Analytic Exercises 2

Pattern Recognition
NTUA 9th Semester

Efthymios Tzinis 03112007

November 17, 2017

Introduction

Most of the analytic exercises demanded convoluted and repeated computations. In order to provide a more delicate report, I used Matlab extensively and put the source code in the respective folders so the reader can also reproduce the reported results. For the exercises demanding Matlab code, we provide the code in the respective **Code Appendix Section**. All the theoretical thinking and assumptions are adequately explained over the next sections. The source code is labeled with the name of each exercise. All Matlab scripts are endowed with useful comments in order to be easily understood.

Exercise 1

1.

The building of the MM-2 equivalent bayessian net was done using the **openfst** for Linux. We simply derive our model by introducing 4 main states of recognizing "1" and "2". But except from that we want an initial zero state for the beginning of the observed sequence as also two extra states for the first symbol observation. I demonstrate the final model, built with **fstdraw** command, in figure 1. For the reproducing you can simply run in the shell:

$$> sh\ run\_openfst.sh$$

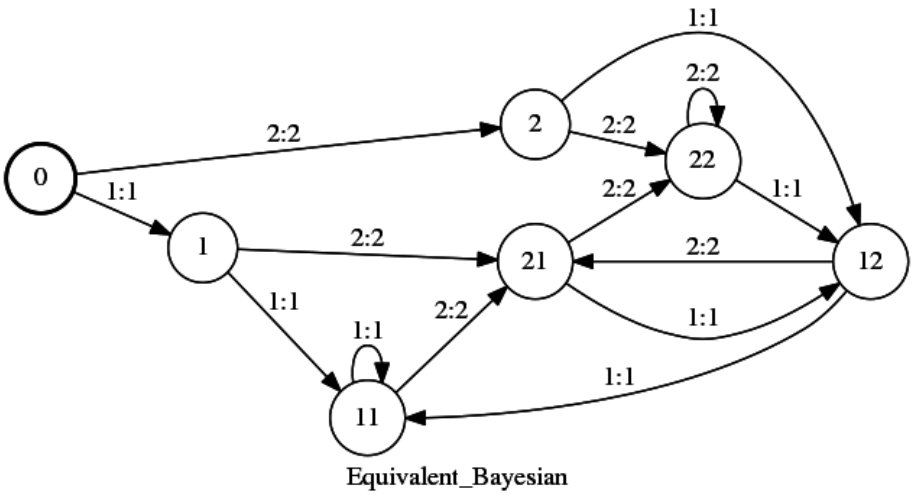


Figure 1

2.

I think the maximizing likelihood step overrides the information of a-priori given information. Thus, from the observed sequence and the ML formula:

$$weight_{ij\to ki} = \frac{\#transitions_{ij\to ki}}{\#transitions_{ij\to *i}},\ i,j,k = 1,2$$

And for the initial symbol states we have respectively:

$$weight_{0\to i} = \frac{\#i}{\#symbols},\ i = 1, 2$$

$$weight_{i\to ki} = \frac{\#transitions_{i\to ki}}{\#transitions_{i\to *i}},\ i, k = 1, 2$$

We could assume that the states 0-1-2 would not be updated but we could also use the a-priori information for building their probability weights. I did not do that because this does not change the model except of its initial condition.

We get the following results, for all the model states:

State vs Input	1	2
0	$\frac{9}{19}$	$\frac{10}{19}$
1	$\frac{5}{9}$	$\frac{4}{9}$
2	$\frac{3}{9}$	$\frac{6}{9}$
11	$\frac{3}{5}$	$\frac{2}{5}$
12	$\frac{1}{3}$	$\frac{2}{3}$
21	$\frac{1}{3}$	$\frac{2}{3}$
22	$\frac{2}{6}$	$\frac{4}{6}$

3.

Assuming that we are already in state "11" we simply compute the probability of getting 10 "1" and an eleventh "2". So the likelihood according to the matrix above:

$$\left(\frac{3}{5}\right)^{10} * \frac{2}{5} = 0.0024186$$

With the same technique we could also assume that we could be in the state in the state "12". Then the likelihood would be:

$$\left(\frac{1}{3}\right)^{10} * \frac{2}{3} = 0.0000113$$

In fact, the exercise should give information about the previous two states of the MM-2 because now seems vaguely expressed. We could actually begin from the states "1" or "2".

4.

For training over the MM-1 we simply get the sub-matrix for the states 0-1-2 of the previous MM-2 model as the latter contains the former. For the computation of the same likelihood as previous we assume that we are already in the state "1" and we cannot make a transition to state 2 until 10 "1"s appear. All in all:

$$\left(\frac{5}{9}\right)^{10} * \frac{4}{9} = 0.0012448$$

By comparison we can see that this likelihood is almost the half of the previous MM-2 result. this is something that diverges from our intuition as this

probability should be the same. Of course, the only thing that caused that different result was the computed matrix of ML updated transition weights. These weight represent probabilities. Of course, if we train our model with little data we can conclude to train our models hastily. Thus, these model cannot represent adequately the data and make predictions.

Exercise 2

All the computations were performed in Matlab:

1.

In order to compute the likelihood we have just to run the **Forward** algorithm or dummy-wise to compute the overall probability. Because the model was tiny we preferred the latter for code simplicity:

$$P(O_0 = H, O_1 = T, O_2 = H) = P(O_0 = H, O_1 = T, O_2 = H|\mathbf{Q})P(\mathbf{Q}) =$$

$$= \sum_{i=1}^{3}\sum_{j=1}^{3}\sum_{k=1}^{3} P(O_0 = H|q_0 = i)P(O_1 = T|q_1 = j)P(O_2 = H|q_2 = k)P(\mathbf{Q})$$

Because all the observations are independent from all the other states except their corresponding states: $O_i \perp\!\!\!\perp q_t|q_i, t \neq i$. Also we can simplify the term corresponding to states distribution:

$$P(\mathbf{Q}) = P(q_2 = k|q_1 = j)P(q_1 = j|q_0 = i)P(q_0 = i)$$

Because all the states are independent in 2-nodes distance $q_i \perp\!\!\!\perp q_{i-2}|q_{i-1}, \forall i$.

All in all, we can acquire all this information by the given tables easily. The overall likelihood found was:

$$P(O_0 = H, O_1 = T, O_2 = H) = 0.1051$$

2.

For getting the most probable state sequence over the observed sequence we have to maximize the probability: $P(\mathbf{Q}|O_0 = H, O_1 = T, O_2 = H)$ using the Viterbi algorithm. I implemented the algorithm for this section with flattening the loop in order to see more clearly the equations:

$$Initialization: \ \delta_1(i) = \pi_i b_i(O_0 = H), i = 1, 2, 3$$

$$Iteration: \ \delta_t(i) = \max_{1\leq i\leq 3} \left(\delta_{t-1}(i)\alpha_{ij}\right) b_j(O_t), 2 \leq t \leq 3$$

$$Finding\ Maximum\ Index: \psi_t(i) = arg \max_{1\leq i\leq 3} \left(\delta_{t-1}(i)\alpha_{ij}\right) b_j(O_t), 2 \leq t \leq 3$$

$$Finalize: \ \psi(final) = \max_{1\leq i\leq 3} \left(\delta_T(i)\right), where\ T = Observed\ Sequence\ Length$$

And we iterate dynamically all over the observed sequence. We have to backtrack our solution over the resulting ψ's in order to get the full state sequence.

The derived matrices demonstrate both iterative variables:

$$\delta_t(i) = \delta(i,t) = \begin{pmatrix} 0.167 & 0.0583 & 0.0204 \\ 0.267 & 0.032 & 0.0154 \\ 0.0667 & 0.0427 & 0.00597 \end{pmatrix}$$

$$\psi_t(i) = \psi(i,t) = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 2 & 2 \\ 0 & 2 & 3 \end{pmatrix}, \psi(final) = 1$$

Thus, the most probable state sequence from backtracking is:

$$q_0 = 1, q_1 = 1, q_2 = 1$$

3.

We run a Viterbi algorithm over the observed states and after that the output hidden states sequence is used for updating the transition weights of the matrix, as also the observation weights for every hidden state. Viterbi algorithm was extensively discussed in the previous section the results are presented below:

$$\delta(i, 1 \leq t \leq 5) = \begin{pmatrix} 0.167 & 0.0583 & 0.0204 & 0.00715 & 0.0025 \\ 0.267 & 0.032 & 0.0154 & 0.00737 & 0.00354 \\ 0.0667 & 0.0427 & 0.00597 & 8.36 \cdot 10^{-4} & 2.95 \cdot 10^{-4} \end{pmatrix}$$

$$\delta(i, 6 \leq t \leq 10) = \begin{pmatrix} 8.75 \cdot 10^{-4} & 3.06 \cdot 10^{-4} & 1.07 \cdot 10^{-4} & 3.75 \cdot 10^{-5} & 1.31 \cdot 10^{-5} \\ 0.0017 & 8.15 \cdot 10^{-4} & 9.78 \cdot 10^{-5} & 1.17 \cdot 10^{-5} & 2.92 \cdot 10^{-6} \\ 1.42 \cdot 10^{-4} & 6.79 \cdot 10^{-5} & 1.3 \cdot 10^{-4} & 7.31 \cdot 10^{-5} & 4.09 \cdot 10^{-5} \end{pmatrix}$$

$$\delta(i, 10 \leq t \leq 15) = \begin{pmatrix} 4.6 \cdot 10^{-6} & 1.61 \cdot 10^{-6} & 6.42 \cdot 10^{-7} & 3.59 \cdot 10^{-7} & 1.26 \cdot 10^{-7} \\ 1.64 \cdot 10^{-6} & 9.16 \cdot 10^{-7} & 5.13 \cdot 10^{-7} & 1.15 \cdot 10^{-6} & 5.52 \cdot 10^{-7} \\ 2.29 \cdot 10^{-5} & 1.28 \cdot 10^{-5} & 7.18 \cdot 10^{-6} & 1.01 \cdot 10^{-6} & 1.41 \cdot 10^{-7} \end{pmatrix}$$

$$\psi_t(i) = \psi(i,t) = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 3 & 3 & 1 \\ 0 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 & 3 & 2 \\ 0 & 2 & 3 & 3 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \end{pmatrix}$$

$$\psi(final) = 2$$

Thus, the most probable hidden state sequence from backtracking is:

$$H_s = 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 2, 2$$

The MLE step of the algorithm is identical to the MLE equation described in the previous exercise. The new weights of the transitions and the weights of observations for every state are updated using the following expressions:

$$a_{ij}^{new} = \frac{\#transitions_{i \to k}}{\#transitions_{i \to *}},\ i, k = 1, 2, 3$$

$$b_i^(weighted)(O_t) = \begin{cases} 1,\ O_t = Observed_s(t), i = H_s(t) \\ 0,\ O_t \neq Observed_s(t), i = H_s(t) \end{cases} \quad b_i^{new}(O_t) = \frac{\sum_{j=1}^{3} b_i^(weighted)(O_t)}{\#i\ state\ occurences}$$

From the above types we can derive our new estimated model for the HMM which happens to be an HMM with only 2 hidden states as the first one does not appear at all in the Hidden state sequence H_s. The trained matrices are represented below:

$$a_{ij}^{new} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0.875 & 0.125 \\ 0 & 0.167 & 0.833 \end{pmatrix}, \; b^{new}(i, O_t) = \begin{pmatrix} 0 & 0.889 & 0 \\ 0 & 0.111 & 1.0 \end{pmatrix}$$

We can practically exclude the first state if we wanted to in order to reduce the complexity.

4.

From the comparative study presented in [1] Baum Welch and Viterbi training are two completely different algorithms. Forward backward is an algorithm that represents the Expectation Maximization Algorithm (EM) and thus it provides the computation of the full conditional likelihood of the given states over the observed data. On the other hand, Viterbi training stands for a Pseudo - EM algorithm and thus, it performs an approximation of Maximum Likelihood in order to save time. In other words, Viterbi's output is a complete chain of most probable hidden states while Baum Welch computes the probability of each hidden state in the chain. Thus in the update step Baum Welch will change the transition weights taking into account the probabilities of all hidden states chains and not only the most probable one like Viterbi. Finally, Baum Welch converges at least at a local minimum instead of Viterbi training that could converge to any point.

Exercise 3

1.

The purpose of building a decision tree is to automatically understand if the user is satisfied or not. In this way we could match the classes ω_1, ω_2 with the satisfaction state of each user. The parameters of the decision tree classifier are all the features provided in the data set. In our cases these features would be: WORD ACCURACY, TASK COMPLETION and TASK DURATION. All the questions that could be expressed would be all the discrete boundaries created by all the possible values of the features provided. For example, the feature WORD ACCURACY in all the data set gets these values: 80, 85 ,90, 95, 100. These values would be all the possible decision boundaries for the question except of 80, 100 that could not produce any division in our data. Then for every sample in the current node we check e.g. $WORD\_ACCURACY(x) < 90$ then this sample belongs to the "No" node or to the "Yes" node otherwise.

2.

According to the theoretical knowledge expressed in [2] for selecting the best decision tree. We simply iterate for all the levels of the tree demanding every

time to acquire the higher decrease in non-purity metric from the derived nodes according to the question. The general expression for measuring non-purity is:

$$\Delta I(t) = I(t) - \frac{N_{tY}}{N} I(t_Y) - \frac{N_{tN}}{N} I(t_N)$$

Where $I(t_Y), I(t_N)$ are the non purities of the derived nodes from the question (response Yes or No): X_Y, X_N respectively. In every level we require this conditional restriction to be satisfied and in this way we choose all the questions in the tree levels. While:

$$I(t) = -\sum_{i=1}^{M} P(\omega_i|t) log_2 P(\omega_i|t)$$

By running a matlab script we can easily compute with the above expressions we repetitively deepen the tree until the entropy impurity gets zero for all the leaves. (In our case this can be achieved easily). We annotate the state "YN" for example as we got a "Y" and after that a "N" in the tree parse. For every deepening, I demonstrate below the computation matrix for the delta impurity for all the questions in every node. If one node has zero impurity (all data belong in only one class) then we do not have to continue divide that node. In every node we select the question with higher delta impurity decrease.

Node vs Entropy	$WA85$	$WA90$	$WA95$	TC	$TD2$	$TD3$	$TD4$
Initial	0.0248	0.0728	0.0183	0.0728	0.0699	0	0.0183
Y	0	0	0.0729	0.171	0	0.322	0.171
N	0	0	0	0.311	0.311	0.311	0
$\mathbf{N}Y$	0	0	0	1.0	0	0	0
$\mathbf{Y}N$	0.252	0	0	0	0.252	0.252	0.252
$\mathbf{N}YN$	0	0	0	0	1.0	1.0	1.0

3.

The tree is displayed for a more user friendly form in figure 2. The most important questions for the tree are actually the ones that go in the top of it. In this resulting tree we can see that the most salient questions are (in descending importance order) $WORD\_ACCURACY \geq 90, TASK\_DURATION \geq 3, TASK\_COMPLETED = YES$. This is somewhat intuitively validated. The most important thing for an ASR system would be to fully understand what we are saying thus we definitely need a highly accurate model for a satisfactory experience with such systems. We also need to be brief in our task and concise in order not to get the user bored, which is something that leads inevitably to discontentment. Lastly, Task completion is something irrelevant for our model because what we thing as the end of a task section would not be similar to what a user thinks that should be. The user could hang up the phone because he is angry with the system but could also hang up because he got what he wanted but did not want to hear "Goodbye". In this way, this feature by itself is rather non discriminatory but as we see from the tree example when we use it as a combination with other features like Task Duration it is a very useful feature (When someone has long TD but has not completed the task then it would be very probable that he was dissatisfied with his experience).

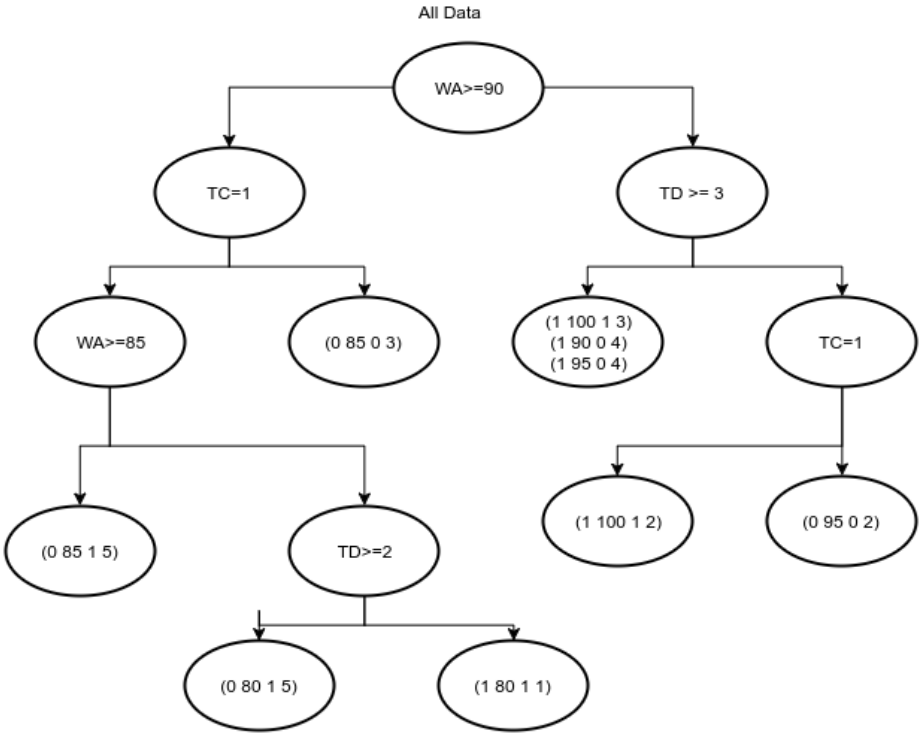


Figure 2: Entropy Impurity Minimizing Tree

Exercise 4

1.

Because the K-NNR is a simple algorithm, based on the computation of the K minimum distances of a point, we need to classify, from all the training data, I thought I could put in use my code from laboratory exercise. The decision on which class the test instance should be classified is based utterly on the K nearest neighbors' classes and performing a majority voting on these values. The expected class for the test element is the majority class over these classes. If a majority cannot be found then the decision is made arbitrarily over the classes that have same number of votes as the majority class. This is why sometimes we could get a classification that is not proper totally, especially for instances too close to the classes boundaries. The results of 3-NNR and 5-NNR are represented graphically in figures 3 and 4 respectively.

We can see that both algorithms conclude to the same classification of the test data (filled points) over the two classes (blue and red). The distances could be easily computed to find the nearest neighbors and thus I did not explain it further. The interesting thing is that some times the classification in a class could be done arbitrarily. For instance, the test point (3,3) in 3NNR would have (3,4) and (3,2) as nearest but it haphazardly chooses (1,3) over (3,1) and ends in blue class (same happens in 5NNR).

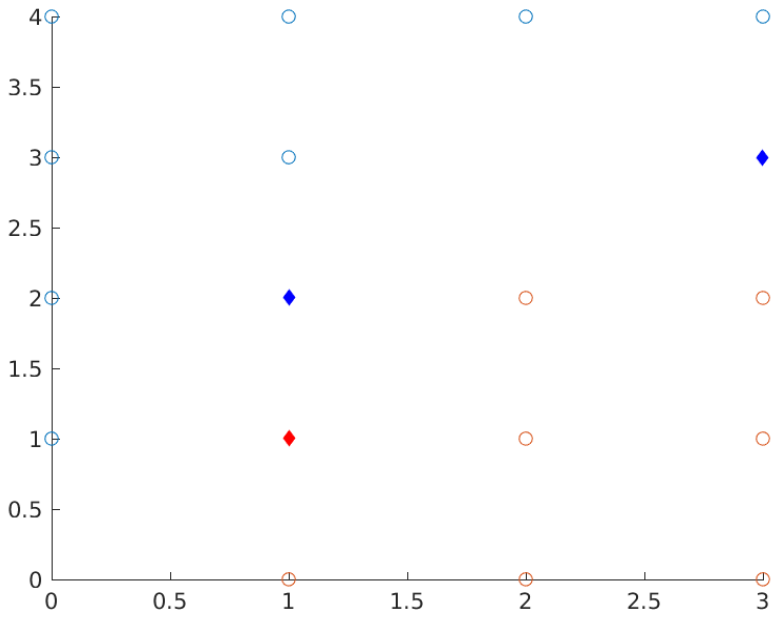


Figure 3: 3NNR Results of Test Classification over Train set

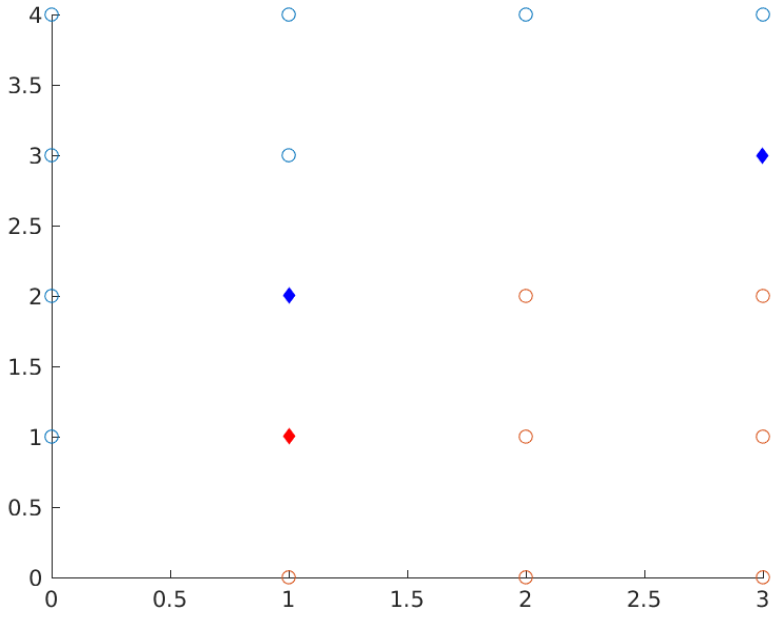


Figure 4: 5NNR Results of Test Classification over Train set

2.

The condensing algorithm over 1NNR is extensively explained in [3] but I think it is useful to highlight its superiority. We use it for space reduction as we can describe the whole train data set with just few points that can practically cover the whole training set. The main idea of the algorithm is: we begin with 2 points $y_1 \in D_1, y_2 \in D_2$ and we test if by using the 1NNR algorithm we can classify all $D_1/\{y_1\}$ in class 1 (basically if y_1 has the minimum distance from all these instances) and simultaneously all $D_2/\{y_2\}$ in class 2. If this condition is true then we stop, otherwise we select the first instance of the train set that was not classified properly and we insert it in the respective condensed set of its class. By doing this repetitively, we will get a training set which is represented by much less points than the initial ones.

The problem with this algorithm is that we choose the initial points arbitrarily and thus we may have different solutions in the problem depending on the initial selected points. From this point of view we can just run the algorithm many times and hope that will find a good solution. Of course this is not guaranteed cause of the NP-Hard Induction of the problem to Cover-Set problem but most of the times we get a very promising solution.

In my implementation I performed 20 iterations and these were more than enough in order to find the best and most compact solution with only two points covering the whole set. In order to keep this solution I just selected the solution with the smaller data point size. These two points were not the same every time I ran these 20 iterations. The results can be found in figure 5. The filled points are the selected representatives of the data set.

Exercise 5

Practically the designated algorithm to be implemented is an EM algorithm for updating the parameters of the assumed normal data distribution over the classes ω_1, ω_2. The equations are perfectly described in [4] CHAPTER 10. UNSUPERVISED LEARNING AND CLUSTERING equations (19-20-21). I implemented these equations in Matlab and iterate over the data set for estimating the parameters of the two Gaussians for the given initial values.

It is useful to see how the parameters are changing over the iterations of the algorithm. (For code see Source folder or Appendix)

$\theta^{(i)}$	$i=0$	$i=1$	$i=2$	$i=3$
$P(\omega_1)^{(i)}$	0.5	0.5	0.5	0.5
$P(\omega_2)^{(i)}$	0.5	0.5	0.5	0.5
$\mu_1^{(i)}$	-5.0	1.67	1.67	1.67
$\mu_2^{(i)}$	10.0	5.33	5.33	5.33
$\sigma_1^{(i)}$	1.0	0.222	0.222	0.222
$\sigma_2^{(i)}$	1.0	0.226	0.222	0.222

After the 2nd iteration we can see that all the parameters of the two gaussians model remain constant and thus the EM algorithm converges.

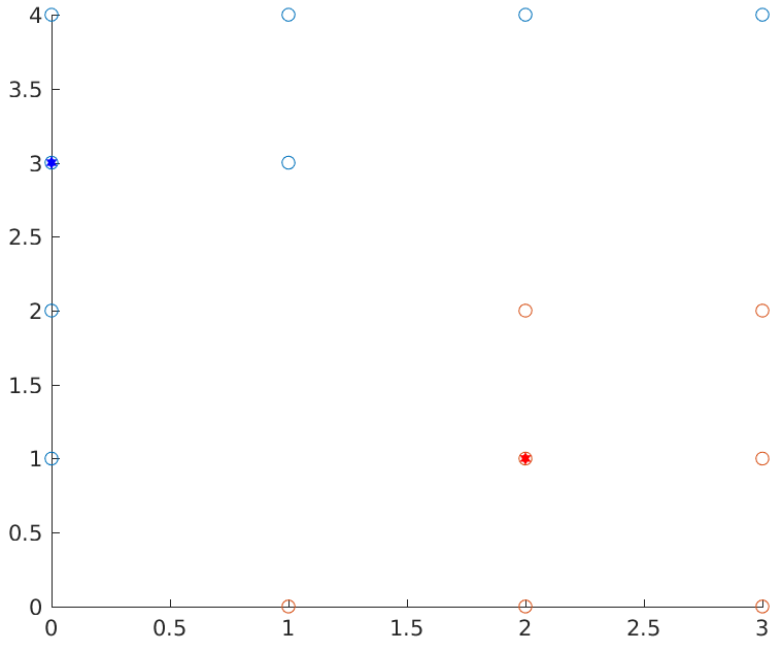


Figure 5: One of the best solutions comprised by only two points and classifying the whole training set.

Exercise 6

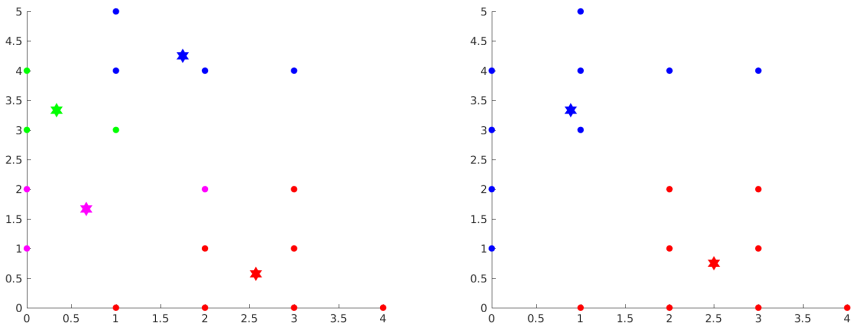
I implemented the ISODATA clustering algorithm which is a refined extent version of K-means with a few special characteristics (For code see Source folder or Appendix):

- Clusters that have too few members are discarded.
- Clusters that have too many members are split into two new cluster groups.
- Clusters that are too large (too disperse) are split into two new cluster groups.
- If two cluster centers are too close together they are merged.

In figures 6 and 7 we can see the results of the clustering for 2 distances Euclidean and Mahalanobis. Both algorithms were tested for the following configurations:

- $K_0 = 4,\ N_{Threshold} = 2,\ Maximum\,Variance = 2,\ Distance\,Threshold = 1.5,\ Iterations\ Ceil = 20$
- $K_0 = 4,\ N_{Threshold} = 1,\ Maximum\,Variance = 1,\ Distance\,Threshold = 2,\ Iterations\ Ceil = 10$

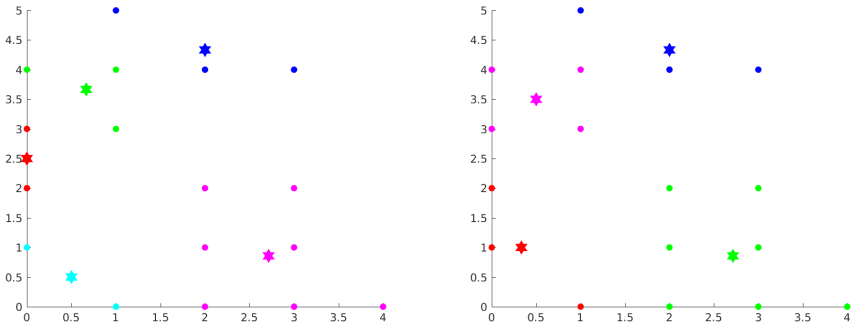
We can see from the figures that the Euclidean distance algorithm is far more dependent on the initial parameters given in the algorithm. On the other hand, the results from the Mahalanobis distance seem more stable even if we simply change the parameters as I shown before. This could be explained, because Mahalanobis distance if a far more accurate metric for measuring distances from centroids rather than the euclidean. All in all, Mahalanobis distance gives us a very similar result for both algorithm's configurations while the euclidean configured algorithm demonstrate a very alienate behavior (In the second configuration it gives us only two clusters).



(a) K 0 = 4, N Threshold = 2, Maximum Variance = 2, Distance Threshold = 1.5 , Iterations Ceil = 20

(b) K 0 = 4, N Threshold = 1, Maximum Variance = 1, Distance Threshold = 2 , Iterations Ceil = 10

Figure 6: ISODATA with Euclidean distance



(a) K 0 = 4, N Threshold = 2, Maximum Variance = 2, Distance Threshold = 1.5 , Iterations Ceil = 20

(b) K 0 = 4, N Threshold = 1, Maximum Variance = 1, Distance Threshold = 2 , Iterations Ceil = 10

Figure 7: ISODATA with Mahalanobis distance

Exercise 7

If the variables $x_1, ..., x_l$ are statistically independent then we can write the equation:

$$P(x|\omega_i) = \prod_{k=1}^{l} P(x_k|\omega_i) \tag{1}$$

In this way the divergence over two different classes ω_i, ω_j would be from the exercise's hypothesis:

$$d_{ij}(x_1, ..., x_l) = \int_{-\infty}^{+\infty} [P(\mathbf{x}|\omega_i) - P(\mathbf{x}|\omega_j)] ln\left(\frac{P(\mathbf{x}|\omega_i)}{P(\mathbf{x}|\omega_j)}\right) d\mathbf{x} =$$

$$\overset{(1)}{=} \int_{-\infty}^{+\infty} [\prod_{k=1}^{l} P(x_k|\omega_i) - \prod_{q=1}^{l} P(x_q|\omega_j)] ln\left(\frac{P(\mathbf{x}|\omega_i)}{P(\mathbf{x}|\omega_j)}\right) d\mathbf{x} \tag{2}$$

For the logarithmic part we can do the following:

$$ln\left(\frac{P(\mathbf{x}|\omega_i)}{P(\mathbf{x}|\omega_j)}\right) = ln\left(P(\mathbf{x}|\omega_i)\right) - ln\left(P(\mathbf{x}|\omega_j)\right) =$$

$$\overset{(1)}{=} ln\left(\prod_{k=1}^{l} P(x_k|\omega_i)\right) - ln\left(\prod_{q=1}^{l} P(x_q|\omega_j)\right) = \sum_{r=1}^{l} ln\left(\frac{P(x_r|\omega_i)}{P(x_r|\omega_j)}\right) \tag{3}$$

Then by substituting equation 3 in equation 2 we have:

$$d_{ij}(x_1, ..., x_l) \overset{3,2}{=} \int_{-\infty}^{+\infty} [\prod_{k=1}^{l} P(x_k|\omega_i) - \prod_{q=1}^{l} P(x_q|\omega_j)] \sum_{r=1}^{l} ln\left(\frac{P(x_r|\omega_i)}{P(x_r|\omega_j)}\right) d\mathbf{x}$$

But we can change the order of computing the sum and the integral:

$$d_{ij}(x_1, ..., x_l) = \sum_{r=1}^{l} \int_{-\infty}^{+\infty} [\prod_{k=1}^{l} P(x_k|\omega_i) - \prod_{q=1}^{l} P(x_q|\omega_j)] ln\left(\frac{P(x_r|\omega_i)}{P(x_r|\omega_j)}\right) d\mathbf{x} =$$

$$= \sum_{r=1}^{l} \int_{-\infty}^{+\infty} [\prod_{k=1}^{l} P(x_k|\omega_i) ln\left(\frac{P(x_r|\omega_i)}{P(x_r|\omega_j)}\right) - \prod_{q=1}^{l} P(x_q|\omega_j) ln\left(\frac{P(x_r|\omega_i)}{P(x_r|\omega_j)}\right)] d\mathbf{x} \tag{4}$$

By taking one of the inner terms of the integral:

$$\int_{-\infty}^{+\infty} \prod_{k=1}^{l} P(x_k|\omega_i) ln\left(\frac{P(x_r|\omega_i)}{P(x_r|\omega_j)}\right) d\mathbf{x} = \int_{-\infty}^{+\infty} ... \int_{-\infty}^{+\infty} \prod_{k=1}^{l} P(x_k|\omega_i) ln\left(\frac{P(x_r|\omega_i)}{P(x_r|\omega_j)}\right) dx_1 ... dx_l$$

By extending the product and changing the order of integrals computations we have:

$$\int_{-\infty}^{+\infty} P(x_r|\omega_i) ln\left(\frac{P(x_r|\omega_i)}{P(x_r|\omega_j)}\right) \int_{-\infty}^{+\infty} ... \int_{-\infty}^{+\infty} \prod_{k=1, k\neq r}^{l} P(x_k|\omega_i) dx_1 ... dx_{r-1} dx_{r+1} ... dx_l dx_r \tag{5}$$

But all the variables are independent thus when we integrate over a variable all the others are considered constants and they can commute with the integration. Consequently, the inner integrals can be viewed as:

$$\int_{-\infty}^{+\infty} ... \int_{-\infty}^{+\infty} \prod_{k=1,k\neq r}^{l} P(x_k|\omega_i)dx_1...dx_{r-1}dx_{r+1}...dx_l = \prod_{k=1,k\neq r}^{l} \int_{-\infty}^{+\infty} P(x_k|\omega_i)dx_k \tag{6}$$

But each integral in the product represents an integral of the whole dependent distribution and this will be one.

$$\int_{-\infty}^{+\infty} P(x_k|\omega_i)dx_k = 1,\ k = 1, ..., (r-1), (r+1), ...l \tag{7}$$

$$(6) \Rightarrow \int_{-\infty}^{+\infty} ... \int_{-\infty}^{+\infty} \prod_{k=1,k\neq r}^{l} P(x_k|\omega_i)dx_1...dx_{r-1}dx_{r+1}...dx_l \overset{(7)}{=} 1 \tag{8}$$

By substituting to equation (5) we will have:

$$\int_{-\infty}^{+\infty} \prod_{k=1}^{l} P(x_k|\omega_i) ln\left(\frac{P(x_r|\omega_i)}{P(x_r|\omega_j)}\right) d\mathbf{x} \overset{(8)}{=} \int_{-\infty}^{+\infty} P(x_r|\omega_i) ln\left(\frac{P(x_r|\omega_i)}{P(x_r|\omega_j)}\right) dx_r \tag{9}$$

Now we just have to substitute into (4) as a final step:

$$d_{ij}(x_1, ..., x_l) = \sum_{r=1}^{l} \int_{-\infty}^{+\infty} [P(x_k|\omega_i) ln\left(\frac{P(x_r|\omega_i)}{P(x_r|\omega_j)}\right) - P(x_q|\omega_j) ln\left(\frac{P(x_r|\omega_i)}{P(x_r|\omega_j)}\right)]dx_r$$

Now it is easy to see that we have proven the statement:

$$d_{ij}(x_1, ..., x_l) = \sum_{r=1}^{l} \int_{-\infty}^{+\infty} [P(x_k|\omega_i) - P(x_q|\omega_j)] ln\left(\frac{P(x_r|\omega_i)}{P(x_r|\omega_j)}\right)]dx_r = \sum_{r=1}^{l} d_{ij}(x_r)$$

References

[1] Rodrguez, L. J., & Torres, I. (2003). Comparative Study of the Baum-Welch and Viterbi Training Algorithms Applied to Read and Spontaneous Speech Recognition (pp. 847857). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-44871-6_98

[2] Theodoridis, S., & Koutroumbas, K. (2009). Pattern recognition. Academic Press.

[3] Amal, M.-A. (2011). Survey of Nearest Neighbor Condensing Techniques. IJACSA) International Journal of Advanced Computer Science and Applications, 2(11). Retrieved from www.ijacsa.thesai.org

[4] R. O. Duda, P. E. Hart, and D. G. Stork, Pattern Classification, Wiley Interscience, 2nd edition, 2000.

Code Appendix

I present here the code only for the exercises which demanded it. All the source can be found in the respective folder provided alongside the report.

Exercise 5

```
close all;
clear all;

Data = [1; 2; 2; 5; 5; 6];

%% Initial Values (iteration 0)
mean_vals(1,:) = [-5 10];
variance(1,:) = [1 1];
prior(1,:) = [0.5 0.5];

%% Algorithm for a given number of iterations
iterations = 4;

for i=2:iterations
    % Compute probabilities for all the data belonging to each class
    for j = 1:size(mean_vals,2)
        for k = 1:length(Data)
            prob(j,k) = prior(i-1,j) * sqrt(variance(i-1,j)) * exp(-0.5 * ((Dat
        end
    end

    for j = 1:size(mean_vals,2)
        for k = 1:length(Data)
            probabilities(j,k) = prob(j,k) / sum(prob(:,k));
        end
    end

    % Update the distributions parameters according to the previous
    % porbabilities
%     [mean_vals(i,:), variance(i,:), prior(i,:)] = evaluate_parameters(Data, p
    for j = 1:size(probabilities,1)
        prior(i,j) = mean(probabilities(j,:));
        normalization_term = sum(probabilities(j,:));
        mean_vals(i,j) = probabilities(j,:) * Data / normalization_term;
        variance(i,j) = probabilities(j,:) * (Data - mean_vals((i),j)).^2 / nor
    end
end

d1 = digits(3);
latex(vpa(sym(['prior'; mean_vals'; variance']))
digits(d1);
```

Exercise 6

```
close all;
clear all;

Data = [0 1; 0 2; 0 3; 0 4; 1 3; 1 4; 1 5; 2 4;
3 4; 1 0; 2 0; 3 0; 2 1; 3 1; 2 2; 3 2; 4 0];

M_vals = [1 3; 3 1];

V_vals(1,:,:) = [1 0; 0 1];
V_vals(2,:,:) = [1 0; 0 1];

% Algorithm parameters
K_0 = 4;
N_Threshold = 1;
Maximum_Variance = 1;
Distance_Threshold = 2;
Iterations_Ceil = 10;

selected_distance = 'mahalanobis';
%selected_distance = 'euclidean';


for iter = 1:Iterations_Ceil

    %% Selecting the appropriate distance metric for the algorithm
    distances = pdist2(Data, M_vals, selected_distance);
    [~, min_indexes] = sort(distances , 2);

    % Create the clusters
    for i = 1:size(M_vals, 1)
        clusters(i) = {find(min_indexes(:,1) == i)};
    end

    % Discard clusters with too few instances
    cluster_sizes = [];
    i = 1;
    while i <= size(M_vals,1)
        cluster_size = length(clusters{i});

        % If one cluster has less members than the minimum
        if cluster_size < N_Threshold
            M_vals(i,:) = [];
            V_vals(i,:,:) = [];

            distances = pdist2(Data, M_vals,selected_distance);
            [~, min_indexes] = sort(distances , 2);

            % Re-Assign to clusters
```

```
            for k = 1:size(M_vals, 1)
                clusters(k) = {find(min_indexes(:,1) == k)};
            end

        else
            i = i + 1;
        end
    end

    % Update M_vals and V_vals
    N = length(clusters);
    M_vals = zeros(N, 2);
    V_vals = zeros(N, 2, 2);
    for i = 1:N
        M_vals(i,:) = mean(Data(clusters{i}, :));
        V_vals(i,:,:) = cov(Data(clusters{i}, :));
    end

    % Decide if we have to split or merge a cluster
    if length(clusters) <= K_0 /2

        % Split a cluster into 2
        i = 1;
        while i <= size(M_vals,1)
            [covariance_max, covariance_max_index] =
            max(diag(squeeze(V_vals(i,:,:))));
            cluster_size = length(clusters{i});
            % If one cluster's greatest covariance is greater than max
            if covariance_max > Maximum_Variance &&
            cluster_size > 2 * N_Threshold
                current_cluster = Data(clusters{i},
                covariance_max_index);
                cluster1 = find(current_cluster >=
                M_vals(i,covariance_max_index));
                cluster2 = find(current_cluster <
                M_vals(i,covariance_max_index));

                clusters(i) = {cluster1};
                clusters(size(M_vals,1)+1) = {cluster2};

                % Update M_vals and V_vals
                N = length(clusters);
                M_vals = zeros(N, 2);
                V_vals = zeros(N, 2, 2);
                for k = 1:N
                    M_vals(k,:) = mean(Data(clusters{k}, :));
                    V_vals(k,:,:) = cov(Data(clusters{k}, :));
                end
            end
            i = i + 1;
```

```
        end

    elseif length(clusters) > 2 * K_0

        % Merge two clusters
        N = length(clusters);

        %First Compute the bhattacharrya distances
        bhat_distances = Compute_bhattacharyya_distance
        (clusters, M_vals, V_vals);

        i = 1;
        while i <= size(bhat_distances,1)
            j = i+1;
            while j <= size(bhat_distances,2)
                if bhat_distances(i, j) < Distance_Threshold
                    clusters(i) = {[clusters{i} ; clusters{j}]};
                    clusters(j) = [];

                    % Update M_vals and V_vals
                    N = length(clusters);
                    M_vals = zeros(N, 2);
                    V_vals = zeros(N, 2, 2);
                    for k = 1:N
                        M_vals(k,:) = mean(Data(clusters{k}, :));
                        V_vals(k,:,:) = cov(Data(clusters{k}, :));
                    end

                    % Recompute the Bhattacharya distances:
                    bhat_distances = Compute_bhattacharyya_distance
                    (clusters, M_vals, V_vals);

                    i = 1;
                    j = i+1;
                else
                    j = j + 1;
                end
            end
            i = i + 1;
        end

    end
end

figure;
colors = {'b','r','g','m','c','y','k'};

for i = 1:length(clusters)
    indexes = clusters{i};
```

```
    scatter(Data(indexes,1), Data(indexes,2),colors{i},'filled');
    hold on;
end

for i=1:size(M_vals,1)
    scatter(M_vals(i,1), M_vals(i,2), 180, 'filled',colors{i}, 'h');
    hold on;
end

saveas(gcf,['ISODATA' selected_distance '.png'])

function [ bhat_distances ] =
Compute_bhattacharyya_distance( clusters, means, variances )
    N = length(clusters);
    bhat_distances = zeros(N, N);
    for i = 1:N
        for j = (i+1):N
            var_i = squeeze(variances(i,:,:));
            var_j = squeeze(variances(j,:,:));
            bhattach1 = 1/4 * (means(i,:) - means(j,:));
            bhattach2 = inv((var_i + var_j)/2);
            bhattach3 = (means(i,:) - means(j,:))';

            bhattach4_1 = det((var_i + var_j)/2);
            bhattach4_2 = sqrt(det(var_i) * det(var_j));
            bhattach4 = log(bhattach4_1 / bhattach4_2);
            bhat_distances(i, j) = bhattach1 * bhattach2 *
            bhattach3 + bhattach4;
        end
    end
end
```