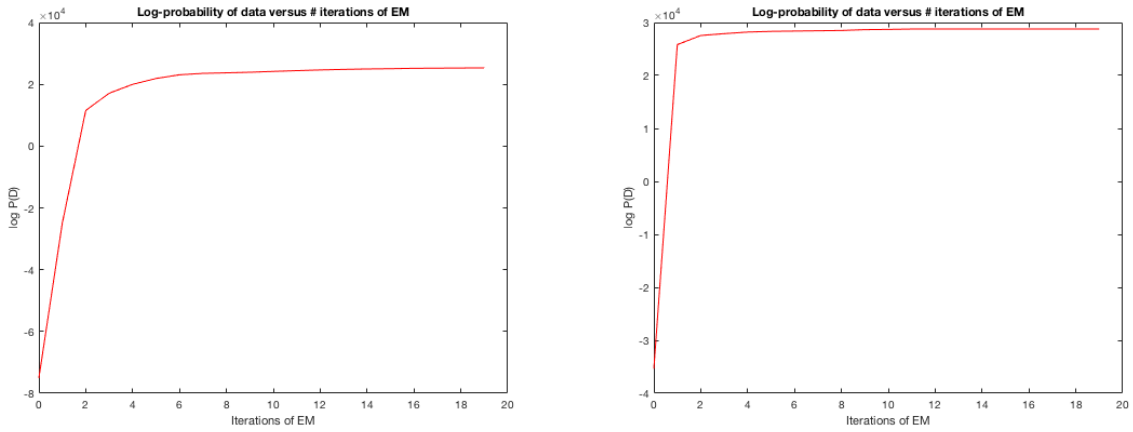


Figure 13: The speed of the vonvergence (The left one is the speed with original initialization and the right one is the speed based on k-means)

It is appreantly that the speed based on k-means is lagger than that with the original initialization.

The final log-prob resulting from the two initialization methods are :

```
% the log-prob with the original initialization
>> run_qmean
logP(x): 25329.564391

% The log-prob with k-means initialization
>> run_qmean
logP(x): 28771.404924
```

The log-prob with k-means initialization is bigger than that with the original initialization. So using k-means method to generate the initialization of MoG works better in both speed and log probability.

## 2.5 Classification using MoGs

Now we use the trained mixture models for classification. (We set d=1 to the 2's and d=2 to the 3's).
I trained 2's and 3's models with 2, 5, 15, 25 components. For each number, computes $P(d = 1|x)$ *and* $P(d = 2|x)$ based on the outputs of the two trained models, and use it to classify the training, text, validation examples.
For each compunents number:

```
% Train a MoG model with K components for digit 2 and 3
[p2,mu2,vary2,logProbX2] = mogEM(train2, K, 20, 0.01, 0);
[p3,mu3,vary3,logProbX3] = mogEM(train2, K, 20, 0.01, 0);

% Caculate the probability P(d=1|x) and P(d=2|x),
% classify examples, and compute the error rate
%Training Part
logProb_train2 = mogLogProb(p2, mu2, vary2, train2);
logProb_train3 = mogLogProb(p3, mu3, vary3, train3);
errorTrain(i) = 1 - (sum(logProb_train2 > 0.5) + ...
sum(logProb_train3 > 0.5))/(size(train2, 2)/2 + size(train3, 2));

%Valid Part
logProb_valid2 = mogLogProb(p2, mu2, vary2, valid2);
logProb_valid3 = mogLogProb(p3, mu3, vary3, valid3);
errorValidation(i) = 1 - (sum(logProb_valid2 > 0.5) + ...
sum(logProb_valid3 > 0.5))/(size(valid2, 2)/2 + size(valid3, 2));

%Test Part
logProb_test2 = mogLogProb(p2, mu2, vary2, test2);
logProb_test3 = mogLogProb(p3, mu3, vary3, test3);
errorTest(i) = 1 - (sum(logProb_test2 > 0.5) + ...
sum(logProb_test3 > 0.5))/(size(test2, 2)/2 + size(test3, 2));
```

After the classification, Plot the error rate:

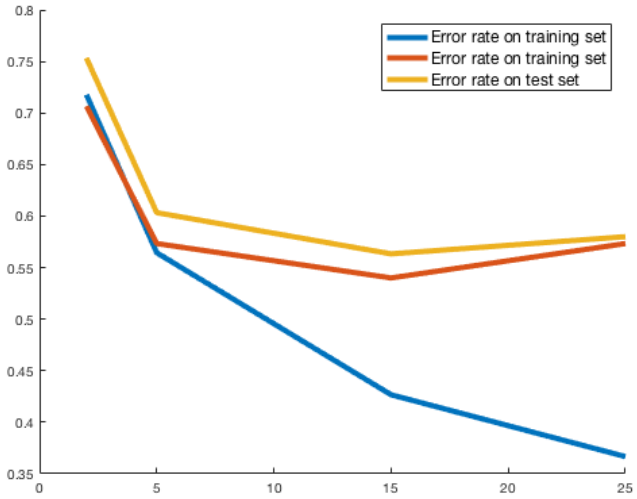


Figure 14: The error rate versus mixture components number

From the plot, we noticed some facts:

1. The error rates on the training sets generally decrease as the number of clusters increases.

As the number of clusters increases, we divided the training data into more groups. When we do the classification with training sets, it fits in the clusters easily, but it leads to over-fitting. It could be seen on the plot: only the error rates on training sets decrease, the error rates on validation sets and test sets have the tends to increase as the number of clusters increases.

2. Examine the error rate curve for the test set.

We could see that the curve of error rate on test set went down at first, then went smoothly, and went up when the number of clusrers is big. As I have mentioned before, when the number of clusters is small, then as the number grows bigger, the model works better on test sets. But if the number of clusters is big, it divided the training sets into too many groups, that a little change on hand wrriten 2's and 3's will make them a new feature in a new group. This leads to over-fitting, thus the error rate curve on test set goes down first and goes up when the number of clusters is big.

3. Choose a particular model from the experiments.

Since I want to achieve the lowest error rate on the newimages I will receive, that is, make my new test error as small as possible. As I have mentioned in fact 1 and 2, I want the number clusters as big as possible but should prevent the over-fitting. From the plot, I think that 15 clusters may be the best choice.

# 3 Matlab Codes

## 3.1 find_eigenvector.m

```
load freyface.mat
X = double(X);

[m, N] = size(X);
% lambda_un is the eigenvalues without removing the mean
% with a descending order.
% Vun is the eigenvectors correspond to lambda_un.
[Vun, Dun] = eig(X*X'/N);
[lambda_un, order] = sort(diag(Dun), 'descend');
Vun = Vun(:, order);

% lambda_ctr is the eigenvalues with removing the mean
% with a descending order.
% Vctr is the eigenvectors correspond to lambda_ctr.
Xctr = X - repmat(mean(X, 2), 1, N);
[Vctr, Dctr] = eig(Xctr*Xctr' /N);
[lambda_ctr, order] = sort(diag(Dctr), 'descend');
Vctr = Vctr(:, order);

figure;
plot(1:m, lambda_ctr(1:m, 1));
xlabel('number of k');
ylabel('lambda\_un');

figure;
plot(1:100, lambda_ctr(1:100, 1));
xlabel('number of k');
ylabel('lambda\_un');

% p  : the number of k
% p1 : the percentage of the first p lambdas take  in all lambdas
% S0 : the weights of the first p lambdas
% S1 : the weights of all lambdas
p = 0;       p1 = 0;
s0 = 0;      s1 = sum(lambda_ctr);
while (p1 <0.95)
p = p + 1;
s0 = s0 + lambda_ctr(p);
p1 = s0/s1;
end
```

### 3.2 ReconstructFace

```
load freyface.mat
X = double(X);

[m, N] = size(X);

Xctr = X - repmat(mean(X, 2), 1, N);
[Vctr, Dctr] = eig(Xctr*Xctr' /N);
[lambda_ctr, order] = sort(diag(Dctr), 'descend');
Vctr = Vctr(:, order);
V = Vctr(:, 1:2);
Yctr = V' * X;

y1 = randi([1,255], 560, 1);
figure;
showfreyface(y1);
figure;
showfreyface(reconstruct(V'*(R-mean(X,2)),V)+mean(X,2));

% Adding noise (choosing the 100 th Fray's face)
X_noise = X(:, 50) + 10*randn(m,1);
figure;
showfreyface(X_noise);
figure;
showfreyface(reconstruct(V'*(X_noise-mean(X,2)),V)+mean(X,2));
```

### 3.3 distmat

```
% SWISS ROLL DATASET

N=2000;
K=12;
d=2;

clf;
colordef none;
colormap jet;
set(gcf,'Position',[200,400,620,200]);

% PLOT TRUE MANIFOLD
tt0 = (3*pi/2)*(1+2*[0:0.02:1]); hh = [0:0.125:1]*30;
xx = (tt0.*cos(tt0))'*ones(size(hh));
yy = ones(size(tt0))'*hh;
zz = (tt0.*sin(tt0))'*ones(size(hh));
```

```
cc = tt0'*ones(size(hh));

subplot(1,3,1); cla;
surf(xx,yy,zz,cc);
view([12 20]); grid off; axis off; hold on;
lnx=-5*[3,3,3;3,-4,3]; lny=[0,0,0;32,0,0]; lnz=-5*[3,3,3;3,3,-3];
lnh=line(lnx,lny,lnz);
set(lnh,'Color',[1,1,1],'LineWidth',2,'LineStyle','-','Clipping','off');
axis([-15,20,0,32,-15,15]);

% GENERATE SAMPLED DATA
tt = (3*pi/2)*(1+2*rand(1,N));  height = 21*rand(1,N);
X = [tt.*cos(tt); height; tt.*sin(tt)];

% SCATTERPLOT OF SAMPLED DATA
subplot(1,3,2); cla;
scatter3(X(1,:),X(2,:),X(3,:),12,tt,'+');
view([12 20]); grid off; axis off; hold on;
lnh=line(lnx,lny,lnz);
set(lnh,'Color',[1,1,1],'LineWidth',2,'LineStyle','-','Clipping','off');
axis([-15,20,0,32,-15,15]); drawnow;

% RUN ISOMAP ALGORITHM
options.dims = 1:2;
Y = isomap(distmat(X'), 'k', K, options);

%SCATTERPLOT OF EMBEDDING
figure(1);
subplot(1,3,3); cla;
temp = Y.coords{2};
scatter(temp(1,:),temp(2,:),12,tt,'+');
grid off;
set(gca,'XTick',[]); set(gca,'YTick',[]);
```

### 3.4 swissroll

```
% function [dist] = distmat(v1, v2)
%
% Calculates pairwise distances between vectors. If v1 and v2 are both
% provided, a size(v1, 1) by size(v2, 1) matrix is returned, where the
% entry at (i,j) contains the Euclidean distance from v1(i,:) to v2(j,:).
% If only v1 is provided, squareform(pdist(v1)) is returned.

function [dist] = distmat(p, q, disttype)
p = p';
```

```
if nargin == 1
q = p;
else
q = q';
end

[d, pn] = size(p);
[d, qn] = size(q);

pmag = sum(p .* p, 1);
qmag = sum(q .* q, 1);
dist = repmat(qmag, pn, 1) + repmat(pmag', 1, qn) - 2*p'*q;
dist = sqrt(dist);
end
```

### 3.5 run_q3

```
load digits;
x = [train2, train3];
% Train a MoG model with 20 components on all 600 training vectors
% with both original initialization and your kmeans initialization.

[p2,mu2,vary2,logProbX2] = mogEM(train2, 2, 20, 0.01, 0);
[p3,mu3,vary3,logProbX3] = mogEM(train3, 2, 20, 0.01, 0);

% plot mean and variance vectors of each data
plot_digit(mu2, 1, 2);
plot_digit(vary2, 1, 2);
plot_digit(mu3, 1, 2);
plot_digit(vary3, 1, 2);

% print the proportion for the clusters within each model
fprintf('proportion of train2:\n');
disp(p2);
fprintf('proportion of train3:\n');
disp(p3);

% Provide logP(TrainingData) for each model
fprintf('logP(Train2): %f\n', logProbX2(end));
fprintf('logP(Train3): %f\n', logProbX3(end));
```