CSC411 A3

Olga (Geya) Xu, 999700658

November 2014

1 Mixtures of Bernoulli distributions

1.1 Responsibilities

We know that at *E-step* $r_{nk} = \mathbb{E}[z_{nk}]$ given data point $\mathbf{x_n}$, mixing proportions π and the Bernoulli parameters μ.

$$r_{nk} = \mathbb{E}[z_{nk}|\mathbf{x_n}]$$

$p(z_{nk}|\mathbf{x_n})$ given μ and π

$$= \sum_{z_{nk} \in \{0,1\}} z_{nk} \cdot p(z_{nk}|\mathbf{x_n})$$
$$= p(z_{nk} = 1|\mathbf{x_n})$$

using Bayes rules

$$= \frac{p(z_{nk} = 1, \mathbf{x_n})}{p(\mathbf{x_n})}$$

using Bayes rules

$$= \frac{p(z_{nk} = 1)p(\mathbf{x_n}|\mathbf{z_{nk}} = \mathbf{1}, \mu_\mathbf{k})}{\sum_j^K \pi_j p(\mathbf{x_n}|\mu_\mathbf{j})}$$

therefore,

$$= \frac{\pi_k p(\mathbf{x_n}|\mu_\mathbf{k})}{\sum_j^K \pi_j p(\mathbf{x_n}|\mu_\mathbf{j})}$$

1.2 Estimates for the Bernoulli parameters

Given $\mathbb{E}_\mathbf{z}[logp(\mathbf{x}, \mathbf{z}|\mu, \pi)] = \sum_\mathbf{n}^\mathbf{N} \sum_\mathbf{k}^\mathbf{K} \mathbf{r_{nk}}\{\mathbf{log}\pi_\mathbf{k} + \sum_\mathbf{i}^\mathbf{D}[\mathbf{x_{ni}log}\mu_\mathbf{ki} + (\mathbf{1} - \mathbf{x_{ni}})\mathbf{log}(\mathbf{1} - \mu_\mathbf{ki})]$,
let $\mathbf{g}(\mu_\mathbf{k}) = \mathbb{E}_\mathbf{z}[\mathbf{logp}(\mathbf{x}, \mathbf{z}|\mu, \pi)]$

$$\nabla \mathbf{g}(\mu_\mathbf{k}) = \begin{pmatrix} \frac{\partial \mathbf{g}}{\partial \mu_{k1}} \\ \vdots \\ \frac{\partial \mathbf{g}}{\partial \mu_{kD}} \end{pmatrix}$$

We want $\nabla \mathbf{g}(\mu_\mathbf{k}) = \mathbf{0}$, we need $\forall i \in \mathbb{Z}, i \in [1, D]$, $\frac{\partial \mathbf{g}}{\partial \mu_{ki}} = 0$

$$
\begin{aligned}
\frac{\partial \mathbf{g}}{\partial \mu_{ki}} &= \frac{\partial \mathbb{E}_\mathbf{z}[logp(\mathbf{x}, \mathbf{z}|\mu, \pi)]}{\partial \mu_{ki}} \\
&= \sum_n^N r_{nk}[\frac{x_{ni}}{\mu_{ki}} - \frac{1 - x_{ni}}{1 - \mu_{ki}}] \\
&= \sum_n^N r_{nk}[\frac{x_{ni} - \mu_{ki}}{\mu_{ki}(1 - \mu_{ki})}] \\
&\implies \sum_n^N r_{nk}(x_{ni} - \mu_{ki}) = 0 \\
&\implies \sum_n^N r_{nk}x_{ni} - \sum_n^N r_{nk}\mu_{ki} = 0 \\
&\implies \sum_n^N r_{nk}x_{ni} - N_k\mu_{ki} = 0 \\
&\implies \frac{\sum_n^N r_{nk}x_{ni}}{N_k} = \mu_{ki}
\end{aligned}
$$

Substitute back into the gradient equation($\nabla \mathbf{g} = \mathbf{0}$), we have $\frac{\sum_n^N r_{nk}x_{ni}}{N_k} = \mu_{ki}, \forall i$.

$$
\begin{aligned}
&\implies \mu_k = \frac{1}{N_k}\sum_n^N r_{nk}\mathbf{x_n} \\
&\implies \mu_k = \bar{\mathbf{x}}_k
\end{aligned}
$$

1.3 Mixing proportion

Let $f(\pi) = \mathbb{E}_{\mathbf{z}}[logp(\mathbf{x}, \mathbf{z}|\mu, \pi)]$ and $g(\pi) = \sum_k pi_k = 1$ (constraint) and $\Lambda(\pi, \lambda) = f(\pi) + (\lambda(g(\pi) - 1))$. Then, $\frac{\partial \Lambda}{\partial \pi_k} = \sum_n^N \frac{r_{nk}}{\pi_k} + \lambda = 0$ and $\frac{\partial \Lambda}{\partial \lambda} = g(\pi) - 1 = 0$. Now we have the following:

$$\lambda = -\sum_n^N r_{nk} \frac{1}{\pi_k}$$
$$\implies -\pi_k \lambda = N_k$$

take summation over K on both sides of the equation ($\sum_k^K$)

$$\implies \sum_k^K -\pi_k \lambda = \sum_k^K N_k$$

we have $\sum_k^K \pi_k = 1$

$$\implies -\lambda = \sum_k^K \sum_n^N r_{nk}$$

we have $r_{nk} = \frac{\pi_k p(x_n|\mu_k)}{\sum_{j=1}^k \pi_j p(x_n|\mu_j)}$

$$\implies -\lambda = \sum_k^K \sum_n^N \frac{\pi_k p(x_n|\mu_k)}{\sum_j^k \pi_j p(x_n|\mu_j)}$$
$$\implies -\lambda = \sum_n^N \frac{\sum_k^K \pi_k p(x_n|\mu_k)}{\sum_j^k \pi_j p(x_n|\mu_j)}$$
$$\implies -\lambda = \sum_n^N 1$$
$$\implies \lambda = -N$$

since $-\pi_k \lambda = N_k$

$$\implies \pi_k = \frac{N_k}{N}$$

2 Training

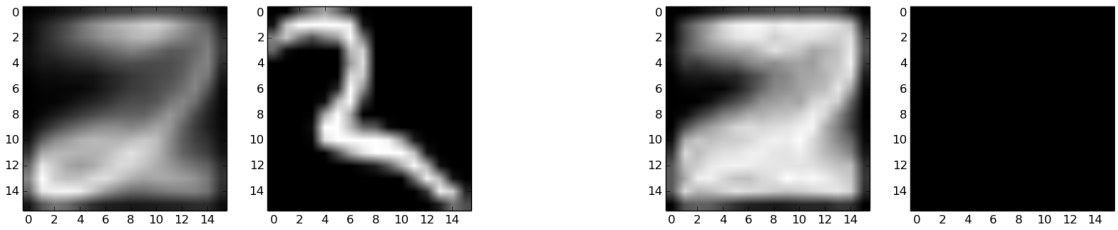
2.1 Choosing *randConst*

2.1.1 Table of data for different *randConst*

Different trial	Trial1	Trial2	Trial3	Trial4	Trial5	Trial6
$randCost = 1$						
# of iters to converge for 2	9	14	13	13	10	15
log prob at convergence for 2	-3868.40634	-4422.74978	-3905.11407	-3905.11407	-3820.21842	-4139.67375
# of iters to converge for 3	10	25	23	28	23	25
log prob at convergence for 3	2290.28350	2264.79180	2288.90081	2264.08976	2264.08976	1765.55304
$randCost = 10$						
# of iters to converge for 2	13	21	27	11	23	10
log prob at convergence for 2	-3837.19828	-3994.33137	-3989.63917	-4364.94553	-3857.65949	-3868.40634
# of iters to converge for 3	22	12	18	9	12	18
log prob at convergence for 3	2288.90081	2355.06382	1523.36632	2355.06382	1502.08240	2288.90081
$randCost = 0.5$						
# of iters to converge for 2	21	6	14	11	16	10
log prob at convergence for 2	-3890.79141	-3820.21842	-3905.26719	-4139.67375	-4408.42268	-3820.21842
# of iters to converge for 3	27	19	17	14	19	12
log prob at convergence for 3	2355.06382	2290.28350	2264.08976	2290.28350	2288.90081	1722.32401

As we can see from the table above, different *randConst*'s do not affect the log probability of data nor convergence rate a lot. However, from the example below we can see the size of *randConst* matters.

2.1.2 Local optimal example



(a) Mean for train2, $randConst = 0.2$ (b) Variance for train2, $randConst = 0.2$

Figure 1: The mean vector(s) and variance vector(s)

It took 5 iterations for the output to converge in this case.

```
Iter 0 logProb -440300.27904
Iter 1 logProb -74639.01433
Iter 2 logProb -12885.39025
Iter 3 logProb -12682.43934
Iter 4 logProb -12682.43933
Iter 5 logProb -12682.43933
Iter 6 logProb -12682.43933
```

However, we can see from *figure1* that the output is stuck in a local optimum solution. This is because the *randConst* we chose is too small(0.2). From, $mu = mn + np.random.randn(N, K) * (np.sqrt(vr)/randConst)$, we may assume that *randConst* acts like an amplifier for the $np.random.randn(N, K)$. The smaller *randConst* is, the larger amplification will be. Small *randConst* will cause *mu* to be far away from the data mean. This implies that the higher chance the program will halt at a local optimum point.
In conclusion, *1* is a valid choice for *randConst*. Within all the trials I ran, $randConst = 1$ worked fine.

2.2 Model: Train2

2.2.1 Parameter setting:

$K = 2$, $iters = 12$, $min\_var = 0.01$, mu (the original initialization with $randConst = 1$)

2.2.2 Output for one of my best trial: (log prob for each iteration)

```
Iter 0 logProb -38472.18856
Iter 1 logProb -18250.47080
Iter 2 logProb -7969.69885
Iter 3 logProb -5408.11841
Iter 4 logProb -4340.39165
Iter 5 logProb -4081.58796
Iter 6 logProb -3944.19421
Iter 7 logProb -3893.68644
Iter 8 logProb -3832.21736
Iter 9 logProb -3820.21896
Iter 10 logProb -3820.21842
Iter 11 logProb -3820.21842
```

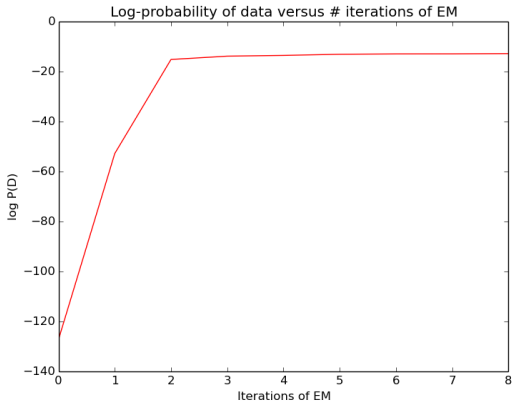


Figure 2: Log-probabilty of data vs. number of iterations of EM

2.2.3 With the same parameter setting, the program generates different results each time (except for *iter*). Below listed some of my trials.

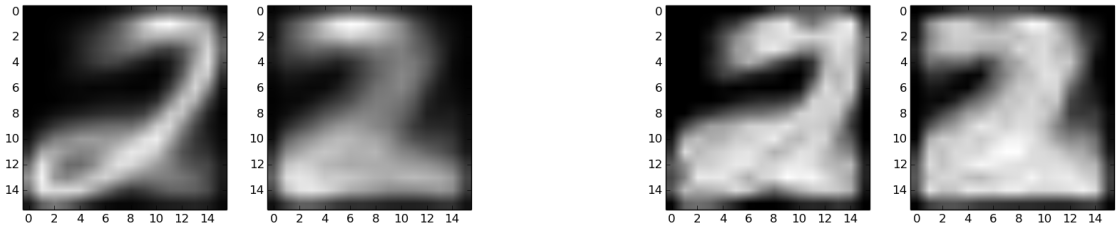
Different trial	Trial1	Trial2	Trial3	Trial4	Trial5	Trial6
# of iters to converge	13	14	13	13	10	15
log prob at convergence	-3837.19828	-4422.74978	-3905.11407	-3905.11407	-3820.21842	-4139.67375
Mixing proportion 1	0.49335392	0.48335698	0.45668994	0.53333334	0.47317614	0.48999999
Mixing proportion 2	0.50664608	0.51664302	0.54331006	0.46666666	0.52682386	0.51000001

2.2.4 log p (train2) and the mixing proportion

Normally, the number of iterations for the model to converge ranges from 10 to 15 and the $logp(train2)$ at convergence ranges from -3820.21842 to -4422.74978.
The mixing proportion of each cluster is even. I got almost 1:1 ratio between the two clusters each time.

2.2.5 Mean and variance

The means for the two clusters each illustrates one out of two common ways how people hand-write '2'.
The variances for the two clusters each illustrates how people's writing of 2's deviates from the two average way(since the mixing proportion is about 1:1 ration) of writing.



(a) Mean for train2

(b) Variance for train2

Figure 3: The mean vector(s) and variance vector(s)

2.3 Model: Train3

2.3.1 Parameter setting:

$K = 2$, $iters = 13$, $min\_var = 0.01$, mu (the original initialization with $randConst = 1$)

2.3.2 Output: (log prob for each iteration)

```
Iter 0 logProb -30376.44462
Iter 1 logProb -10437.17786
Iter 2 logProb 1052.57472
Iter 3 logProb 1665.61376
Iter 4 logProb 1834.56362
Iter 5 logProb 1936.79120
Iter 6 logProb 2004.25089
Iter 7 logProb 2058.32881
Iter 8 logProb 2107.44514
Iter 9 logProb 2203.55460
Iter 10 logProb 2239.30400
Iter 11 logProb 2239.30624
Iter 12 logProb 2239.30624
```

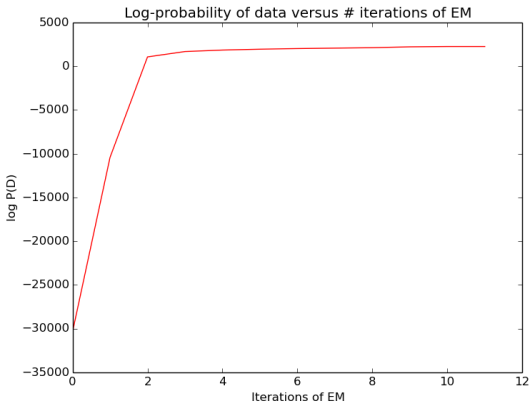


Figure 4: Log-probabilty of data vs. number of iterations of EM

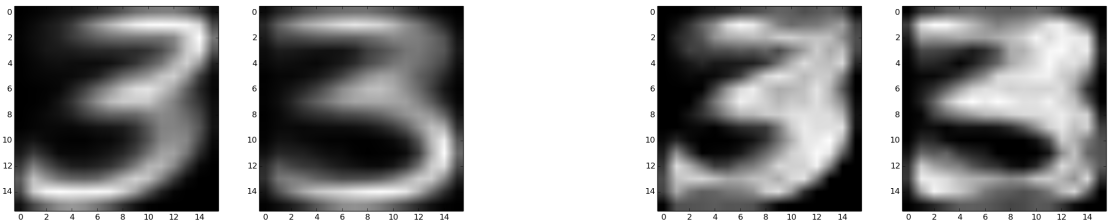
2.3.3 With the same parameter setting, the program generates different results each time (except for *iter*). Below listed some of my trials.

Different trial	Trial1	Trial2	Trial3	Trial4	Trial5	Trial6
# of iters to converge	10	25	23	28	23	25
log prob at convergence	2290.28350	2264.79180	2288.90081	2264.08976	2264.08976	1765.55304
Mixing proportion 1	0.47967269	0.50342948	0.5162365	0.54331006	0.51000001	0.53333116
Mixing proportion 2	0.52032731	0.49657052	0.4837635	0.45668994	0.48999999	0.46666884

2.3.4 log p (train3) and the mixing proportion

Normally, the number of iterations for the model to converge ranges from 10 to 25 and the $logp(train3)$ at convergence ranges from 1765.55304 to 2290.28350.
The mixing proportion of each cluster is even. I got almost 1:1 ratio between the two clusters each time.

2.3.5 Mean and variance



(a) Mean for train3

(b) Variance for train3

Figure 5: The mean vector(s) and variance vector(s)

The means for the two clusters each illustrates one out of two common ways how people hand-write '3'. The variances for the two clusters each illustrates how people's writing of 3's deviates from the two average way(since the mixing proportion is about 1:1 ration) of writing.

3 Initializing a mixture of Gaussians with k-means

3.1 Without k-means($randConst = 1$)

3.1.1 Parameter setting:

$K = 20$, $min\_var = 0.01$, mu (the original initialization with $randConst = 1$)

3.1.2 Trial data

I ran 18 trials of MOG with original initialization for mu and got the following results:

Different trial	Trial1	Trial2	Trial3	Trial4	Trial5	Trial6
# of iters to converge	30	35	29	25	33	20
log prob at convergence	23123.40160	25727.51822	22897.25553	27454.55034	24482.73141	25260.76403
Different trial	Trial7	Trial8	Trial9	Trial10	Trial11	Trial12
# of iters to converge	38	38	21	30	34	15
log prob at convergence	26052.17582	24947.90980	25278.64435	26479.61644	28126.28544	24014.76008
Different trial	Trial13	Trial14	Trial15	Trial16	Trial17	Trial18
# of iters to converge	31	21	41	31	37	32
log prob at convergence	26436.04200	24859.01881	26509.37466	25019.04188	26561.56203	25395.66665

In these 18 trials, **the average number of iterations needed for MOG converge without means is 30.0556**.

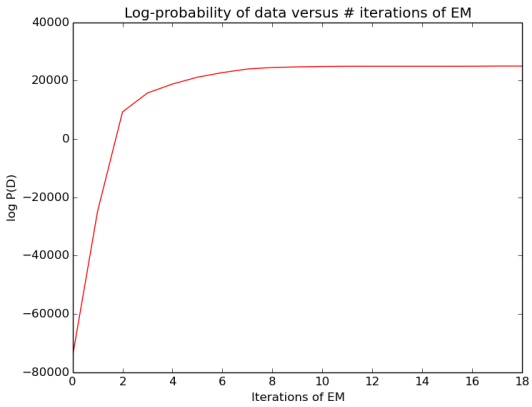


Figure 6: A typical without k-mean trial's graph: 20 iterations

3.2 With k-means ($kmeanIters = 5$)

3.2.1 Parameter setting:

$K = 20$, $min\_var = 0.01$, mu (with kmean setting where $kmeanIters = 5$)

3.2.2 Trial data

Different trial	Trial1	Trial2	Trial3	Trial4	Trial5	Trial6
# of iters to converge	21	17	14	9	13	19
log prob at convergence	28129.58984	27637.80480	28798.07786	29556.83176	28228.64393	28625.49894
Different trial	Trial7	Trial8	Trial9	Trial10	Trial11	Trial12
# of iters to converge	8	23	16	25	16	11
log prob at convergence	28824.37113	27567.05735	28834.92173	28695.59972	27323.85238	28503.69286
Different trial	Trial13	Trial14	Trial15	Trial16	Trial17	Trial18
# of iters to converge	10	11	13	32	16	21
log prob at convergence	29727.18618	29699.94198	29481.80547	29389.66338	30253.23854	26900.96145

In these 18 trials, **the average number of iterations needed for MOG converge with k-means is 16.3889**.

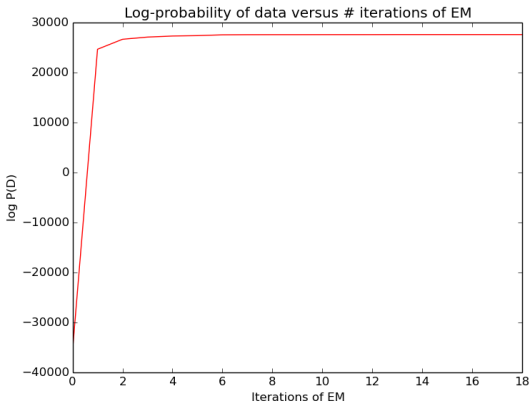


Figure 7: A typical with k-mean trial's graph: 20 iterations

3.3 Results

We can see that with k-mean the convergence rate is much faster and the log probability of x is much higher. With *randConst*, it is more likely for the result to be trapped in a local optimal. This would cause the probability of x become more unlikely and that's why the log probability of x without k-mean initialization is much smaller. However, k-means computation might be costly. The k-means algorithm we are using, *kmeans.py*, computes onlythe hard k-means which is not as computational expensive as the soft k-means algorithm. In general, initialization with k-means has faster speed of convergence and higher log probability of the input data.

4 Classification using MOGs

4.1 Error rate

I first ran my algorithm twice and got following error rate graphs:

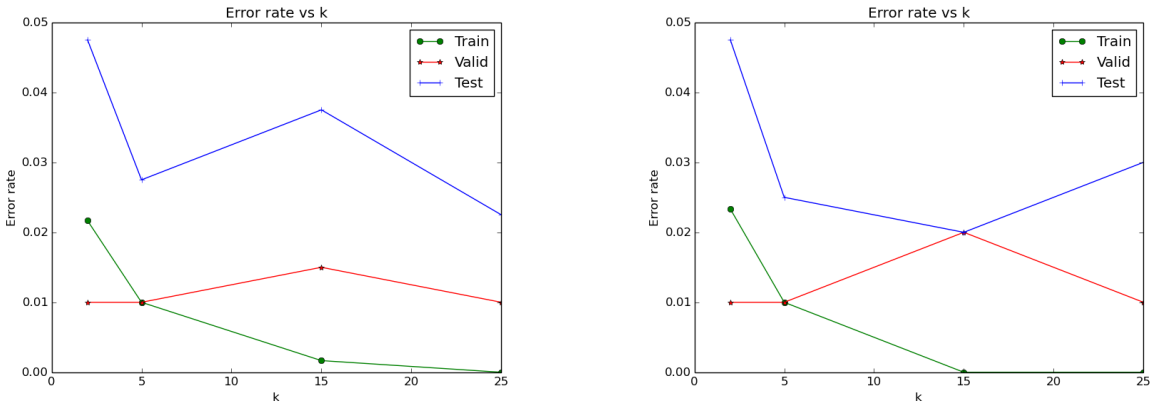


Figure 8: Error rate in train/valid/test for different ks

I noticed that the error rates varies. It is due to the different optimal the algorithm found at different run-time. Therefore, I chose to ran MOGs 20 times and average the error rate of those 20 trials.

4.2 Average error rate

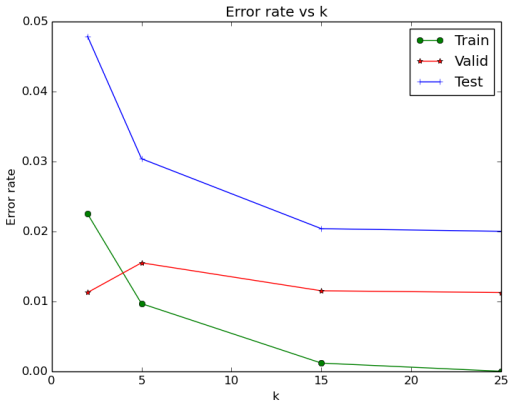


Figure 9: Average error rate for 20 runs

4.3 Error rate on training set

The error rates on the training sets generally decrease as the number of clusters increases. This is because we are doing learning on the training data. Increasing the number of clusters leads to better fitting of the model to the training data. The model would start to generate clusters for the outliers in the training data. Because the model starts to recognizing the outliers in the training data, the error rates on the training sets generally would decrease when the number of cluster increase.

4.4 Error rate on validation set/test set

The error rate starts decreasing then starts increasing as the number of clusters increases. This is because at first the model is learning various way of people's writing (decrease in classification error) and when the number of clusters increases to a large number the model starts to overfit the training data (increase classification error). Increasing the number of clusters, the model becomes more sensitive to the outliers in the training data. This flaw in the model can cause increment in the error rate for classifying validation and test set. The overfitting

problem seems more obvious on the validation set when I increase the number of clusters to 50. Below is a graph illustrating the overfitting problem on the validation set.

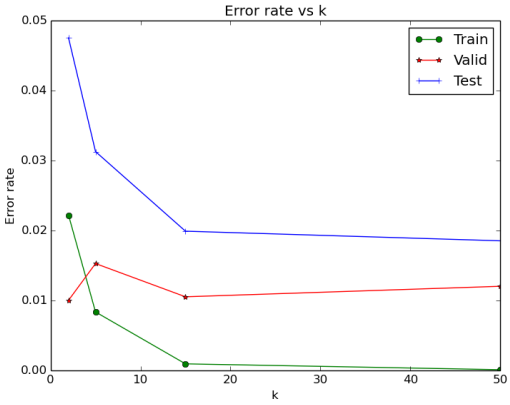


Figure 10: Average error rate for 20 runs when $k = 40$

When $k = 15$, the model seems to fit the validation set the best. When $k = 15$, the model's classification error rate on the validation set is about 1.2% and on the test set is about 2%. On the best case, my error rate on the validation set is 1.1%

5 Mixture of Gaussian vs Neural Network

5.1 $k = 15$ - the best k from question 4

I used the best parameter setting got from part 2 ($k = 15$) and got the following results.

5.1.1 Change in the input to hidden weights during training

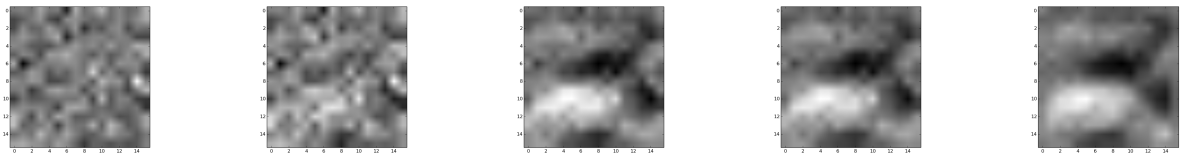
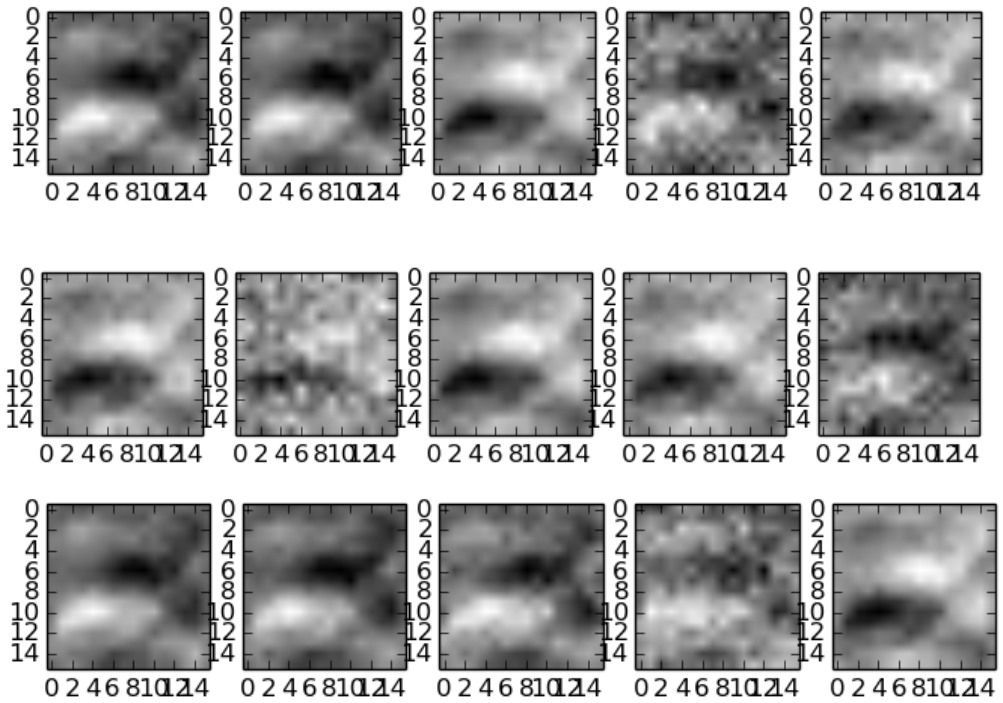


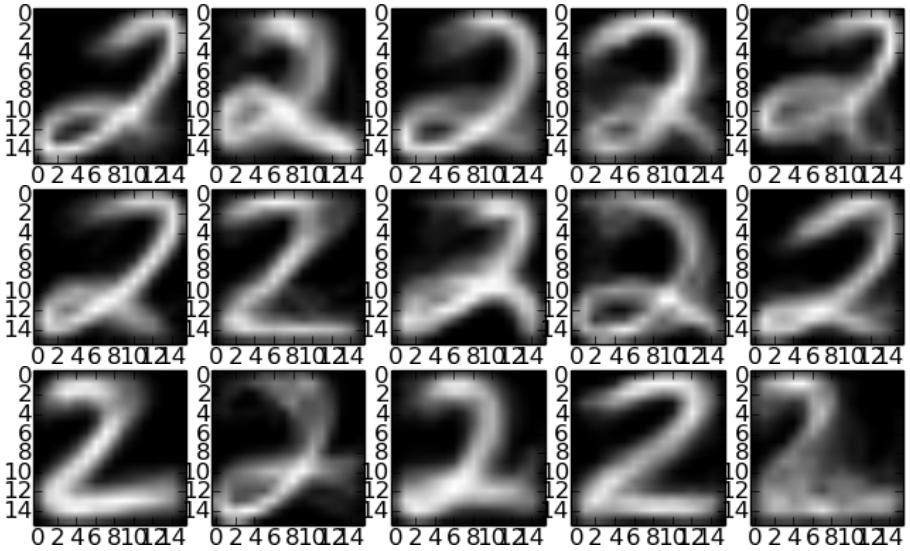
Figure 11: Change in the input to hidden weights every 200 iterations

The first one shows the initial weights; the second one is the weights learned after 200 iterations;...; the third one is the weights learned after 1000 iterations. As the learning progresses, the features of 3 become more and more obvious.

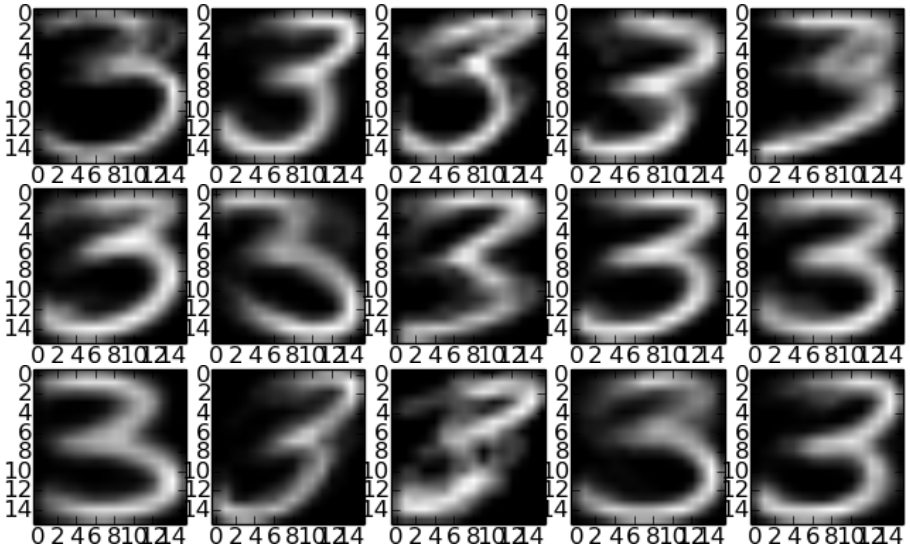
5.1.2 Visualization of all weights after learning for 15 hidden units



5.1.3 Visualization of all 15 cluster with data of 2s

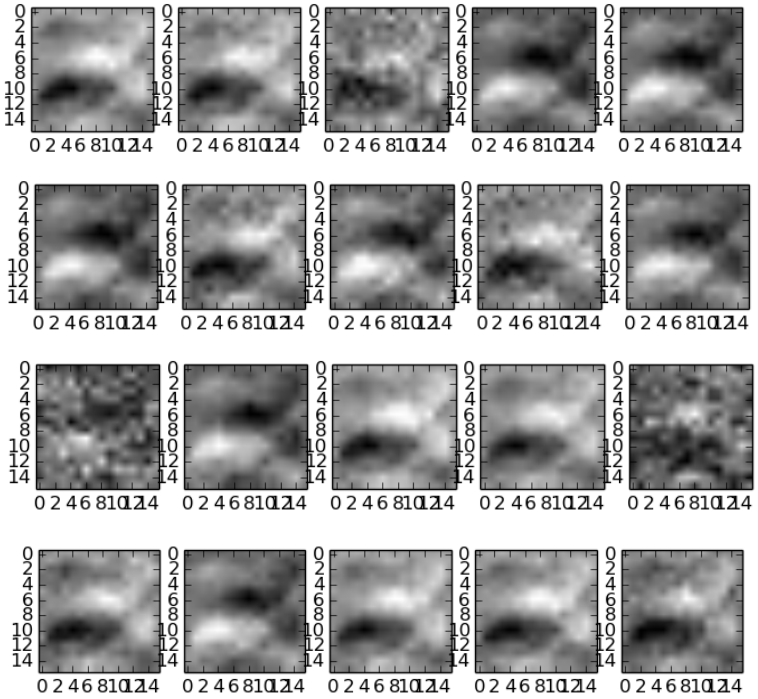


5.1.4 Visualization of all 15 cluster with data of 3s

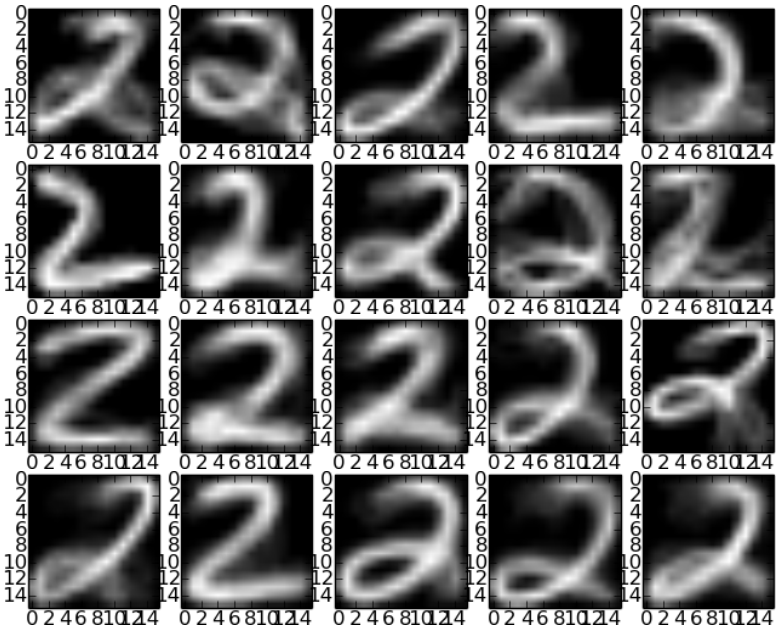


5.2 $k = 20$ - parameter setting from question 3

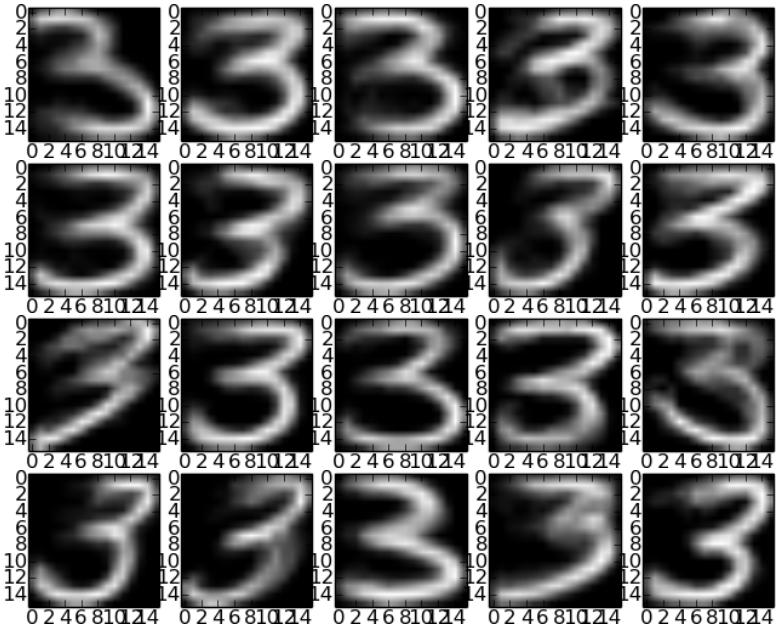
5.2.1 Visualization of all weights after learning for 20 hidden units



5.2.2 Visualization of all 20 clusters with data of 2s

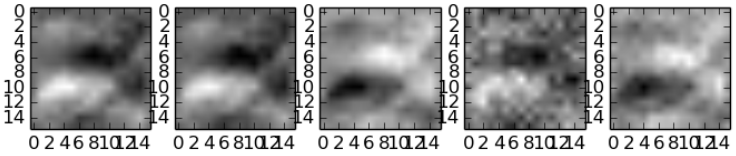


5.2.3 Visualization of all 20 clusters with data of 3s



5.3 Neural network

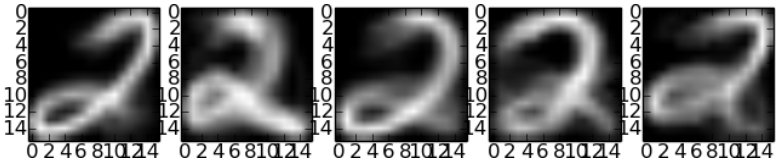
The visualizations of the weights show that neural network learns the feature of the training data. For example, here are a group weights learned. 3 and 2 very similar top part. However, 3 has the middle outward pointing



stroke which 2 does not have. We can see from the first two weights above. The first two weights emphasizes the middle stroke of 3s'. This evidence supports that the model is learning the features of 3 omitting the common features of 3 that shared by 2. The unique features learned by the model allows the program to distinguish future inputs.

5.4 Mixture of Gaussian

Unlike neural network, mixture of Gaussian's method is more like classifying the data.



When we ran the MOG model on the seperate data of 2 and 3s, we are basically learning 20 common writings of 2 and 3's.

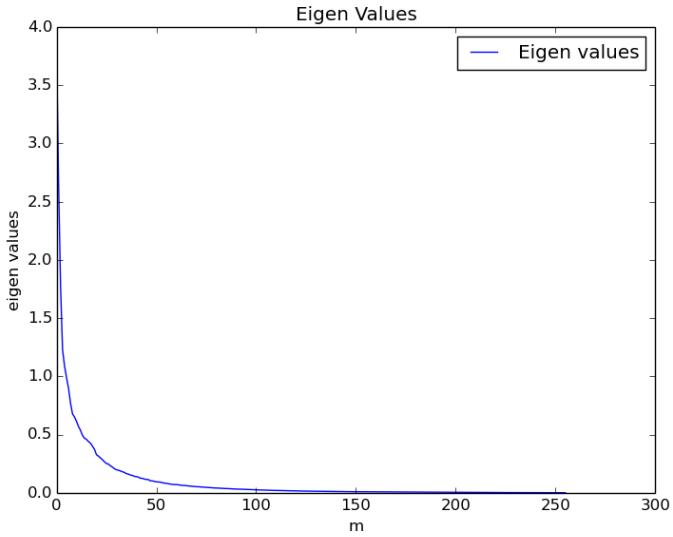
5.5 Mixture of Gaussian vs. Neural Network

The validation rate for neural networks is about 97.0%. The validation rate for neural networks is about 98.0%. Mixture of Gaussian has better performance than neural network may due to the fact that we only have one hidden layer in our neural network. Some non-linear decision boundaries are very hard to be drawn by one hidden layer. On the other hand, it would be easier for MOG to find such a non-linear decision boundary.

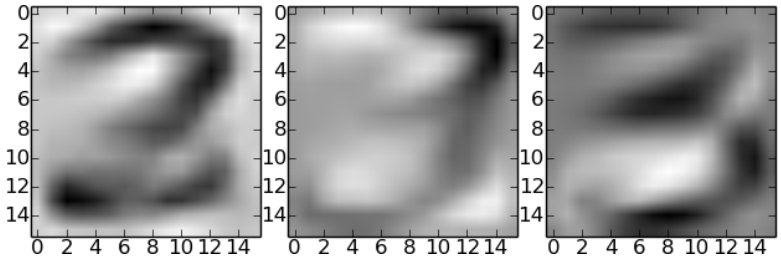
6 Classification using PCA

6.1 PCA Training $m = 256$

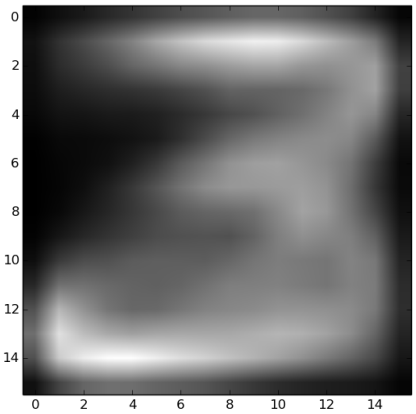
6.1.1 PCA Eigen-values (decreasing order)



6.1.2 PCA first 3 eigen-vectors



6.1.3 PCA mean



6.2 Classification error rates versus number of eigenvectors (2,5,10,20)

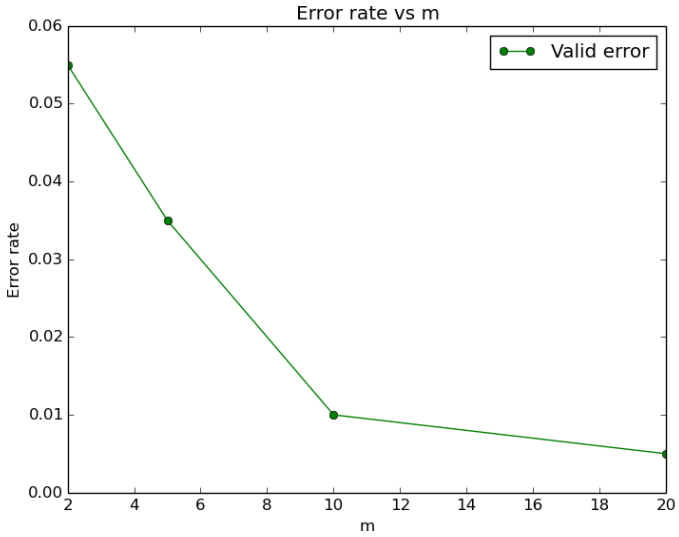


Figure 23: average validation set classification error rates versus number of eigenvectors [2,5,10,20]

From the output, it seems when the number of component is 20 yields the best classification rate(99.5%). From the graph, we observer that the classification error goes down as the number of eigenvectors increases. If you wanted to choose a particular model from your experiments as the best, **I would choose the number of component to be 20.** Although from the trend of the graph, the error seems to be strickly decreasing as the number of components increases. However, I am not sure if the curve has a parabola shape. It might start to increase at some m larger than 20 or it might be already started increasing at some $10 < m < 20$. Therefore, I do not want to choose an m which is greater than 20.

6.3 Classification error rates versus number of eigenvectors (new model: 1-30)

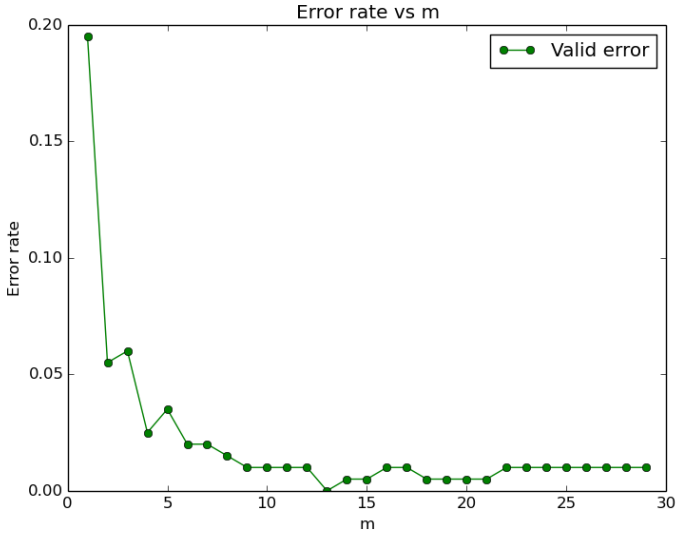


Figure 24: average validation set classification error rates versus number of eigenvectors (1-30)

```
error_rate for m = 1: 0.195
error_rate for m = 2: 0.055
error_rate for m = 3: 0.06
error_rate for m = 4: 0.025
error_rate for m = 5: 0.035
```

```
error_rate for m = 6: 0.02
error_rate for m = 7: 0.02
error_rate for m = 8: 0.015
error_rate for m = 9: 0.01
error_rate for m = 10: 0.01
error_rate for m = 11: 0.01
error_rate for m = 12: 0.01
error_rate for m = 13: 0.0
error_rate for m = 14: 0.005
error_rate for m = 15: 0.005
error_rate for m = 16: 0.01
error_rate for m = 17: 0.01
error_rate for m = 18: 0.005
error_rate for m = 19: 0.005
error_rate for m = 20: 0.005
error_rate for m = 21: 0.005
error_rate for m = 22: 0.01
error_rate for m = 23: 0.01
error_rate for m = 24: 0.01
error_rate for m = 25: 0.01
error_rate for m = 26: 0.01
error_rate for m = 27: 0.01
error_rate for m = 28: 0.01
error_rate for m = 29: 0.01
```

I chose to print out all error rate for number of component ranges from 1 to 30. From the results listed above, I would choose my m to be 13 whose classification error is 0%. It is because the error rate before $m = 13$ is constantly decreasing and also the error rate around $m = 13$ does not have dramatic fluctuation. The error rate around $m = 13$ is about 0.005% - 0.01%.

6.4 Comparing with the mixture of Gaussian

The classification error rate for $m = 13$ is about 1% on my test data which is really close to the 2% error rate yield by the mixture of Gaussian model. Both models have very nice results for this handwriting problem.