Assignment 2 - Report

Pietro Alovisi

January 6, 2019

For these first questions I assumed shaded nodes are observed.

Question 2.2.1

Yes.

Question 2.2.2

None.

Question 2.2.3

No.

Question 2.2.4

Yes.

Question 2.2.5

Yes.

Question 2.2.6

Minimum set : $A = \{\mu_{r,c}, \mu_{r,c-1}, \mu_{r,c+1}, \mu_{r-1,c}, \mu_{r+1,c}, \lambda\}$

Question 2.2.7

No.

Question 2.2.8

No.

Question 2.2.9

Minimum set :$B = \{Z_m^n, C^n, n \in [N], m \in [M]\}$

Question 2.3.10

To compute $p(\beta|T, \Theta)$ we can take inspiration from the lecture notes, by calling $ x_r $ the root node of the tree:

$$p(\beta|T, \Theta) = \sum_i^K s(r, i)p(x_r = i) = \sum_i^K p(x_{\downarrow r \cap O}|x_r = i)p(x_r = i)$$

Where the function $s(u, i) = p(x_{\downarrow u \cap O}|x_u = i)$ can be computed easily and efficiently. By calling $x_{1:C}$ the set children of node x_u we can expand the above expression in:

$$s(u, i) = \sum_{x_{1:C}} p(x_{\downarrow u \cap O}|x_{1:C})p(x_{1:C}|x_u = i)$$

We can use the fork structure of the tree to factorize the formula in each of the children of x_u, which are indexed by x_c:

$$s(u, i) = \sum_k^K \left(\prod_c p(x_{\downarrow c \cap O}|x_c = k) \prod_c p(x_c = k|x_u = i) \right)$$

We can then group each term that refer to each child, effectively swapping the product with the sum, and so we get and expression that works:

$$s(u, i) = \prod_c \sum_k^K (p(x_{\downarrow c \cap O}|x_c = k)p(x_c = k|x_u = i)) = \prod_c \sum_k^K s(x_c, k)p(x_c = k|x_u = i)$$

Where the term $p(x_c = k|x_u = i)$ is clearly the CPD. For the leaves the computation of $s(u, i)$ is just:

$$s(l, i) = \begin{cases} 1 & x_l = i \\ 0 & x_l \neq i \end{cases}$$

Now we can give a pseudocode of the algorithm:

```
Given tree

while node = postorder_traverasal(tree)
    if node is leaf:
        node.s = compute_leaf_s(node)
    else:
        node.s = compute_s(children of node)

output = weighted_sum(root.s,prob_root)
```

Given its simplicity I also list the code here.

```python
def compute_s(node):
    # if this is a leaf
    K = len(node.cat[0])
    s = [0] * K

    if len(node.descendants) == 0:
        s[node.sample] = 1
        # stop recursion
        node.s_i = s
        return
    # recurse to compute s in subtrees
    for child in node.descendants:
        compute_s(child)

    # once s is computed then compute it
    # for the current node.

    for i in range(K):
        children_sub = []
        for child in node.descendants:
            s_uv_i = compute_factor_s(child, i)
            children_sub.append(s_uv_i)
        # update s distribution

        s[i] = np.exp(np.sum(np.log(children_sub)))  # np.prod(children_sub)

    # save in the tree structure
    node.s_i = s
    return

def compute_factor_s(child, father_value):
    K = len(child.cat[0])
    results = [0] * K

    for i in range(K):
        results[i] = child.cat[father_value][i] * child.s_i[i]

    return np.sum(results)

def compute_leaves_probability(tree):
    root = tree.root
    total_sum = []
    for i in range(len(root.cat[0])):
        # for numerical reasons put each addend in a list
        # and use np.sum
        total_sum.append(root.s_i[i] * root.cat[0][i])
    return np.sum(total_sum)
```

Complexity

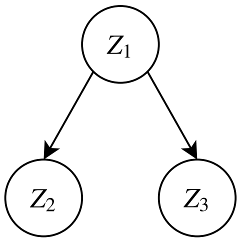
To perform this computation we used a classical traversal of the tree, which takes $O(N)$ and then per each node the perform a computation that takes $O(K^2)$, which for $K \ll N$ can be regarded as $O(1)$.In general the procedure has the complexity of $O(N \cdot K^2)$.

I used a recursive procedure for the traversal, a more efficient one would work the same way, but the traversal would occur in a single while loop.

Question 2.3.11

To test the correctness of the algorithm first I will apply it to some simple trees: first with one having some deterministic CPD, and then one which I can compute the probability by hand. An example is shown in figure 1, where the tree has deterministic CPD and I can test the correctness immediately.

In the list below I listed the results on the samples given in the assignment repo. The list is structured so that I report the likelihood of each sample set for each tree.



PDF, Z_1	$p(Z_1 = 0)$	$p(Z_1 = 1)$
	0.1	0.9

$CPD, i = 1, 2$	$p(Z_i = 0 \mid Z_1)$	$p(Z_i = 1 \mid Z_1)$
$Z_1 = 0$	0	1
$Z_1 = 1$	1	0

$Observation$	$p(Observation)$
$Z_2 = 0, Z_3 = 0$	0.9
$Z_2 = 1, Z_3 = 1$	0.1
$Z_2 = 0, Z_3 = 1$	0

Figure 1: Test of the algorithm on a simple tree

```
#### TREE : tree_k_3_md_3_mb_7_alpha_[0.1 0.5 0.5]
    sample_1 : 0.015575367637587234
    sample_2 : 0.08802394661204103
    sample_3 : 0.1325663427163154
#### TREE : tree_k_2_md_5_mb_5_alpha_[0.5 0.5]
    sample_1 : 9.548083102859963e-09
    sample_2 : 1.7019600222273034e-08
    sample_3 : 1.9887772349364952e-08
#### TREE : tree_k_3_md_10_mb_2_alpha_[0.1 0.1 0.5]
    sample_1 : 0.6263154216756738
    sample_2 : 0.039521551540131525
    sample_3 : 0.6263154216756738
#### TREE : tree_k_3_md_3_mb_5_alpha_[0.1 0.1 0.5]
    sample_1 : 0.9801324240627479
    sample_2 : 0.9801324240627479
    sample_3 : 0.9801324240627479
#### TREE : tree_k_2_md_10_mb_2_alpha_[0.1 0.5]
    sample_1 : 0.8877139127276715
    sample_2 : 0.8877139127276715
    sample_3 : 0.8877139127276715
#### TREE : tree_k_2_md_3_mb_3_alpha_[0.1 0.1]
    sample_1 : 0.3810083320125482
    sample_2 : 0.3810083320125482
    sample_3 : 0.6150083011873109
#### TREE : tree_k_5_md_3_mb_5_alpha_[1.  0.1 0.1 0.1 0.1]
    sample_1 : 0.9691786463025794
    sample_2 : 0.9691786463025794
    sample_3 : 0.9691786463025794
#### TREE : tree_k_2_md_5_mb_7_alpha_[0.1 0.1]
    sample_1 : 1.6817751988519815e-07
    sample_2 : 1.0842133852299218e-07
    sample_3 : 1.9962991536944692e-06
#### TREE : tree_k_5_md_10_mb_3_alpha_[0.1 0.1 1.  0.1 0.1]
    sample_1 : 0.47097862357148107
    sample_2 : 0.47097862357148107
    sample_3 : 0.512871879450965
#### TREE : tree_k_2_md_3_mb_5_alpha_[0.1 0.1]
    sample_1 : 0.0031340499162434772
    sample_2 : 0.41227206024808
    sample_3 : 0.41227206024808
#### TREE : tree_k_5_md_5_mb_7_alpha_[0.1 0.1 0.1 0.5 0.1]
    sample_1 : 3.551741843757672e-33
    sample_2 : 3.518902907183992e-29
    sample_3 : 1.1132309079060315e-35
#### TREE : tree_k_5_md_5_mb_3_alpha_[0.1 0.1 0.1 0.1 0.1]
    sample_1 : 0.01324142322614498
    sample_2 : 0.00020422494191284083
    sample_3 : 0.001638604224482264
#### TREE : tree_k_3_md_10_mb_4_alpha_[0.1 0.1 0.1]
    sample_1 : 1.0618251222255887e-127
    sample_2 : 1.0498286341397606e-114
    sample_3 : 3.921691876030344e-124
#### TREE : tree_k_2_md_10_mb_4_alpha_[0.1 0.1]
    sample_1 : 8.340290810103572e-29
    sample_2 : 3.169821630344867e-30
    sample_3 : 1.9958775342294182e-26
#### TREE : tree_k_2_md_5_mb_3_alpha_[0.1 0.1]
    sample_1 : 0.0011832038376663092
    sample_2 : 0.012738265223240591
    sample_3 : 0.023390644794753102
#### TREE : tree_k_3_md_5_mb_3_alpha_[1.  1.  0.1]
    sample_1 : 0.009426548851486758
    sample_2 : 0.0011604736795091796
    sample_3 : 0.0034806316622987754
#### TREE : tree_k_3_md_5_mb_5_alpha_[0.1 0.5 0.1]
    sample_1 : 0.09704906598867392
    sample_2 : 0.015478489631606493
    sample_3 : 0.5517569755108211
#### TREE : tree_k_2_md_3_mb_7_alpha_[0.1 0.5]
    sample_1 : 0.0013813279441867985
    sample_2 : 0.024451219849260204
    sample_3 : 0.0022937953990344595
#### TREE : tree_k_3_md_5_mb_7_alpha_[0.5 0.1 0.1]
    sample_1 : 4.085008345315396e-43
    sample_2 : 8.961838609320557e-46
    sample_3 : 5.440333532434163e-50
#### TREE : tree_k_5_md_3_mb_3_alpha_[0.1 0.5 0.1 0.1 0.1]
    sample_1 : 0.22758221442785015
    sample_2 : 0.22758221442785015
    sample_3 : 0.17414907764218807
#### TREE : tree_k_3_md_3_mb_3_alpha_[0.1 1.  1. ]
    sample_1 : 0.2771973872404016
    sample_2 : 0.2771973872404016
    sample_3 : 0.11153894154904093
#### TREE : tree_k_5_md_10_mb_2_alpha_[1.  0.5 0.1 0.5 0.5]
    sample_1 : 0.04059159016234465
    sample_2 : 0.03728845792384831
    sample_3 : 0.15311568273629658
#### TREE : tree_k_2_md_10_mb_3_alpha_[1.  0.1]
    sample_1 : 0.15039488984191376
    sample_2 : 0.017782275009702007
    sample_3 : 0.1630417128776933
#### TREE : tree_k_3_md_10_mb_3_alpha_[0.5 0.1 0.5]
    sample_1 : 0.06372451094688635
    sample_2 : 0.5388568329045139
    sample_3 : 0.3093635150318118
#### TREE : tree_k_5_md_3_mb_7_alpha_[0.1 0.1 0.5 1.  0.1]
    sample_1 : 7.29290846900528e-05
    sample_2 : 8.223041997975555e-05
    sample_3 : 4.591814922197935e-07
#### TREE : tree_k_5_md_10_mb_4_alpha_[0.1 0.1 0.1 0.1 0.1]
    sample_1 : 2.44568054618614e-299
    sample_2 : 6.063002427025665e-289
    sample_3 : 8.745578576577856e-304
#### TREE : tree_k_5_md_5_mb_5_alpha_[0.5 0.5 0.1 1.  1. ]
    sample_1 : 4.482685715195033e-38
    sample_2 : 2.0737157063027863e-33
    sample_3 : 1.9518267273562535e-32
```

Question 2.5.15

The posterior on the nodes of the tree given the leaves does not factor in the product of the posterior of each node, instead it still follows the tree factorization. Each term in the factoriziation not only has as given the parent value, but also the leaves value. As a demonstration we perform the first factorization at the root:

$$p(X|x_O) = \left(\prod_{s \text{ subtree of } x_r} p(s|x_r, x_o) \right) p(x_r|x_o)$$

Then this factorization is reiterate in each of the subtree, until the leaves. The core idea is to perform ancestral sampling, so we will start from the root, and sample from its posterior, then move in each subtree and sample their root by first computing their posterior given also the parent value. From the question of before, we already know how t compute $s(u, i)$:

$$s(u, i) = p(x_{\downarrow u \cap O}|x_u = i) = \sum_{x_{\downarrow u \setminus O}} p(x_{\downarrow u \cap O}|x_u = i)$$

The posterior in each node is just:

$$p(x_u = j|O, x_{pa(u)}) \propto p(x_{\downarrow u \cap O}|x_u = i)p(x_u = i|x_{pa(u)}) = s(u, i)p(x_u = i|x_{pa(u)})$$

And since the root has no parent, then the second term is just the probability distribution in the node. We can think of it in a recursive/dynamic programming perspective, by sampling the root we then create N independent subtree, which than can be sampled with the same procedure.

This can be easily performed by a top down traversal of the tree and use the already computed statistics in each node.

Pseudocode and complexity

The procedure first computes the s distribution for each node, computed bottom-up. Then it is performed a top-down traversal where in each node, we have to compute the posterior, but it still takes constant time in each node.

```
compute_s(tree)

while node = depth_first(tree):
    node.posterior = compute_posterior(node)
    node.sample = node.sample_from_posterior(parent)
```

By following the previous argumentations we can see that the algorithm must perform 2 tree traversal and compute constant time quantities in each node. And so the total time is still linear in the number of tree nodes.

$$T(n) = O(n)$$

Question 2.5.16

The algorithm below implements the algorithm proposed above, the code is quite simple and readable and should not need too many comments. It uses the functions developed in the previous task.

```python
def sample_tree(tree):
    compute_s(tree.root)
    sample_node(tree.root)


def sample_node(node):
    # leaves
    if len(node.descendants) == 0:
        return

    compute_node_posterior(node)
    node.sample = np.random.choice(len(node.posterior), p=node.posterior)

    for child in node.descendants:
        sample_node(child)

def compute_node_posterior(node):
    # if node is the root
    if node.ancestor is None:
        node.posterior = np.multiply(node.s_i, node.cat[0])
        node.posterior = node.posterior / np.sum(node.posterior)
    else:
        # if it is a inner node
        node.posterior = np.multiply(node.cat[node.ancestor.sample], node.s_i)
        node.posterior = node.posterior / np.sum(node.posterior)
```

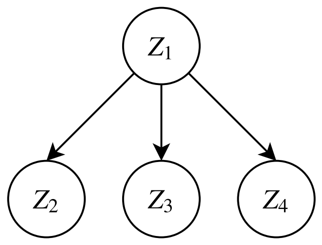
Question 2.5.17

Again I tried the correctness of the algorithm by first looking at the posterior distribution on a small tree (figure 2), and then verifying that the result makes sense. I used the tree pictured below, and we can see how the probability mass of the posterior moves on the most likely value of the root given the leaves.

Here is the result of the algorithm on the given tree, we report the sample in each node, and the posterior computed in each node given the leaves and the parent. The leaves are without posterior. Of course the posterior can change based on the sample of the parent, here is just one instance.

The nodes are expressed as `name:parent:sample|posterior` .

```
1: sample : 0  | post : [0.66763201 0.33236799]
2:1: sample : 1  | post : [0.44951963 0.55048037]
3:1: sample : 1  | post : [0.04826137 0.95173863]
4:1: sample : 1  | post : [0.47690732 0.52309268]
5:2: sample : 0
6:2: sample : 1
7:3: sample : 1  | post : [0.51556917 0.48443083]
8:3: sample : 0  | post : [0.97033051 0.02966949]
9:3: sample : 0  | post : [0.94403478 0.05596522]
```



PDF, Z_1	$p(Z_1 = 0)$	$p(Z_1 = 1)$
	0.5	0.5

$CPD, i = 2, 3, 4$	$p(Z_i = 0 \mid Z_1)$	$p(Z_i = 1 \mid Z_1)$
$Z_1 = 0$	0.9	0.1
$Z_1 = 1$	0.1	0.9

Observation	Posterior of Z_1
$Z_2 = 0, Z_3 = 0, Z_4 = 0$	[0.99863 0.00137]
$Z_2 = 1, Z_3 = 0, Z_4 = 0$	[0.9 0.1]
$Z_2 = 1, Z_3 = 1, Z_4 = 0$	[0.1 0.9]

Figure 2: Example of sampling in a small tree

```
10:3: sample : 1  | post : [0.14878487 0.85121513]
11:4: sample : 1  | post : [0.29351627 0.70648373]
12:4: sample : 1  | post : [0.33993407 0.66006593]
13:7: sample : 1  | post : [0.07828721 0.92171279]
14:7: sample : 0  | post : [0.94977111 0.05022889]
15:7: sample : 1  | post : [0.07308321 0.92691679]
16:7: sample : 1  | post : [0.27578656 0.72421344]
17:8: sample : 1  | post : [0.38228583 0.61771417]
18:8: sample : 0  | post : [0.99804483 0.00195517]
19:8: sample : 0  | post : [0.32884468 0.67115532]
20:8: sample : 0  | post : [0.39494232 0.60505768]
21:9: sample : 0
22:10: sample : 1  | post : [0.07293832 0.92706168]
23:10: sample : 1  | post : [0.19671643 0.80328357]
24:11: sample : 0  | post : [0.65369436 0.34630564]
25:12: sample : 0
26:13: sample : 0
27:13: sample : 1
28:14: sample : 1
29:15: sample : 0
30:15: sample : 1
31:15: sample : 1
32:16: sample : 1
33:16: sample : 1
34:16: sample : 1
35:16: sample : 1
36:17: sample : 0
37:17: sample : 0
38:17: sample : 1
39:17: sample : 1
40:18: sample : 0
41:18: sample : 1
42:18: sample : 1
43:18: sample : 1
44:19: sample : 0
```

```
45:19: sample : 1
46:20: sample : 0
47:22: sample : 1
48:22: sample : 1
49:22: sample : 0
50:22: sample : 0
51:23: sample : 0
52:23: sample : 0
53:23: sample : 1
54:24: sample : 0
```

Question 2.6.18

We can recognize a structure similar to HMM in the problem, the only thing that changes is the emission distribution that now is a continuous distribution. Let's see more in detail with the figure 3 below.

$$\begin{tikzcd}
Z_1 \arrow[r] \arrow[d] & Z_2 \arrow[r] \arrow[d] & \cdots \arrow[r] & Z_M \arrow[d] \\
X_1^{(n,n',r)} & X_2^{(n,n',r)} & \cdots & X_M^{(n,n',r)}
\end{tikzcd}$$

Figure 3: Representation as a graphical model

Each Z_i represents in which row of the subfield the match occurs. So we can think of the whole field as being a table of $K = 2$ rows and M columns.

$$Z_i = \begin{cases} 0 & \text{match occurs in first row} \\ 1 & \text{match occurs in second row} \end{cases}$$

And the transition probability between each Z_i is depicted by the table below.

	0	**1**
0	1/4	3/4
1	3/4	1/4

And I chose the probability for the first hidden variable Z_1 is uniform among the two states.

$$p(Z_1 = 0) = p(Z_1 = 1) = \frac{1}{2}$$

The emission distribution is a sum of two independent gaussians, so the sum is itself again another gaussian(I treat β as the variance).

$$a + a' = \mathcal{N}(\mu_s^n, \beta) + \mathcal{N}(\mu_s^{n'}, \beta) = \mathcal{N}(\mu_s^n + \mu_s^{n'}, 2\beta)$$

So we can introduce a new parameter $\mu_s^{(n,n')} = \mu_s^n + \mu_s^{n'}$ for each pair of players leading to a emission distribution of :

$$p(x_t|z_t = k) = \mathcal{N}(x_t|\mu^{(n,n')}, 2\beta)$$

In order to develop the EM algorithm we need the expected complete log likelihood. We begin by writing the complete likelihood:

$$p(X,Z) = \prod_{(r,n,n')}^{[R\times N\times N]} \left[\prod_{m=1}^{M} \left[\prod_{j,k\in\{0,1\}} p\left(z_m = k|z_{m-1} = j\right)^{I(z_m^{(n,n',r)}=k, z_{m-1}^{(n,n',r)}=j)} \right.\right.$$
$$\left.\left. \prod_{k\in\{0,1\}} p\left(x_m^{(n,n')} = x_{(r,n,n')}^m | z_m = k\right)^{I(z_m^{(n,n',r)}=k)} \right]\right]$$

Then we take the *log*:

$$log(p(X,Z)) = \sum_{(r,n,n')}^{[R\times N\times N]} \left[\sum_{m=1}^{M} \left[\sum_{j,k\in\{0,1\}} I(z_m^{(n,n',r)} = k, z_{m-1}^{(n,n',r)} = j) log\left(p\left(z_m = k|z_{m-1} = j\right)\right) \right.\right.$$
$$\left.\left. + \sum_{k\in\{0,1\}} I(z_m^{(n,n',r)} = k) log\left(p(x_m^{(n,n')} = x_{(r,n,n')}^m | z_m = k)\right) \right]\right]$$

$$log(p(X,Z)) = \sum_{(r,n,n')}^{[R\times N\times N]} \left[\sum_{m=1}^{M} \left[\sum_{j,k\in\{0,1\}} I(z_m^{(n,n',r)} = k, z_{m-1}^{(n,n',r)} = j) log\left(A_{j,k}\right) \right.\right.$$
$$\left.\left. + \sum_{k\in\{0,1\}} I(z_m^{(n,n',r)} = k) log\left(\mathcal{N}(x_{(r,n,n')}^m | \mu_{(m,k)}^{(n,n')}, 2\beta)\right) \right]\right]$$

The only part of the $ECLL$ that depends on the parameters, is the summation:

$$\sum_{(r,n,n')}^{[R\times N\times N]} \left[\sum_{m=1}^{M} \left[\sum_{k\in\{0,1\}} \gamma_{m,k}^{(n,n',r)} log\left(\mathcal{N}(x_{(r,n,n')}^m | \mu_{(m,k)}^{(n,n')}, 2\beta)\right) \right]\right] =$$
$$= \sum_{(r,n,n')}^{[R\times N\times N]} \left[\sum_{m=1}^{M} \left[\sum_{k\in\{0,1\}} \frac{-1}{4\beta} \gamma_{m,k}^{(n,n',r)} \left(x_{(r,n,n')}^m - \mu_{(m,k)}^n - \mu_{(m,k)}^{n'}\right)^2 \right] + c(\beta) \right]$$

Then by deriving by $\mu_{m,k}^n$:

$$\frac{\partial ECLL}{\partial \mu_{m,k}^n} = \frac{1}{\cancel{2}\beta} \sum_{n'\neq n} \sum_{r}^{R} \gamma_{m,k}^{(n,n',r)} \left(x_{(r,n,n')}^m - \mu_{(m,k)}^n - \mu_{(m,k)}^{n'}\right) = 0$$

$$\sum_{n'\neq n} \sum_{r}^{R} \gamma_{m,k}^{(n,n',r)} \left(\mu_{(m,k)}^n\right) + \sum_{r}^{R} \gamma_{m,k}^{(n,n',r)} \left(\mu_{(m,k)}^{n'}\right) = \sum_{n'\neq n} \sum_{r}^{R} \gamma_{m,k}^{(n,n',r)} \left(x_{(r,n,n')}^m\right)$$

This leads to a linear system of equations in the variables $\mu_{m,k}^n$, which can then be solved in practice using appropriate library functions. To develop the update rule for β, we derive the above ECLL by β:

$$\frac{\partial ECLL}{\partial \beta} = \sum_{(r,n,n')}^{[R\times N\times N]} \left[\sum_{m=1}^{M} \left[\sum_{k\in\{0,1\}} \gamma_{m,k}^{(n,n',r)} log\left(\mathcal{N}(x_{(r,n,n')}^m | \mu_{(m,k)}^{(n,n')}, 2\beta)\right) \right]\right] =$$

$$= \sum_{(r,n,n')}^{[R\times N\times N]} \left[\sum_{m=1}^{M} \left[\sum_{k\in\{0,1\}} \gamma_{m,k}^{(n,n',r)} \partial \left(-\frac{1}{2} log(2\pi) - \frac{1}{2} log(2\beta) - \frac{1}{4\beta} \left(x_{(r,n,n')}^{m} - \mu_{(m,k)}^{(n,n')} \right)^2 \right) \right] \right] =$$
$$= \sum_{(r,n,n')}^{[R\times N\times N]} \left[\sum_{m=1}^{M} \left[\sum_{k\in\{0,1\}} \gamma_{m,k}^{(n,n',r)} \partial \left(-\frac{1}{2\beta} + \frac{1}{4\beta^2} \left(x_{(r,n,n')}^{m} - \mu_{(m,k)}^{(n,n')} \right)^2 \right) \right] \right] = 0$$

By simplifying a factor of $1/2\beta$ and rearranging the terms we get to the expression:

$$\beta = \frac{\sum_{(r,n,n')}^{[R\times N\times N]} \sum_{m=1}^{M} \sum_{k\in\{0,1\}} \gamma_{m,k}^{(n,n',r)} \left(x_{(r,n,n')}^{m} - \mu_{(m,k)}^{(n,n')} \right)^2}{2 \sum_{(r,n,n')}^{[R\times N\times N]} \sum_{m=1}^{M} \sum_{k\in\{0,1\}} \gamma_{m,k}^{(n,n',r)}}$$

The expectation step is performed by computing $p(z|\mathcal{D}, \theta^{old})$. This is performed using the usual backward forward algorithm. The forward backward algorithm also changes because we have a continuous observed variable, and also its parameters change in every state. This is done by changing expression $p(x|z)$ with the the corresponding normal probability density function.

Question 2.6.19

I will split this question in each of the components I used to solve the problem. I listed some pseudocode for the different part of the algorithm, along with a small explaination. To see the whole code refer to the Appendix.

Initialization parameter pseudocode

The initialization uses the sample means and variance to compute an initial estimate, and then adds a random component. The idea is to solve the following linear system of equations per each subfield using sample means:

$$\begin{cases} \mu_s^{n_1} + \mu_s^{n_2} = \mu_s^{(n_1,n_2)} \\ \mu_s^{n_1} + \mu_s^{n_2} = \mu_s^{(n_1,n_3)} \\ \dots \\ \mu_s^{n_{p-1}} + \mu_s^{n_p} = \mu_s^{(n_{p-1},n_p)} \end{cases}$$

Which is overdetermined, and is solved with a least square approach. For the variance I chose to use the mean of the sample variances divide by two.

```
def generate_parameters(M, N, obs):

    means = compute_sample_means()
    variances = compute_sample_variances()

    starting_variance = mean(variances)
    starting_std_dev = sqrt(starting_variance)

    # solve linear system of equation
    decomposed_m = decompose_means(M, N, means)

    starting_means = [decomposed_m + random.normal(0, starting_std_dev),
                      decomposed_m + random.normal(0, starting_std_dev)]

    return starting_means, starting_std_dev
```

Backward-Forward pseudocode

In the backward-forward code I did not changed much, the only thing to take into consideration is that the emission probability changes per every couple of players and subfield. I choose the initial probability to be uniform between the rows.

EM part

Here is the core of the algorithm, and implements the formulas I derived above. The high level function for the EM algorithm is pretty standard and I can list the code below:

```python
def EM(obs, M, N, threshold=1e-6):

    old_parameters = generate_parameters(M, N, obs)
    converged = False
    rounds = 0

    while not converged:
        responsibilities = E_step(old_parameters, obs)
        parameters = M_step(responsibilities, obs)
        rounds += 1

        distance = np.linalg.norm(old_parameters[0][:, :, 0] - parameters[0][:, :, 0])
                    + np.linalg.norm(old_parameters[0][:, :, 1] - parameters[0][:, :,1])
                    + np.linalg.norm(old_parameters[1] - parameters[0])

        if distance < threshold or rounds > 200:
            converged = True
        else:
            old_parameters = parameters
    return parameters
```

While I also list the pseudocode for the two step of the EM.

```python
def E_step(parameters, observations):

    responsibilities = []

    for couple in couples_players:
        responsibilities[couple] = compute_resp()

    return responsibilities


def M_step(responsibility, observ):
    new_means = np.zeros((M, N, 2))
    new_variance = 0

    # compute new means
    # each column
    for m in range(M):
        # each row
        for k in range(2):
            new_means[m, :, k] = solve_linear_system()


    # compute new variance
    denominator = 0
    numerator = 0

    # apply weighted mean derived formula
    new_variance = compute_variance()

    return new_means, np.sqrt(new_variance)
```

Then to compute the responsabilities I followed the usual approach in HMM, that is to compute the product between the forward and backward variable per each sunbfield m, and then normalize the responsabilities. Again i leave a pseudocode down below.

```
def compute_obs_responsibilities(observation, mean, std_dev):
    alpha, A = forward()
    beta, B = backward()

    R = np.multiply(alpha, beta)
    # normalize
    R = R / np.sum(R, axis=1)[0]

    return R
```

Question 2.6.20

Here below are some results of the algorithm. We can see that the estimated parameters are very similar to the original ones. The approximation of these parameters is better and better as we increase R, which gives more clues about the parameters to estimate, and so less uncertainty. Increasing M or N increases the size of the problem and it requires more data(rounds) to be performed in order to have a good result.

```
M = 4   # columns of the fields
N = 4   # number of players
K = 2   # the two rows
R = 10  # rounds

# Estimated means parameters [MxNxK]

# K = 0
[15.1702861  13.93583887 11.65165402  3.37682257]
[20.4078229   0.65103682 19.37400131 17.5432185 ]
[ 4.85219904  3.75624459 12.2625226   2.20257803]
[ 9.65468817 18.67081753 12.03615569  0.08624936]

# K = 1
[18.05983316 25.94985371 21.70677285 24.03023164]
[18.99806825 22.0579273  34.96114805 19.67350373]
[30.33752971 16.17056561 22.43968071  9.73868544]
[17.39637415 27.42023416 27.65685289 35.11292983]

# Original means parameters [MxNxK]

# K = 0
[11.57765911 14.4973337  15.7178611   4.87061876]
[19.39679232  1.89329753 15.6036313  18.8090173 ]
[ 4.67518618  3.07639342  8.66939437  5.40686221]
[ 8.14856741 18.82199517 11.74382706  2.44278157]

# K = 1
[18.73520686 24.32862951 21.73074779 23.26005311]
[18.14181847 22.2543712  30.75168875 20.41928902]
[31.62149381 14.18290494 21.43482659  9.85930277]
[17.69210866 26.91743931 26.16145816 36.05962046]

# Beta
Estimated : 2.91029627923644
Orignal   : 3.14159265359
```

Question 2.7.21

To perform the clustering we will use a maximum likelihood approach, where we compute the likelihood for each sequence X^n with respect to each class and choose the class that maximizes it, that is:

$$\arg\max_k p(C^n = k|X^n) = \arg\max_k p(X^n|C^n = k)p(C^n = k)$$

The problem is similar to the GMM one.

$$p(X^n, Z^n) = \sum_c \pi_c \left\{p(X^n, Z^n|HHM = c)\right\}$$

We can write the complete likelihood as:

$$p(\mathcal{X}, \mathcal{Z}, \mathcal{C}) = \prod_{n=1}^{N} \prod_{c=1}^{C} \left[p(C = c)p(X^n, Z^n|C^n = c)\right]^{I(C^n=c)}$$

Where the distribution $p(X^n, Z^n|C^n = c)$ represent the complete likelihood for the c^{th} HMM. It is possible to apply the Baum Welch algorithm on each HMM independently, by weighting by the responsibility of each class.

$$E_{CLL} = \sum_{n=1}^{N}\sum_{c=1}^{C} E\left[I(C^n = c)\right] log(p(C = c)) + \sum_{n=1}^{N}\sum_{c=1}^{C} E\left[I(C^n = c) log\left(p(X^n, Z^n|C^n = c)\right)\right] =$$

$$= \sum_{n=1}^{N}\sum_{c=1}^{C} \nu_{n,c} log(\pi_c) + \sum_{n=1}^{N}\sum_{c=1}^{C} \nu_{n,c} E\left[log\left(p(X^n, Z^n|C^n = c)\right)\right]$$

The term $log\left(p(X^n, Z^n|C^n = c)\right)$ is the usual :

$$log\left(p(X^n, Z^n|C^n = c)\right) = log\left(\prod \left(B^c_{ij}\right)^{I(x_t=j, z_t=i)} \prod_{i,j} \left(A^c_{ij}\right)^{I(z_t=i, z_{t-1}=j)}\right)$$

Where we have assumed the starting probability D^c_i as inglobated in the term A^c_{ij}, later we will manage this. The ECLL is then:

$$E_{CLL} = \sum_{c=1}^{C} log(p(C = c)) E\left[\sum_{n=1}^{N} I(C^n = c)\right] + \sum_{n=1}^{N}\sum_{c=1}^{C} I(C^n = c) E\left[p(X^n, Z^n|C^n = c)\right]$$

Where $E[I(C^n = c)] = \nu_{n,c}$ is the responsibility of class c for sequence n. The propagation of the expectation operator into $p(X^n, Z^n|C^n = c)$ gives rise to other factors, the expected counts $\gamma^c_{n,m,r}$ and $\xi^c_{n,m}(j, k)$:

$$p(Z_m = r|X^n, C^n = c) = \gamma^c_{n,m,r}$$
$$p(Z_{m-1} = j, Z_m = k|X^n, C^n = c) = \xi^c_{n,m}(j, k)$$

Which are computed with the usual forward/backward algorithm. For completeness I also add the responsability mentioned before:

$$\nu_{n,c} = p(C^n = c|X^n) = \frac{p(X^n|C^n = c)p(C = c)}{\sum_{c'} p(C^n = c'|X^n)}$$

Where the value $p(X^n|C^n = c)$ is again computed through the forward-backward algorithm. Then in the M step we obtain the new set of parameters by the following formulas, obtained by minimizing the ECLL using the Lagrange multipliers. To note is that we need to compute each of these for each class c.

$$\pi_c = \frac{\sum_n^N \nu_{c,n}}{\sum_{c'}^C \sum_n^N \nu_{c',n}} = \frac{\sum_n^N \nu_{c,n}}{N}$$

To estimate the transition probability per each class the obtained formula is then:

$$A_{ij}^c = \frac{\sum_n^N \nu_{n,c} \sum_{m=2}^M \xi_{n,m}^c(i,j)}{\sum_n^N \nu_{n,c} \sum_{m=2}^M \sum_{j'} \xi_{n,m}^c(i,j')}$$

And for the emission is:

$$B_{ij}^c \propto \sum_n^N \nu_{n,c} \sum_{m=1}^M \gamma_{n,m,i}^c I(x_m^n = j)$$

Where the normalizing factor is computed similar to A_{ij}^c, and can be found by enforcing the fact that given i, we must have a probability distribution in j. For the starting probability we have a simpler expression that is:

$$D_i^c \propto \sum_n^N \nu_{n,c} \gamma_{n,1,i}^c$$

Again this is normalized by the enforcing D_i^c to be a probability distribution in i.

Question 2.7.22

The algorithm is at a high level similar to the one in the previous task, and again we will separate it into the macro functions they perform.I listed some pseudocode for the different part of the algorithm, along with a small explaination. To see the whole code refer to the Appendix.

Forward Backward pseudocode

Here little modifications are to perform: we just have to hand to the functions the correct set of parameters relative to the particular class we are using.

EM algorithm pseudocode

First we start with the initialization of parameters. The transition probabilities are completely random, while the emission probability are also random but they have hyperparameters that make $p(x_n = l|z_n = l)$ more probable. The starting probability are set uniformly, while the class probability still at random.

```
def generate_parameters(observations):
    class_prob = np.random.dirichlet([1] * nr_classes)

    start_prob = [1.0 / float(nr_rows)] * nr_rows

    rand_transition_prob = np.zeros((nr_classes, nr_rows, nr_rows))
    rand_emission_prob = np.zeros((nr_classes, nr_rows, nr_rows))

    for _class in range(nr_classes):
        for r in range(nr_rows):
            transition_prob[_class] = dirichlet(hyperpar)
            hyperpar[r] = nr_rows
            emission_prob[_class] = dirichlet(hyperpar)

    return class_prob, rand_start_prob, transition_prob, emission_prob
```

Then there is the body of the EM algorithm. This is again pretty standard, and the only interesting thing is the stop condition that uses threshold on the sum of the Euclidean distances between all the new parameters and the old ones.

```python
def EM(obs, threshold=1):

    old_parameters = generate_parameters(obs)

    converged = False
    rounds = 0

    while not converged:
        responsibilities = E_step(old_parameters, obs)
        parameters = M_step(responsibilities, obs)
        rounds += 1

        distance = 0
        for i in range(len(parameters)):
            distance += np.linalg.norm(parameters[i] - old_parameters[i])

        if distance < threshold or rounds > 200:
            converged = True
        else:
            old_parameters = parameters
    return parameters
```

The E-step computes the responsibility, that are the parameters ν, ξ, γ, while the M-step computes the new estimate of the parameters.

```python
def E_step(old_parameters, obs):
    # compute class resp for HMM (nu in assignment)
    class_resp = compute_class_responsibility(old_parameters, obs)

    # compute gamma values
    gamma_resp = compute_gamma_resp(old_parameters, obs)

    # compute xi
    xi_resp = compute_xi_resp(old_parameters, obs)

    return class_resp, gamma_resp, xi_resp


def M_step(responsibilities, obs):
    class_resp, gamma_resp, xi_resp = responsibilities

    # new class prob
    new_class_prob = ...

    # transition prob
    new_transition_prob = ...

    # emission prob
    new_emission_prob = ...

    # start prob
    new_start_prob = ...


    return new_class_prob, new_start_prob, new_transition_prob, new_emission_prob
```

The responsibility computation that is performed following the formulas derived above. To see the code, head to the Appendix.

Once the EM is applied to the problem we can use the estimated parameters of each class to finally assign a class to each sequence. This is performed by the function below that computes the first formula of the previous question. Here is a pseudocode.

```
def ML_assignment(params, observations):

    M = zeros([nr_vehicles, nr_classes])

    for n in range(nr_vehicles):
        for c in range(nr_classes):
            a, likelihood = forward()
            M[n, c] = _class_prob[c] * likelihood

    # argmax per each row
    return argmax(M)
```

Question 2.7.23

The first test I performed was with the following parameters: `nr_vehicles = 40 nr_classes = 2 nr_rows = 4 nr_columns = 10`, and the result of the clustering was:

```
# Class of each sample
## Estimated class
[0 0 1 0 0 0 1 1 1 1 0 1 1 0 0 1 1 1 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 1 0 1 0 1 0]
## True classes
[1 1 0 1 1 1 0 0 0 0 1 0 0 1 1 0 0 0 0 0 1 1 1 1 0 1 1 0 1 1 1 0 1 0 0 1 0 1 0 1]
```

As we can see the clustering procedure gives an arbitrary label to each class, and therefore to compute the errors of the algorithm we would need to perform class matching between the two models. In this case the errors are low (*only 2!*) because we had a lot of data (cars), and a small number of classes. Moreover the relatively high number of columns helps because it gives more clues to tell apart the two classes.

If we increase the number of classes it happens sometimes that two classes converge to the same parameters. This is of course an unwanted behaviour because it does not model one class, and makes other two class indistinguishable. To fix this we could define some sparsity measure that we want to optimize.

A result applied to the problem with parameters `nr_vehicles = 40 nr_classes = 4 nr_rows = 7  nr_columns = 10` is shown below:

```
# Class of each sample
## Estimated class
[1 0 1 1 3 3 3 3 3 1 3 1 3 3 1 1 2 2 3 1 0 1 3 1 1 0 0 3 0 1 3 3 3 3 1 0 2 1 3 1]
 ## True classes
[1 0 1 1 2 3 3 3 3 1 3 1 2 3 1 1 1 1 3 1 0 1 3 1 1 3 0 0 0 1 0 0 1 2 1 3 1 1 3 1]
```

As we can see there are more errors, and also it seems that the model for 2 is wrong or overlapped on the model of 1, because it always mispredicts the class as 1.

Appendix

All the code can be found in the repo here, but the main code part are listed below based on the question they belong to.

Question 2.3

```python
def compute_s(node):
    # if this is a leaf
    K = len(node.cat[0])
    s = [0] * K

    if len(node.descendants) == 0:
        s[node.sample] = 1
        # stop recursion
        node.s_i = s
        return

    # recurse to compute s in subtrees
    for child in node.descendants:
        compute_s(child)

    # once s is computed then compute it
    # for the current node.

    for i in range(K):
        children_sub = []
        for child in node.descendants:
            s_uv_i = compute_factor_s(child, i)
            children_sub.append(s_uv_i)
        # update s distribution

        s[i] = np.exp(np.sum(np.log(children_sub)))  # np.prod(children_sub)

    # save in the tree structure
    node.s_i = s
    return

def compute_factor_s(child, father_value):
    K = len(child.cat[0])
    results = [0] * K

    for i in range(K):
        results[i] = child.cat[father_value][i] * child.s_i[i]

    return np.sum(results)

def compute_leaves_probability(tree):
    root = tree.root
    total_sum = []
    for i in range(len(root.cat[0])):
        # for numerical reasons put each addend in a list
        # and use np.sum
        total_sum.append(root.s_i[i] * root.cat[0][i])
    return np.sum(total_sum)
```

Question 2.5

```
def sample_tree(tree):
    compute_s(tree.root)
    sample_node(tree.root)


def sample_node(node):
    # leaves
    if len(node.descendants) == 0:
        return

    compute_node_posterior(node)
    node.sample = np.random.choice(len(node.posterior), p=node.posterior)

    for child in node.descendants:
        sample_node(child)

def compute_node_posterior(node):
    # if node is the root
    if node.ancestor is None:
        node.posterior = np.multiply(node.s_i, node.cat[0])
        node.posterior = node.posterior / np.sum(node.posterior)
    else:
        # if it is a inner node
        node.posterior = np.multiply(node.cat[node.ancestor.sample], node.s_i)
        node.posterior = node.posterior / np.sum(node.posterior)
```

Question 2.6

Initialization parameter code

```
def generate_parameters(M, N, obs):
    means = dict()
    variances = []
    for key, seq in obs.items():
        means[key] = np.mean(seq.values(), axis=0)
        variances.append(np.mean(np.var(seq.values(), axis=0)) / 2)

    starting_variance = np.mean(variances)
    starting_std_dev = np.sqrt(starting_variance)

    decomposed_m = np.matrix(decompose_means(M, N, means))
    starting_means = np.zeros([M, N, 2])
    starting_means[:, :, 0] = decomposed_m + np.random.normal(0, starting_std_dev)
    starting_means[:, :, 1] = decomposed_m + np.random.normal(0, starting_std_dev)

    return starting_means, starting_std_dev

# solve linear system of equation
def decompose_means(M, N, means):
    combinations = int(scipy.misc.comb(N, 2))
    contributions = np.zeros([combinations, N])

    for idx, keys in enumerate(means.keys(), 0):
        contributions[idx, keys[0] - 1] = 1
        contributions[idx, keys[1] - 1] = 1

    values = means.values()
    players_mean_s = []

    for i in range(M):
        results = []

        for j in range(combinations):
            results.append(values[j][i])

        players_mean_s.append(np.dot(np.linalg.pinv(contributions), results))

    return players_mean_s
```

Backward-Forward code

```
def get_transition(r1, m1, r2):
    # calculate A((r1, m1), (r2, m1+1))
    if r1 == r2:
        return 0.25
    else:
        return 0.75


def get_emission(r, o, m, means, std_dev):
    # calculate O((m,r), o)
    mean = means[r][m]

    result = norm.pdf(o, loc=mean, scale=np.sqrt(2) * std_dev)

    return result


def get_init():
    # provide an array containing the initial state probability
    # having size R
    pi = np.array([0.5, 0.5])
    # number of rows
    R = pi.shape[0]
    return pi, R


def forward(get_init, get_transition, get_emission, observations, means, std_dev):
    pi, R = get_init()
    M = len(observations)
    alpha = np.zeros((M, R))

    # base case
    O = []
    for r in range(R):
        O.append(get_emission(r, observations[0], 0, means, std_dev))
    alpha[0, :] = pi * O[:]

    # recursive case
    for m in range(1, M):
        for r2 in range(R):
            for r1 in range(R):
                transition = get_transition(r1, m, r2)
                emission = get_emission(r2, observations[m], m, means, std_dev)
                alpha[m, r2] += alpha[m - 1, r1] * transition * emission

    return (alpha, np.sum(alpha[M - 1, :]))


def backward(get_init, get_transition, get_emission, observations, means, std_dev):
    pi, R = get_init()
    M = len(observations)
    beta = np.zeros((M, R))

    # base case
    beta[M - 1, :] = 1

    # recursive case
    for m in range(M - 2, -1, -1):
        for r1 in range(R):
            for r2 in range(R):
                transition = get_transition(r1, m, r2)
                emission = get_emission(r2, observations[m + 1], m + 1, means, std_dev)
                beta[m, r1] += beta[m + 1, r2] * transition * emission

    O = []
    for r in range(R):
        O.append(get_emission(r, observations[0], 0, means, std_dev))

    return beta, np.sum(pi * O[:] * beta[0, :])
```

EM part

```
def EM(obs, M, N, threshold=1e-6):
    # parameters = (means, std_dev)
    old_parameters = generate_parameters(M, N, obs)
    converged = False
    rounds = 0

    while not converged:
        responsibilities = E_step(old_parameters, obs)
        parameters = M_step(responsibilities, obs)
        rounds += 1

        distance = np.linalg.norm(old_parameters[0][:, :, 0] - parameters[0][:, :, 0])
                   + np.linalg.norm(old_parameters[0][:, :, 1] - parameters[0][:, :,1])
                   + np.linalg.norm(old_parameters[1] - parameters[0])

        if distance < threshold or rounds > 200:
            converged = True
        else:
            old_parameters = parameters
    return parameters


def E_step(parameters, observations):
    means = parameters[0]
    std_dev = parameters[1]
    responsibilities = dict()

    for couple_p in observations.keys():
        responsibilities[couple_p] = []

        # summing up the means of the two players
        couple_means = [means[:, couple_p[0] - 1, 0] + means[:, couple_p[1] - 1, 0],
                        means[:, couple_p[0] - 1, 1] + means[:, couple_p[1] - 1, 1]]

        for instance in observations[couple_p].values():
            instance_resp = compute_obs_responsibilities(instance,
                                                         couple_means, std_dev)
            responsibilities[couple_p].append(instance_resp)
    return responsibilities


def M_step(responsibility, observ):
    new_means = np.zeros((M, N, 2))
    new_variance = 0
    for m in range(M):
        for k in range(2):

            # compute system of equation
            linear_system = []
            linear_known = []
            for p in range(1, N + 1):
                coefficients = [0] * N
                known_term = 0

                # compute known term
                for pair in observ.keys():
                    if p not in pair:
                        continue
                    for r in observ[pair].keys():
                        known_term = known_term + observ[pair][r][m]
                                * responsibility[pair][r - 1][m, k]

                    for r in observ[pair].keys():
                        p2 = pair[0] if p == pair[1] else pair[1]
                        coefficients[p - 1] += responsibility[pair][r - 1][m, k]
                        coefficients[p2 - 1] += responsibility[pair][r - 1][m, k]

                linear_system.append(coefficients)
                linear_known.append(known_term)

            param_m_k = np.linalg.solve(linear_system, linear_known)
            new_means[m, :, k] = param_m_k

    # compute new variance
    denominator = 0
    numerator = 0
    for pair in observation.keys():
        for m in range(M):
            for k in range(2):
```

```
            pair_mean = new_means[m, pair[0] - 1, k] + new_means[m, pair[1] - 1, k]
            for r in observation[pair].keys():
                numerator += responsibility[pair][r - 1][m, k]
                    * np.power(observation[pair][r][m] - pair_mean, 2)
                denominator += responsibility[pair][r - 1][m, k]

    new_variance = numerator / (2.0 * denominator)

    return new_means, np.sqrt(new_variance)

def compute_obs_responsibilities(observation, mean, std_dev):
    alpha, A = forward(get_init, get_transition, get_emission,
                       observation, mean, std_dev)
    beta, B = backward(get_init, get_transition, get_emission,
                       observation, mean, std_dev)

    R = np.multiply(alpha, beta)
    # normalize
    R = R / np.sum(R, axis=1)[0]

    return R
```

Question 2.7

Forward Backward

```
def get_transition(r1, m1, r2, param):
    return param[r1][r2]

def get_emission(r, o, param):
    return param[r][o]

def get_init(param):
    return np.array(param), nr_rows


def forward(get_init, get_transition, get_emission,
                observations, trans, emiss, start_prob):
    pi, R = get_init(start_prob)
    M = len(observations)
    alpha = np.zeros((M, R))

    # base case
    O = []
    for r in range(R):
        O.append(get_emission(r, observations[0], emiss))
    alpha[0, :] = pi * O[:]

    # recursive case
    for m in range(1, M):
        for r2 in range(R):
            for r1 in range(R):
                transition = get_transition(r1, m, r2, trans)
                emission = get_emission(r2, observations[m], emiss)
                alpha[m, r2] += alpha[m - 1, r1] * transition * emission
    return (alpha, np.sum(alpha[M - 1, :]))


def backward(get_init, get_transition, get_emission,
                     observations, trans, emiss, start_prob):
    pi, R = get_init(start_prob)
    M = len(observations)
    beta = np.zeros((M, R))

    # base case
    beta[M - 1, :] = 1

    # recursive case
    for m in range(M - 2, -1, -1):
        for r1 in range(R):
            for r2 in range(R):
                transition = get_transition(r1, m, r2, trans)
                emission = get_emission(r2, observations[m + 1], emiss)
                beta[m, r1] += beta[m + 1, r2] * transition * emission

    O = []
    for r in range(R):
        O.append(get_emission(r, observations[0], emiss))

    return beta, np.sum(pi * O[:] * beta[0, :])
```

EM algorithm

```
def generate_parameters(observations):
    rand_class_prob = np.random.dirichlet([1] * nr_classes)

    rand_start_prob = np.matrix([[1.0 / float(nr_rows)] * nr_rows]*nr_classes)

    rand_transition_prob = np.zeros((nr_classes, nr_rows, nr_rows))
    rand_emission_prob = np.zeros((nr_classes, nr_rows, nr_rows))

    for c in range(nr_classes):
        for r in range(nr_rows):
            hyperpar = [1] * nr_rows
            rand_transition_prob[c, r, :] = np.random.dirichlet(hyperpar)
            hyperpar[r] = nr_rows
            rand_emission_prob[c, r, :] = np.random.dirichlet(hyperpar)

return rand_class_prob, rand_start_prob, rand_transition_prob, rand_emission_prob


def EM(obs, threshold=1):

    old_parameters = generate_parameters(obs)

    converged = False
    rounds = 0

    while not converged:
        responsibilities = E_step(old_parameters, obs)
        parameters = M_step(responsibilities, obs)
        rounds += 1

        distance = 0
        for i in range(len(parameters)):
            distance += np.linalg.norm(parameters[i] - old_parameters[i])

        if distance < threshold or rounds > 200:
            converged = True
        else:
            old_parameters = parameters
    return parameters


def E_step(old_parameters, obs):
    # compute class resp for HMM (nu in assignment)
    # size : classes x cars
    class_resp = compute_class_responsibility(old_parameters, obs)

    # compute gamma values
    # size : dict(car) = column x row_value x classes
    gamma_resp = compute_gamma_resp(old_parameters, obs)

    # compute xi
    # size : dict(car){dic(class){[row x row x columns]}}
    xi_resp = compute_xi_resp(old_parameters, obs)

    return class_resp, gamma_resp, xi_resp


def M_step(responsibilities, obs):
    class_resp, gamma_resp, xi_resp = responsibilities

    # new class prob
    new_class_prob = np.sum(class_resp, axis=1) / np.sum(class_resp)

    # transition prob
    new_transition_prob = np.zeros([nr_classes, nr_rows, nr_rows])
    for c in range(nr_classes):
        for n in range(nr_vehicles):
            new_transition_prob[c, :, :] += class_resp[c, n] *
                                            np.sum(xi_resp[n][c], axis=2)
        # normalization
        for r in range(nr_rows):
            new_transition_prob[c, r, :] = new_transition_prob[c, r, :] /
                                     (np.sum(new_transition_prob[c, r, :]))

    # emission prob
    new_emission_prob = np.zeros([nr_classes, nr_rows, nr_rows])

    for c in range(nr_classes):
        for n in range(nr_vehicles):

            resp = gamma_resp[n][:, :, c]
            for m in range(nr_columns):
                o = obs[n, m]
                for r in range(nr_rows):
                    new_emission_prob[c, :, r] += class_resp[c, n] *
                                                    resp[m, :] * (o == r)
            # normalization
            for r in range(nr_rows):
                new_emission_prob[c, r, :] = new_emission_prob[c, r, :] /
                                            (np.sum(new_emission_prob[c, r, :]))


new_start_prob = np.zeros([nr_classes,nr_rows])

    for n in range(nr_vehicles):
        for c in range(nr_classes):
            new_start_prob[c, :] += class_resp[c, n] * gamma_resp[n][1, :, c]

    # normalizing
    for r in range(nr_classes):
        new_start_prob[r,:] = new_start_prob[r,:] / np.sum(new_start_prob[r,:])


    return new_class_prob, new_start_prob, new_transition_prob, new_emission_prob


# what I called nu in the previous question
def compute_class_responsibility(params, obs):
    _class_prob, _start_prob, _transition_prob, _emission_prob = params
    # matrix C (classes) x N (samples)
    class_resp = np.zeros([nr_classes, nr_vehicles])

    for c in range(nr_classes):
        for n in range(nr_vehicles):
            o = obs[n, :]
            b, L = forward(get_init, get_transition, get_emission, o,
                           _transition_prob[c, :, :], _emission_prob[c, :, :])
            class_resp[c, n] = L * _class_prob[c]

    class_resp = class_resp / np.sum(class_resp, axis=0)

    return class_resp


def compute_gamma_resp(params, obs):
    _class_prob, _start_prob, _transition_prob, _emission_prob = params

    gamma = dict()

    for n in range(nr_vehicles):
        gamma[n] = np.zeros([nr_columns, nr_rows, nr_classes])

        for c in range(nr_classes):
            # compute responsibility for column
            # m of class c of the observation of
            # car n
            o = obs[n, :]
            alpha, A = forward(get_init, get_transition, get_emission, o,
                    _transition_prob[c, :, :], _emission_proB[c, :, :],_start_prob[c,:])
            beta, B = backward(get_init, get_transition, get_emission, o,
                    _transition_prob[c, :, :], _emission_prob[c, :, :],_start_prob[c,:])

            for m in range(nr_columns):
                gamma[n][m, :, c] = np.multiply(alpha[m, :], beta[m, :])
                # normalize
                gamma[n][m, :, c] = gamma[n][m, :, c] / np.sum(gamma[n][m, :, c])

    return gamma


def compute_xi_resp(params, obs):
    # size : dict(car){dic(class){[row x row x columns]}}
    _class_prob, _start_prob, _transition_prob, _emission_prob = params
    xi = dict()
    for n in range(nr_vehicles):
        xi[n] = dict()
        o = obs[n, :]
        for c in range(nr_classes):
            xi[n][c] = np.zeros([nr_rows, nr_rows, nr_columns])

            alpha, A = forward(get_init, get_transition, get_emission, o,
                   _transition_prob[c, :, :], _emission_proB[c, :, :],_start_prob[c,:])
            beta, B = backward(get_init, get_transition, get_emission, o,
                    _transition_prob[c, :, :], _emission_prob[c, :, :],_start_prob[c,:])

            transition_v = _transition_prob[c, :, :]
            # now compute through time
            for m in range(1, nr_columns):
                emission_v = _emission_prob[c, :, o[m]]
                temp = np.multiply(emission_v, np.transpose(beta[m, :]))
                xi[n][c][:, :, m] = np.multiply(transition_v,
                                                np.outer(alpha[m - 1, :], temp))
    return xi

def ML_assignment(params, obs):
    _class_prob, _start_prob, _transition_prob, _emission_prob = params
    M = np.zeros([nr_vehicles, nr_classes])

    for n in range(nr_vehicles):
        for c in range(nr_classes):
            a, likelihood = forward(get_init, get_transition, get_emission, o,
                   _transition_prob[c, :, :], _emission_proB[c, :, :],_start_prob[c,:])
            M[n, c] = _class_prob[c] * likelihood

    return np.argmax(M,axis=1)
```