DD2434 Advanced Machine Learning
2021-2022

Assignment 3: Random Projections, Graph Theory, Variational Inference, Expectation-Maximization, and Hidden Markov Models

---

Martín Iglesias Goyanes
martinig@kth.se

School of Electrical Engineering and Computer Science
Stockholm, 23 December 2021

3.1 Success probability in the Johnson-Lindenstrauss lemma

In the proof of Johnson-Lindenstrauss lemma we first bounded the probability that a single projection maintains all pairwise distances with distortion that is between $(1-\epsilon)$ and $(1+\epsilon)$. In particular, we showed that the probability of achieving such a distortion for all pairs of points is at least $1/n$. Assume now, that we want to boost the probability of success to be at least 95%.

Question 3.1.1

Show that O(n) independent trials are sufficient for the probability of success to be at least 95%. An independent trial here refers to generating a new projection of the data points with a newly generated projection matrix.

We know that a single projection keeps all pairwise distances with a distortion between $(1-\epsilon)$ and $(1+\epsilon)$, from Johnson-Lindenstrauss [1]. Also, from [1], achieving a distortion between $(1-\epsilon)$ and $(1+\epsilon)$ for all pairs of points has a probability of at least $1/n$. By repeating this projection more times we can increase our success probability to any desired constant [1].

The algorithm we employ follows like this: we sample a random projection and check if it lies inside our distortion interval, else, the sample is discarded and the process is restarted. From now on, we denote Y as the number of times that our projection lies inside the desired distortion interval. Also, we denote p as the probability the projection has of being inside that distortion interval, i.e, $p = 1/n$. We will do $O(n)$ independent trials. Therefore $Y \sim Bin(O(n)p)$.

We are asked to prove that $O(n)$ independent trials are sufficient for the probability of success to be at least 95%, i.e, $p(Y \geq 1) \geq 0.95$ has to be proved. Or what is the same $p(Y = 0) \leq 0.05$ since $p(Y \geq 1) = 1 - p(Y = 0)$.

Since we are doing $O(n)$ trials then we use cn as the number of trials because $O(n)$ refers to $n, 2n, 3n, ...$, this in combination with the fact that $Y \sim Bin(O(n)p)$ leads to:

$$p(Y \geq 1) = 1 - p(Y = 0) = 1 - \binom{cn}{0} p^0 (1-p)^{n-0} = 1 - (1 - \frac{1}{n})^{cn}$$

where as n grows to infinity, $p(Y \geq 1) = 1 - \frac{1}{e^c}$.

From here, it is easy to see how to obtain a probability of success of at least 95% we will have to solve for c such that $1 - \frac{1}{e^c} \geq 0.95$. Which in this case, it will result in $c \geq \log(20)$. However, this is true in the case we are able to work with inifinity n, if we work with finite trials then we have to use again cn as the number of trials:

$$p(Y \geq 1) = 1 - p(Y = 0) = 1 - \binom{cn}{0} p^0 (1-p)^{n-0} = 1 - (1 - \frac{1}{n})^{cn} \rightarrow (1 - \frac{1}{n})^{cn} \leq 0.05$$

When solving it, we obtain: $cn \geq \frac{\log(0.05)}{\log(1-\frac{1}{n})}$ which is a valid solution to what was asked since $cn \in O(n)$.

3.2 Node similarity for representation learning

Let $G = (V, E)$ be an undirected and connected graph and let $\mathbf{A}$ be the adjacency matrix of G, that is, $\mathbf{A}_{ij} = 1$ if $(i, j) \in E$ and $\mathbf{A}_{ij} = 0$ otherwise. Let $\mathbf{D}$ be a diagonal matrix with $\mathbf{D}_{ii} = \sum_j \mathbf{A}_{ij}$, and let $\mathbf{P} = \mathbf{D}^{-1}\mathbf{A}$. In graph representation learning, our goal is to learn vector representations

(embeddings) for the nodes of the graph. The main idea is to define an appropriate similarity measure between the graph nodes, and then learn vector representations for the graph nodes, so that the similarity between pairs of learned vectors approximates the similarity between the corresponding graph nodes. Assume now that for a similarity measure between graph nodes, we define

$$\mathbf{S}_{ij} = \sum_{k=1}^{\infty} \alpha^k \left[\mathbf{P}^k\right]_{ij}$$

for each pair of nodes $i, j \in V$, and for some real $0 < \alpha < 1$. Here, by $\left[\mathbf{P}^k\right]_{ij}$ we refer to the (i, j) entry of the matrix $\mathbf{P}^k$.

Question 3.2.2

Explain the intuition for the definition of the similarity measure S.

Leo Katz introduced this equation in the context of voting and it is known as Katz Centrality or Katz Index [2]. Katz, with the above equation, proposed a measure of centrality in a network which measures the influence of each node [2]. The intuition behind it is that a node i should be as important as other nodes in the graph G choose it to be. Graph nodes vote on each other directly if a pair of nodes are direct neighbors or indirectly otherwise. As the chain of neighbors increases, the effect of votes from indirect neighbors is decreased with the use of α, a dampling factor [2].

$S_{i,j}$ counts the amount of all lengths paths between node i and node j. We want close neighbors to contribute more to the similarity measure therefore we use $0 < \alpha < 1$ to penalize long walks through the graph. When using an α close to 1 we are favorable to long walks, whereas α closer to 0 leads to harsh penalization of long walks.

Question 3.2.3

Show that S can be computed efficiently using matrix addition, and a single matrix inversion operation, while avoiding computing an infinite series.

If we refer again to [2] we can see how to use only additions and matrices inversions to compute geometric series with the following theorem: $\sum_{k=0}^{\infty} \mathbf{P}^k = (\mathbf{I} - \mathbf{P})^{-1}$ which holds only when $(\mathbf{I} - \mathbf{P})$ has an inverse, i.e, is non singular and when $\mathbf{P}$'s largest eighenvalue $\lambda < 1$ [2]. We define $s_n = \sum_{k=0}^{n} \mathbf{P}^k$.

Now if we apply some algebra to s_n:

$$\mathbf{P}s_n = \mathbf{P}\sum_{k=0}^{n} \mathbf{P}^k = \sum_{k=1}^{n+1} \mathbf{P}^k$$

Now combining equations we get:

$$s_n - \mathbf{P}s_n = \sum_{k=0}^{n} \mathbf{P}^k - \sum_{k=1}^{n+1} \mathbf{P}^k$$

$$s_n(\mathbf{I} - \mathbf{P}) = \mathbf{I} + \sum_{k=1}^{n} \mathbf{P}^k + \sum_{k=1}^{n} \mathbf{P}^k - \mathbf{P}^{n+1}$$

$$s_n = (\mathbf{I} - \mathbf{P}^{n+1})(\mathbf{I} - \mathbf{P})^{-1}$$

As stated above we are assuming $\mathbf{P}$'s largest eigenvalue $\lambda < 1$ therefore $\mathbf{P}^n = 0$ as n approaches infinity. This applied to the definition of s_n shows that:

$$\lim_{x \to \infty} s_n = \lim_{x \to \infty} (\mathbf{I} - \mathbf{P}^{n+1})(\mathbf{I} - \mathbf{P})^{-1} = (\mathbf{I} - \mathbf{P})^{-1}$$

If we now apply this to the computation of $S_{i,j}$ we can see how it can be computed with only matrix inversion and addition:

$$S_{i,j} = (\mathbf{I} - \alpha\mathbf{P})^{-1} - \mathbf{I}$$

3.3 Complicated likelihood for leaky units on a tree

Consider the following model. A binary tree T has random variables associated with its vertices. A vertex u has an observable variable X_u and a latent class variable Z_u. Each class $c \in [C]$ has a Normal distribution $\mathcal{N}(\mu_c, \sigma^2)$. If the three neighbors of u are v_1, v_2, and v_3, then

$$p(X_u \mid Z_u = c, Z_{v_1} = c_1, Z_{v_2} = c_2, Z_{v_3} = c_3) \sim \mathcal{N}\left(X_u \mid (1-\alpha)\mu_c + \sum_{i=1}^{3} \frac{\alpha}{3}\mu_{c_i}, \sigma^2\right)$$

The class variables are i.i.d., each follows the categorical distribution π.

Question 3.3.4

Provide a linear time algorithm that computes $p(X \mid T, M, \sigma, \alpha, \pi)$ when given a tree T (with vertices $V(T)$), observable variables for its vertices $X = \{X_v : v \in V(T)\}$, and parameters $M = \{\mu_c : c \in [C]\}, \sigma, \alpha$. Note: : For root vertex you can assign the weights as $(1-\alpha)$ & $(\alpha/2)$ and for a leaf vertex $(1-\alpha)$ & α.

We will design a dynamic programming algorithm that is able to compute $p(X \mid T, M, \sigma, \alpha, \pi)$ in linear time. In order to do so we will split the computation of $p(X \mid T, M, \sigma, \alpha, \pi)$ in smaller sub-problems until we hit a base or leaf case where the dynamic programming algorithm can stop at. For simplicity, we will denote $p(X \mid T, M, \sigma, \alpha, \pi)$ as $p(X)$. We will compute this $p(X)$ by marginalizing:

$$p(X) = \sum_{Z} p(X, Z)$$

Given a binary tree, T, we will split the computation in 3 different smaller sub-problems: leaf nodes, inner nodes and the root node. The derivation will be done as the dynamic programming would run, in a bottom up approach, from leaves to root.

- **Leaf nodes:** Let a leaf node be named u with parent $pa(u)$, since it is a leaf node, it has no children v_1 and v_2. From the statement we know the weights are $(1-\alpha)$ and α.

$$p(X_u) = p\left(X_u \mid Z_u, Z_{\text{pa}(u)}\right) = \mathcal{N}\left(X_u \mid (1-\alpha)\mu_{Z_u} + \alpha\mu_{Z_{\text{pa}(u)}}, \sigma^2\right) \tag{1}$$

- **Inner nodes:** Let a inner node be named u with parent $pa(u)$ and children v_1 and v_2. From now on $\downarrow$ refers to the children of that node. Note the latent variables Z are independent.

$$\begin{aligned}
p(X_u) &= \sum_{Z_{u\downarrow}} p\left(X_u, X_{u\downarrow}, Z_{u\downarrow} \mid Z_u, Z_{\text{pa}(u)}\right) = \\
&= \sum_{Z_{u\downarrow}} p\left(X_u, X_{v_1}, X_{v_2}, X_{v_1\downarrow}, X_{v_2\downarrow}, Z_{v_1\downarrow}, Z_{v_2\downarrow} \mid Z_u, Z_{\text{pa}(u)}, Z_{v_1}, Z_{v_2}\right) p(Z_{v_1})\, p(Z_{v_2}) \\
&= \sum_{Z_{u\downarrow}} [p\left(X_u \mid Z_u, Z_{\text{pa}(u)}, Z_{v_1}, Z_{v_2}\right) p(Z_{v_1})\, p(Z_{v_2}) \\
&\quad p(X_{v_1}, X_{v_1\downarrow}, Z_{v_1\downarrow} \mid Z_{v_1}, Z_u) \\
&\quad p(X_{v_2}, X_{v_2\downarrow}, Z_{v_2\downarrow} \mid Z_{v_2}, Z_u)]
\end{aligned}$$

Now we can we can sum over the children of u, since T is binary the only children are v_1 and v_2:

$$= \sum_{Z_{v_1}, Z_{v_2}} \left[p\left(X_u \mid Z_u, Z_{\text{pa}(u)}, Z_{u_1}, Z_{u_2}\right) p\left(Z_{u_1}\right) p\left(Z_{u_2}\right) \times \right.$$
$$\underbrace{\left(\sum_{Z_{v_1\downarrow}} p(X_{v_1}, X_{v_1\downarrow}, Z_{v_1\downarrow} \mid Z_{v_1}, Z_u)\right)}_{p(X_{v_1})} \times \underbrace{\left(\sum_{Z_{v_2\downarrow}} p(X_{v_2}, X_{v_2\downarrow}, Z_{v_2\downarrow} \mid Z_{v_2}, Z_u)\right)}_{p(X_{v_2})} \left.\right]$$

The recursion can now start to be seen in the two last terms, its again the computation of inner nodes, v_1 and v_2. Then the algorithm just needs to keep computing the $p(X_u)$ for all inner nodes until it reaches the leaf nodes, in which case it returns the value from the value from previously defined expression 1. The next step to simplify this expression is substitute in the known probability densities and group the terms to make the recursion more explicit:

$$\begin{aligned} p(X_u) &= \sum_{Z_{v_1}, Z_{v_2}} [p(X_u \mid Z_u, Z_{\text{pa}(u)}, Z_{v_1}, Z_{v_2}) \times p(Z_{v_1}) \times p(Z_{v_2}) \times p(X_{v_1}) \times p(X_{v_2})] \\ &= \sum_{Z_{v_1}, Z_{v_2}} [\mathcal{N}(X_u \mid (1-\alpha)\mu_{Z_u} + \frac{\alpha}{3}(\mu_{Z_{v_1}} + \mu_{Z_{v_2}} + \mu_{Z_{\text{pa}(u)}}), \sigma^2) \times \pi_{Z_{v_1}} \times \pi_{Z_{v_2}} \times p(X_{v_1}) \times p(X_{v_2})] \end{aligned} \tag{2}$$

Since $p(X_{v_1})$ is independent of X_{v_2} or Z_{v_2} and the same happens for $p(X_{v_2})$ with X_{v_1} and Z_{v_1}, when summing over Z_{v1} the algorithm only has to compute $p(X_{v_1})$ once, store it and reuse it. **This property is what allows the algorithm to run in linear time since it reuses previously calculated values instead of computing them again and again.**

- **Root node:** Let a root node be name as u and have children v_1 and v_2. Then when the algorithm arrives to the root node it can reuse all the computed values up to v_1 and v_2 to evaluate the following expression:

$$\begin{aligned} p(X) &= \sum_{Z} p\left(X_u, X_{v_1}, X_{v_2}, X_{v_1\downarrow}, X_{v_2\downarrow}, Z_{v_1\downarrow}, Z_{v_2\downarrow} \mid Z_u, Z_{v_1}, Z_{v_2}\right) p\left(Z_u, Z_{v_1}, Z_{v_2}\right) \\ &= \sum_{Z} p\left(X_u \mid Z_{v_1}, Z_{v_2}, Z_u\right) p\left(Z_{v_1}, Z_{v_2}, Z_u\right) p\left(X_{v_1}, X_{v_1\downarrow}, Z_{v_1\downarrow} \mid Z_{v_1}, Z_u\right) p\left(X_{v_2}, X_{v_2\downarrow}, Z_{v_2\downarrow} \mid Z_{v_2}, Z_u\right) \\ &= \sum_{Z_u, Z_{v_1}, Z_{v_2}} \left[p\left(X_u \mid Z_{v_1}, Z_{v_2}, Z_u\right) p\left(Z_{v_1}, Z_{v_2}, Z_u\right) \right. \\ &\left. \left(\sum_{Z_{v_1\downarrow}} p\left(X_{v_1}, X_{v_1\downarrow}, Z_{v_1\downarrow} \mid Z_{v_1}, Z_u\right)\right) \left(\sum_{Z_{v_2\downarrow}} p\left(X_{v_2}, X_{v_2\downarrow}, Z_{v_2\downarrow} \mid Z_{v_2}, Z_u\right)\right)\right] \end{aligned}$$

Now we substitute the known densities and use the fact that the latent variables Z are independent given π. Also, we must note that the weights in this case are $(1-\alpha)$ and $\frac{\alpha}{2}$.

$$p(X) = \sum_{Z_u, Z_{v_1}, Z_{v_2}} \left[\mathcal{N}\left(X_u \mid (1-\alpha)\mu_{Z_u} + \frac{\alpha}{2}\left(\mu_{Z_{v_1}} + \mu_{Z_{v_2}}\right), \sigma^2\right) \pi\left(Z_u\right) \pi\left(Z_{v_1}\right) \pi\left(Z_{v_2}\right) \right.$$
$$\underbrace{\left(\sum_{Z_{v_1\downarrow}} p\left(X_{v_1}, X_{v_1\downarrow}, Z_{v_1\downarrow} \mid Z_{v_1}, Z_u\right)\right)}_{p(X_{v_1})} \underbrace{\left(\sum_{Z_{v_2\downarrow}} p\left(X_{v_2}, X_{v_2\downarrow}, Z_{v_2\downarrow} \mid Z_{v_2}, Z_u\right)\right)}_{p(X_{v_2})} \left.\right]$$

Where $p(X_{v1})$ and $p(X_{v2)}$ can be computed from the stored values by the dynamic programming algorithm with expression 2, since they are just inner nodes of the binary tree T.

3.4 Super Epicentra - Variational Inference

As in Task 2.4 from Assignment 2, we have seismographic from an area with frequent earthquakes emanating from K super epicentra. In fact, the core of the present model is the model described in Task 2.4 from Assignment 2, but now the parameters also have conjugate prior distributions. As shown in Figure 1 , the present model has the following prior distributions.

1. π has a $\mathrm{Dir}(\alpha)$ prior.
2. $\tau_{k,i}$ has a $\mathrm{Ga}\left(\alpha', \beta'\right)$ prior.
3. $\mu_{k,i}$ has a $\mathcal{N}\left(\mu, (C\tau_{k,i})^{-1}\right)$ prior.
4. λ_k has a $\mathrm{Ga}\left(\alpha_0, \beta_0\right)$ prior.

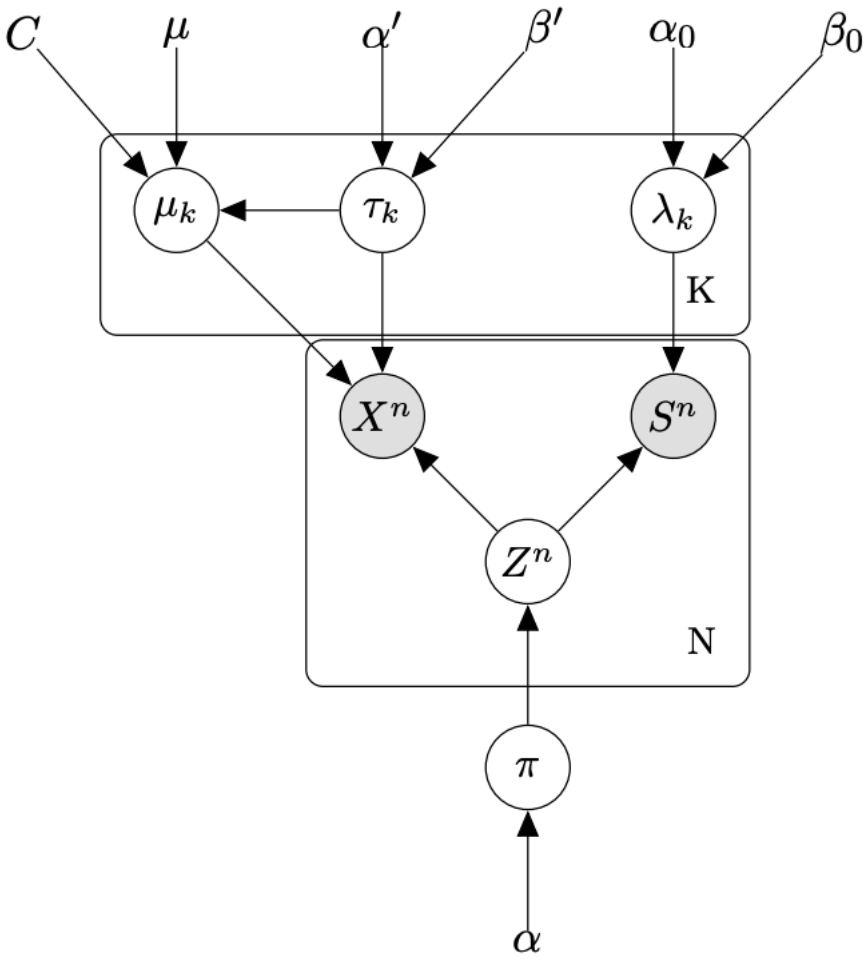


Figure 1: The K super epicentra model with priors

Question 3.4.5

Derive a VI algorithm that estimates the posterior distribution for this model.

The derivation for the following Variational Inference algorithm is based on notes from lectures and the course reference book [3].Now we define:

- $\mathbf{X} = \{X_n : n \in [N]\}$
- $\mathbf{Z} = \{Z_n : n \in [N]\}$
- $\mathbf{M} = \{\mu_k : k \in [K]\}$
- $\mathbf{T} = \{\tau_k : k \in [K]\}$
- $\mathbf{\Lambda} = \{\lambda_k : k \in [K]\}$
- $\mathbf{S} = \{S_n : n \in [N]\}$

From the statement we can note the all the densities for the variables in the model:

- $\pi \mid \alpha \sim \mathbf{Dir}(\alpha) = \frac{1}{B(\alpha)} \cdot \prod_{k=1}^{k} \pi_k^{\alpha_{k-1}}$ where $B(\alpha)$ is the beta function.
- $\tau_{\mathbf{k,i}} \mid \alpha^{'}, \beta^{'} \sim \mathbf{Gamma}(\alpha^{'}, \beta^{'}) = \frac{(\beta^{'})^{\alpha^{'}}}{\Gamma(\alpha^{'})} \tau_{k,i}^{[\alpha^{'}-1]} e^{\left[-\beta^{'}\tau_{k,i}\right]}$ where $\Gamma(\alpha^{'})$ is the gamma function.
- $\mu_{\mathbf{k,i}} \mid \mu, \mathbf{C}, \tau_{\mathbf{k,i}} \sim \mathbf{Normal}(\mu, (\mathbf{C}\tau_{\mathbf{k,i}})^{-\mathbf{1}}) = \sqrt{\frac{C\tau_{k,i}}{2\pi}} e^{\left[-\frac{C\tau_{k,i}}{2}\left(\mu_{k,i}-\mu\right)^2\right]}$
- $\lambda_{\mathbf{k}} \mid \alpha_{\mathbf{0}}, \beta_{\mathbf{0}} \sim \mathbf{Gamma}(\alpha_{\mathbf{0}}, \beta_{\mathbf{0}}) = \frac{(\beta_0)^{\alpha_0}}{\Gamma(\alpha_0)} \lambda_k^{[\alpha_0-1]} e^{[-\beta_0\lambda_k]}$
- $\mathbf{Z_n} \mid \pi \sim \mathbf{Cat}(\pi)$
- $\mathbf{X_n} \mid \mathbf{Z_n} = \mathbf{k}, \mathbf{M}, \mathbf{T} \sim \mathbf{Normal}(\mu_{\mathbf{k}}, \tau_{\mathbf{k}}^{-\mathbf{1}}) = \sqrt{\frac{\tau_k}{2\pi}} e^{\left[-\frac{\tau_k}{2}(x_n-\mu_k)^2\right]}$
- $\mathbf{S_n} \mid \mathbf{Z_n} = \mathbf{k}, \mathbf{L} \sim \mathbf{Poisson}(\lambda_{\mathbf{k}}) = \frac{\lambda_k^{S_n} e^{-S_n}}{S_n!}$

From the given graphical model we can express the posterior distribution in this way:

$$p(Z, \Theta \mid X, S) = p(Z, \pi, T, M, \Lambda \mid X, S)$$

and the joint distribution in this manner, where the factorization comes from the independence relations shown in the graphical model:

$$\begin{aligned} p(\pi, T, M, \Lambda, X, S, Z) &= p(X, S, Z \mid M, T, \Lambda, \pi) p(M, T, \Lambda, \pi) \\ &= p(X \mid Z, M, T) p(S \mid \Lambda, Z) p(Z \mid \pi) p(M \mid T) p(T) p(\Lambda) p(\pi) \end{aligned}$$

Now we want a variational distribution that factorizes. In this case the factorization splits apart the latent variables $\mathbf{Z}$ and the parameters $\Theta = (\pi, T, M, \Lambda)$:

$$q(Z, \Theta) = q(Z) q(\Theta)$$

Now, we want to obtain the update equations the algorithm is going to iterate on in order to obtain the optimal (*) parameters that would result in the optimal approximate posterior $q^*(Z, \Theta)$

1. **Update equation for $\mathbf{Z_n}$**:

$$\begin{aligned} \log\left(q^*(Z)\right) &= E_\Theta[\log(p(\Theta, X, S, Z))] + \mathrm{A} \\ &= E_\Theta[p(X \mid Z, M, T) p(S \mid \Lambda, Z) p(Z \mid \pi) p(\Theta) + \mathrm{A} \\ &= E_\Theta\left[\log\left(\prod_{n=1}^{N}\prod_{k=1}^{K}\left(p\left(X_n \mid \boldsymbol{\mu}_k, \boldsymbol{\tau}_k\right) p\left(S_n \mid \lambda_k\right) p\left(Z_{n,k}=1 \mid \pi_k\right)\right)^{Z_{n,k}} p(\Theta)\right)\right] + \mathrm{A} \end{aligned}$$

   The term $p(\Theta) = p(M, T, \Lambda, \pi)$ does not depend on Z so it is absorbed into the constant. Now using the properties of the logarithms we obtain:

$$\log(q^*(Z)) = E_\Theta[\sum_{n=1}^{N}\sum_{k=1}^{K} Z_{n,k}\left(\log\left(p\left(X_n \mid \boldsymbol{\mu}_k, \boldsymbol{\tau}_k\right)\right) + \log\left(p\left(S_n \mid \lambda_k\right)\right) + \log\left(p\left(Z_{n,k}=1 \mid \pi_k\right)\right)\right)] + \mathrm{A}$$

We now develop further the know probability densities:

$$\log\left(\mathcal{N}\left(X_n \mid \boldsymbol{\mu}_k, (\boldsymbol{\tau}_k)^{-1}\right)\right) = -\frac{1}{2}\left(K(\log(2\pi)) + \log\left(\left|\boldsymbol{\tau}_k^{-1}\right|\right)\right) + \frac{1}{2}\left(X_n - \boldsymbol{\mu}_k\right)^T\left(\boldsymbol{\tau}_k^{-1} I\right)^{-1}\left(X_n - \boldsymbol{\mu}_k\right)$$
$$\log\left(p\left(S_n \mid \lambda_k\right)\right) = S_n \log\left(\lambda_k\right) - \lambda_k - \log\left(S_n!\right)$$
$$\log\left(p\left(Z_{n,k}=1 \mid \pi_k\right)\right) = \log\left(\pi_k\right)$$

We input this distributions in the expression and simplify:

$$\begin{aligned}\log\left(q^*(Z)\right) = \sum_{n=1}^{N}\sum_{k=1}^{K} Z_{n,k}\Bigg(&-\frac{1}{2}\left(K(\log(2\pi)) + E_\tau\left[\log\left(\left|\boldsymbol{\tau}_k^{-1}\right|\right)\right]\right) \\ &+ E_{\mu,\tau}\left[\frac{1}{2}\left(X_n - \boldsymbol{\mu}_k\right)^T\left(\boldsymbol{\tau}_k^{-1} I\right)^{-1}\left(X_n - \boldsymbol{\mu}_k\right)\right] + \\ &+ S_n E_\lambda\left[\log\left(\lambda_k\right)\right] - E_\lambda\left[\lambda_k\right] - \log\left(S_n!\right) + E_\pi\left[\log\left(\pi_k\right)\right]\Bigg) + \mathrm{A}\end{aligned}$$

Now, we use an auxiliary variable ρ:

$$\log\left(q^*(Z)\right) = \sum_{n=1}^{N}\sum_{k=1}^{K} Z_{n,k}\log\left(\rho_{n,k}\right) + \mathrm{A} = \log\left(\prod_{n=1}^{N}\prod_{k=1}^{K}\rho_{n,k}^{Z_{n,k}}\right) + \mathrm{A}$$

We remove the log from both sides:

$$q^*(Z) = \prod_{n=1}^{N}\prod_{k=1}^{K}\rho_{n,k}^{Z_{n,k}} + \ \mathrm{A} \ \propto \prod_{n=1}^{N}\prod_{k=1}^{K}\rho_{n,k}^{Z_{n,k}}$$

However, there is one last step to take. Normalization is needed since for a fixed value of n $Z_{n,k}$ must add up to 1.

$$q^*(Z) = \prod_{n=1}^{N}\prod_{k=1}^{K} r_{n,k}^{Z_{n,k}} \quad \text{where} \quad r_{n,k} = \frac{\rho_{n,k}}{\sum_{k=1}^{K}\rho_{n,k}}$$

Finally we obtain for $q(Z)$ that $E\left[Z_{n,k}\right] = r_{n,k}$. Also, we define for convenience $N_k = \sum_{n=1}^{N} r_{n,k}$

2. **Update equations for Θ**:
In this part we have to compute $q^*(\Theta)$ by using a similar approach as before, however, now the expectation will be taken over Z. Therefore the part corresponding to $p(\Theta)$ can not be ignored now.

$$\begin{aligned}\log\left(q^*(\Theta)\right) &= E_Z[\log(p(\Theta, X, S, Z))] + \mathrm{A} \\ &= E_Z[\log(p(X, S, Z \mid \pi, M, T, \Lambda)) + \log(p(\pi)) + \log(p(M, T)) + \log(p(\Lambda))] + \mathrm{A} \\ &= \sum_{n=1}^{N}\sum_{k=1}^{K} E_Z\left[Z_{n,k}\right]\log(Normal(X_n \mid \mu_k, \tau_k)) + \sum_{n=1}^{N}\sum_{k=1}^{K} E_Z\left[Z_{n,k}\right]\log(Poisson(S_n \mid \lambda_k)) \\ &+ \sum_{n=1}^{N}\sum_{k=1}^{K} E_Z\left[Z_{n,k}\right]\log(\pi_k) + \sum_{k=1}^{K}\log(p(\mu_k, \tau_k)) + \sum_{k=1}^{K}\log(p(\lambda_k)) + \sum_{k=1}^{K}(\alpha_k - 1)\log(\pi_k) + \mathrm{A}\end{aligned}$$

We can see how the decomposition is expressed with terms containing π, λ and the combination of μ and τ which results in the following factorization:

$$q(\Theta) = q(\Pi)\prod_{k=1}^{K} q(M, T, \Lambda) = q(\Pi)\prod_{k=1}^{K} q(M \mid T)q(T)q(\Lambda)$$

(a) **Update for π**:
We have to find in $q(\Theta)$ the terms dependent on π:

$$\ln q^*(\pi) = (\alpha - 1)\sum_{k=1}^{K} \log \pi_k + \sum_{k=1}^{K}\sum_{n=1}^{N} r_{nk} \log \pi_k + \mathrm{A}$$

By removing the log on both sides we find a $Dir(\pi \mid \alpha^*)$:

$$q^*(\pi) = Dir\left(\pi \mid \alpha^*\right) \text{ where } \alpha^* \text{ has components } \alpha_k^* = \alpha_k + N_k$$

(b) **Update for $\mu_\mathbf{k}$ and $\tau_\mathbf{k}$**: Now, likewise, we find in $q(\Theta)$ the terms dependent on μ_k and τ_k:

$$\ln q^*\left(\mu_k, \tau_k\right) = \sum_{n=1}^{N} E_Z\left[Z_{n,k}\right] \log(Normal(X_n \mid \mu_k, \tau_k)) + \log(p(\mu_k, \tau_k))$$

$$= \sum_{n=1}^{N} r_{nk}\left(\frac{1}{2}\log\tau_k - \frac{1}{2}\tau_k\left(X_n - \mu_k\right)^2\right) + \frac{1}{2}\log\tau_k - \frac{1}{2}C\left(\mu_k - \mu\right)^2 + (\alpha' - 1)\log\tau_k - \beta'\tau_k$$

By removing log from both sides we find that:

$$q^*\left(\mu_k \mid \tau_k\right) = \mathcal{N}\left(\mu_k \mid \mu^*, \tau^*\right) \text{ where } \tau^* = (C + N_k)\,\tau_k \text{ and } \mu^* = \frac{\left(\tau_k \sum_{n=1}^{N} r_{nk} X_n\right) + C\tau_k\mu}{\tau^*}$$

and:

$$q^*\left(\tau_k\right) = \mathrm{Gamma}\left(\tau_k \mid \alpha'^*, \beta'^*\right) \text{ where } \alpha'^* = \alpha' + N_k \text{ and } \beta'^* = \beta' + \frac{1}{2}C\mu^2 + \frac{1}{2}\sum_{n=1}^{N} r_{nk} X_n^2$$

(c) **Update for $\lambda_\mathbf{k}$**: We do the same procedure as before and find the terms dependent on λ_k

$$\ln q^*\left(\lambda_k\right) = \sum_{n=1}^{N} E_Z\left[Z_{n,k}\right] \log(Poisson(S_n \mid \lambda_k)) + \log(p(\lambda_k))$$

$$= \sum_{n=1}^{N} \left(r_{nk}\left(S_n \ln\lambda_k - \lambda_k - \ln\left(S_n!\right)\right)\right) + (\alpha_0 - 1)\ln\lambda_k - \beta_0\lambda_k$$

Again, removing the log from both sides we find the underlying distribution:

$$q^*\left(\lambda_k\right) = \mathrm{Gam}\left(\lambda_k \mid \alpha_0^*, \beta_0^*\right) \text{ where } \alpha_0^* = \alpha_0 + \sum_{n=1}^{N} r_{nk} S_n \text{ and } \beta_0^* = \beta + N_k$$

3.5 The Casino Model and Sampling Tables Given Dice Sums

Consider the following generative model. There are $2K$ tables in a casino, $t_1, \ldots, t_K, t'_1, \ldots, t'_K$ of which each is equipped with a single dice (which may be biased, i.e., any categorical distribution

on $\{1,\ldots,6\}$) and N players $P_1,\ldots,P_N$ of which each is equipped with a single dice (which also may be biased, i.e., any categorical distribution on $\{1,\ldots,6\}$). Each player P_i visits K tables. In the k :th step, if the previous table visited was t_{k-1}, the player visits t_k with probability $1/4$ and t'_k with probability $3/4$, and if the previous table visited was t'_{k-1}, the player visits t'_k with probability $1/4$ and t_k with probability $3/4$. So, in each step the probability of staying among the primed or unprimed tables is $1/4$. At table k player i throws her own dice as well as the table's dice. We then observe the sum S^i_k of the two dice, while the outcome of the table's dice X_k and the player's dice Z_k are hidden variables. So for player i, we observe $S^i = S^i_1,\ldots,S^i_K$, and the overall observation for N players is $S^1,\ldots,S^N$.

Question 3.5.6

Provide a drawing of the Casino model as a graphical model. It should have a variable indicating the table visited in the $k-th$ step, variables for all the dice outcomes, variables for the sums, and plate notation should be used to clarify that N players are involved.

From the statement we define:

1. $\mathbf{X_{n,k}} \sim Cat(\beta_k)$
2. $\mathbf{Z_{n,k}} \sim Cat(\alpha_n)$

Also, for convenience, $\mathbf{Y_{n,k}}$ which determines the type of table player n is at step k. $\mathbf{Y_{n,k}} = 0$ if table k is t_k, whereas, $\mathbf{Y_{n,k}} = 1$ if table k is t'_k

$Y_{n,k}/Y_{n,k+1}$	0	1
0	1/4	3/4
1	3/4	1/4

In figure 2, the graphical model for the casino model can be seen where the grayed out variables are the observable variables.

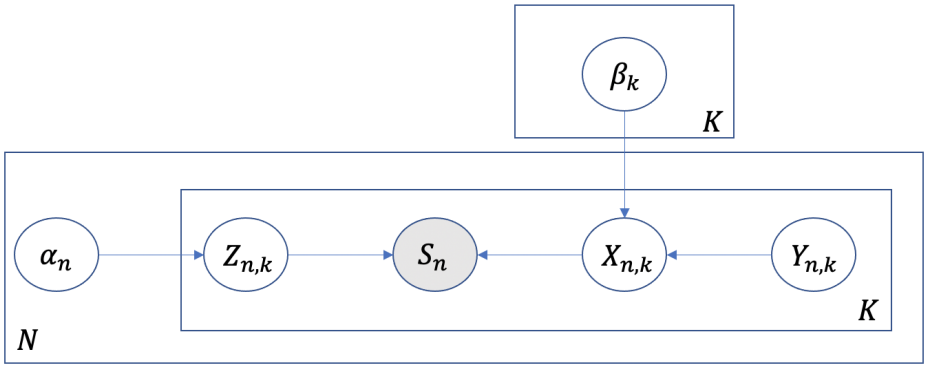


Figure 2: Casino graphical model.

Question 3.5.7

Implement the casino model.

See code in Appendix 2: 3.5.8 Code and Appendix 1: Casino Code.

Question 3.5.8

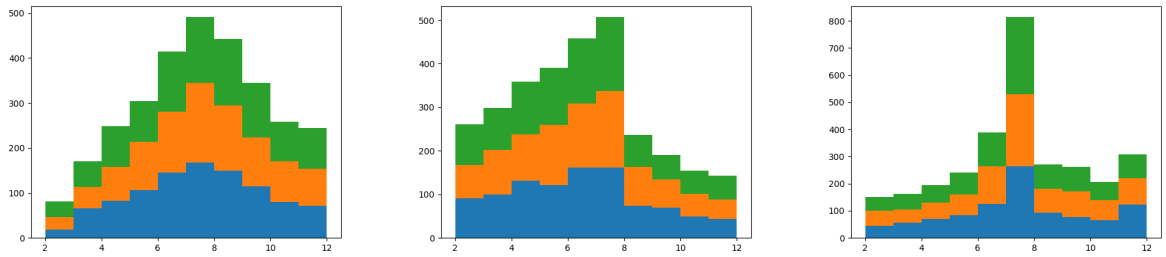
Provide data generated using at least three different sets of categorical dice distributions – what does it look like for all unbiased dice, i.e., uniform distributions, for example, or if some are biased in the same way, or if some are unbiased and there are two different groups of biased dice

Here we present the data (S_k) obtained from 1000 players playing a casino of 3 tables (each one mapping to a color). The initial state distribution (prime or not) of the tables is a uniform distribution $\pi = [0.5, 0.5]^T$.

In figure 3a we can see the case where all the dice follow the a uniform distribution for their 6 output values. As expected, since we are plotting the sum between the table's dice and the player's dice, we get a distribution of S_k that peaks in the middle since the outcomes on that area have available more combinations of outcomes. More precisely, 7 is the most likely S_k with 6 combinations of outcomes that lead to it, $p(S_k = 7) = \frac{6}{36}$.

In figure 3b, we can see the case where the dice being used at the tables are fair but the players are using a dice that is biased with $p(Z_{n,k} = 1) = 1/2$ and $p(Z_{n,k} \neq 1) = 1/10$. As it can be seen, this makes the lowest half of the outcomes the most probable since obtaining values above 8 is hard when the player's outcome is equal to 1 half of the times.

Finally, in figure 3c, we can see the case where the table dices are biased towards 6 with $p(Z_{n,k} = 6) = 1/2$ and $p(Z_{n,k} \neq 6) = 1/10$ and the player's dice is biased as before. As expected, since half of the times 6 is the outcome of the dice rolled in the table and 1 is the outcome of the dice rolled by the player, this results in 7 being the most likely S_k, by far.



(a) All dice unbiased (b) Player's dice biased. (c) All dice biased.

Figure 3: Casino model after N players played with different dice settings

Question 3.5.9

Describe an algorithm that, given (1) the parameters Θ of the full casino model of Task 2.2 (so, Θ is all the categorical distributions corresponding to all the dice), (2) a sequence of tables $r_1 \ldots, r_K$ (that is, r_i is t_i or t'_i), and (3) an observation of dice sums $s_1, \ldots, s_K$, outputs $p(r_1, \ldots, r_K \mid s_1, \ldots, s_K, \Theta)$.

We will denote $r_1, ..., r_k$ and $s_1, ..., s_k$ as $R_{1:K}$ and $S_{1:K}$. Therefore the expression our algorithm needs to compute is: $p(R_{1:K} \mid S_{1:K}, \Theta)$. If we employ the Bayes rule we can write out the expression of this probability to be:

$$p(R_{1:K} \mid S_{1:K}, \Theta) = \frac{p(S_{1:K} \mid R_{1:K}) p(R_{1:K} \mid \Theta)}{p(S_{1:K} \mid \Theta)}$$

Now, if we look individually at each term in the expression:

- From the graphical model we know that $S_k \perp S_{k+1} \mid R_k$ therefore we can decompose the first term into:

$$\begin{aligned} p(R_{1:K} \mid S_{1:K}, \Theta) &= \prod_{k=1}^{K} p\left(s_k \mid R_k, \Theta\right) \\ &= \prod_{k=1}^{K} \sum_{z}^{6} \sum_{x}^{6} p\left(X_k = x \mid R_k, \Theta\right) p\left(Z_k = z \mid \Theta\right) I\left(x + z = s_k\right) \\ &= \prod_{k=1}^{K} \sum_{x}^{6} \sum_{z}^{6} \operatorname{Cat}\left(x \mid \beta_k\right) \operatorname{Cat}(z \mid \alpha) I\left(x + z = s_k\right) \\ &= \prod_{k=1}^{K} \sum_{x}^{6} \sum_{z}^{6} \beta_{k,x} \alpha_z I\left(x + z = s_k\right) \end{aligned}$$

- Now, we can expand the next expression into another factorization by the application of the first order Markov Property:

$$p\left(R_{1:K} \mid \Theta\right) = p(R_1 = r_1) \prod_{k=2}^{K} p\left(R_k = r_k \mid R_{k-1} = r_{k-1}, \Theta\right) = \pi_{r_1} A_{r_{k-1} r_k}$$

  where we call A the transition matrix and π defines the distribution of the tables initially.

- Lastly, we look at the denominator, where we find a recursive function.

$$p\left(S_{1:K} \mid \Theta\right) = \sum_{m} p\left(S_{1:K}, R_K = m \mid \Theta\right) = \sum_{m} \alpha_K(m)$$

  where α_K is shown to be recursive:

$$\begin{aligned} &\alpha_K(m) = p\left(S_{1:K}, r_K = m \mid \Theta\right) \\ &= p\left(s_K, S_{1:K-1}, r_K = m \mid \Theta\right) = p\left(s_K \mid r_K = m, s_{1:K-1}, \Theta\right) p\left(r_k = m, S_{1:K-1} \mid \Theta\right) \\ &= p\left(s_K \mid r_K = m, \Theta\right) \sum_{j} p\left(r_K = m, r_{K-1} = j, S_{1:K-1} \mid \Theta\right) \\ &= p\left(s_K \mid r_K = m, \Theta\right) \sum_{j} p\left(r_K = m \mid r_{K-1} = j, S_{1:K-1}, \Theta\right) p\left(r_{K-1} = j, S_{1:K-1} \mid \Theta\right) \\ &= p\left(s_K \mid r_K = m, \Theta\right) \sum_{j} p\left(r_K = m \mid r_{K-1} = j\right) \alpha_{K-1}(j) \end{aligned}$$

Finally, we arrive to a dynamic programming algorithm as the following:

1. Initialization of α with the initial state distribution of the tables:

$$\alpha_1(m) = \pi_m p(s_1 \mid r_1 = m)$$

2. for $k = 2$ until $k = K$:

$$\alpha_k(m) = p\left(s_k \mid r_k = m, \Theta\right) \sum_{j} p\left(r_k = m \mid r_{k-1} = j\right) \alpha_{k-1}(j)$$

   where $p\left(s_k \mid r_k = l, \theta\right) = \sum_{x,z}^{6} I\left(x + z = s_k\right) \beta_{k,x} \alpha_z$

Question 3.5.10

You should also show how to sample $r_1, \ldots, r_K$ from $p(R_1, \ldots, R_K \mid s_1, \ldots, s_K, \Theta)$ as well as implement and show test runs of this algorithm. In order to design this algorithm show first how to sample r_K from

$$p(R_K \mid s_1, \ldots, s_K, \Theta) = p(R_K, s_1, \ldots, s_K \mid \Theta) / p(s_1, \ldots, s_K \mid \Theta)$$

and then r_{K-1} from

$$p(R_{K-1} \mid r_K, s_1, \ldots, s_K, \Theta) = p(R_{K-1}, r_K, s_1, \ldots, s_K \mid \Theta) / p(r_K, s_1, \ldots, s_K \mid \Theta)$$

First of all, as in the previous section, we rewrite the expression into something we can compute by the application of Bayes rule:

$$p(R_{1:K} \mid S_{1:K}, \Theta) = \frac{p(R_{1:K}, S_{1:K} \mid \Theta)}{p(S_{1:K} \mid \Theta)} = p(R_K = r_K \mid S_{1:K}, \Theta) p(R_{K-1} = r_{K-1} \mid R_K = r_k, S_{1:K}, \Theta)$$

Now we do the same with the other expressions:

1. In this case we identify the previously defined expression for α_K in the numerator:

$$p(R_K = r_K \mid S_{1:K}, \Theta) = \frac{p(R_K = r_k, S_{1:K} \mid \Theta)}{p(S_{1:K} \mid \Theta)} = \frac{\alpha_K(r_K)}{\sum_m \alpha_K(m)}$$

2. And now, the expression for R_{K-1}:

$$\begin{aligned} p(R_{K-1} = r_{K-1} \mid R_K = r_k, S_{1:K}, \Theta) &= \frac{p(R_{K-1} = r_{k-1}, R_K = r_k, S_{1:K} \mid \Theta)}{p(R_k = r_k, S_{1:K} \mid \Theta)} \\ &= \frac{p(R_{K-1} = r_{k-1}, S_{1:K} \mid \Theta) \, p(R_K = r_k \mid R_{K-1} = r_{k-1}, \Theta)}{p(R_k = r_k, S_{1:K} \mid \Theta)} \\ &= \frac{\alpha_{K-1}(r_{k-1}) \, A_{r_K R_{K-1}}}{\sum_{k'} \alpha_{K-1}(r_{k'}) \, A_{r_K r_{k'}}} \end{aligned}$$

Where A is the transition matrix and where we sample backwards from the Bernoulli distribution that we get from $p(R_{K-1} = r_{K-1} \mid R_K = r_k, S_{1:K}, \Theta)$ and then we just sample from $K-2$ backwards until 1.

Now for test runs we have one player play in 6 tables in the casino model. The first table in the sequence has equal probability of being primed or not primed. The player uses an unbiased dice. The prime tables use a biased dice with the following distribution: $p(X_k = 1) = 1$ and $p(X_k \neq 1) = 0$. The not prime tables use a fair dice.

As we can observe in the table below, the observed $s_k \leq 7$ are mostly related with the prime tables using a biased dice towards 1 which makes the expected value of S_k drop significantly. Remember that $Y_k = 1$ means that table k is prime. Therefore, we see how the conditions imposed on the bias of the dice used by the prime tables is respected by our sampled posterior in most of the cases.

See code in Appendix 3: 3.5.10 Code and Appendix 1: Casino Code.

Observations	True Posterior (Y_k)	Sampled posterior ($\hat{Y}_k$)
6, 7, 6, 4, 7, 4	0, 1, 0, 1, 0, 1	0, 0, 1, 1, 0, 1
10, 3, 6, 2, 2, 11	0, 0, 0, 1, 1, 0	1, 1, 0, 0, 1, 0
3, 8, 2, 6, 3, 5	1, 0, 1, 0, 0, 1	1, 0, 1, 0, 1, 0
5, 7, 3, 6, 3, 7	1, 0, 1, 0, 1, 0	0, 1, 0, 0, 1, 0

Table 1: Sampled Y_k from the posterior distribution of the casino model.

3.6 The Casino Model - Expectation-Maximization

Consider the following simplification of the casino model from Problem 3.5 . There are K tables in the casino $t_1, \ldots, t_K$ of which each is equipped with a single dice (which may be biased, i.e., any categorical distribution on $\{1, \ldots, 6\}$) and N players $P_1, \ldots, P_N$ of which each is equipped with a single dice (which also may be biased, i.e., any categorical distribution on $\{1, \ldots, 6\}$). Let Θ be the parameters of all these categorical distributions. Each player P_i visits the K tables in the order $1, \ldots, K$. At table k the player i throws her own dice as well as the table's dice. We then observe the sum S_k^i of the dice, while the outcome of the table's dice X_k and the player's dice Z_k are hidden variables. So for player i, we observe $s^i = s_1^i, \ldots, s_K^i$, and the overall observation for N players is $s^1, \ldots, s^N$. Design and describe an EM algorithm for this model. That is, an EM algorithm that given $s^1, \ldots, s^N$ finds locally optimal parameters for the categorical distributions (i.e., the dice), that is, the Θ maximising $P(s_1^i, \ldots, s_K^i \mid \Theta)$.

Question 3.6.11

Present the algorithm written down in a formal manner (using both text and mathematical notation, but not pseudo code).

We assume the dice from the K tables, $d_k \sim Cat(\beta_k)$ and that the N players' dices, $p_n \sim Cat(\alpha_n)$. We have to derive an EM algorithm that finds the $\Theta = [\beta_k, \alpha_n]$ which maximizes $P(s_1^i, \ldots, s_K^i \mid \Theta)$. Since we are deriving an Expectation-Maximization algorithm, our goal is to obtain update equations for the Expectation (E) step and for the Maximization (M) step.

If we are given the complete dataset $D = \{(X_{1:K,1:N}, Z_{1:K,1:N}, S_{1:K,1:N})\}$ then we can compute the complete likelihood on the data. Since X and Z are independent then:

$$\begin{aligned} p(D \mid \Theta) &= \prod_{n=1}^{N} p\left(x_{n,1:K}, z_{n,1:K}, s_{n,1:K} \mid \Theta\right) \\ &= \prod_{n=1}^{N} p\left(s_{n,1:K} \mid x_{n,1:K}, z_{n,1:K}, \Theta\right) p\left(x_{n,1:K} \mid \Theta\right) p\left(z_{n,1:K} \mid \Theta\right) \\ &= \prod_{n,k=1}^{N,K} p\left(s_{nk} \mid x_{nk}, z_{nk}, \Theta\right) p\left(x_{nk} \mid \Theta\right) p\left(z_{nk} \mid \Theta\right) \end{aligned}$$

We know $p\left(s_{nk} \mid x_{nk}, z_{nk}, \Theta\right)$ is binary; 1 or 0. It is only 1 when $s_{nk} = x_{nk} + z_{nk}$, otherwise, it is 0. We will employ the indicator function to denote this.

$$\begin{aligned} p(D \mid \Theta) &= \prod_{n,k=1}^{N,K} I(s_{nk} = x_{nk} + z_{nk}) p\left(x_{nk} \mid \Theta\right) p\left(z_{nk} \mid \Theta\right) = \prod_{n,k=1}^{N,K} p\left(x_{nk} \mid \Theta\right) p\left(z_{nk} \mid \Theta\right) \\ &= \prod_{n,k=1}^{N,K} \prod_{m}^{6} \prod_{t}^{6} [p\left(x_{nk} = m \mid \Theta\right) p\left(z_{nk} = t \mid \Theta\right)]^{I(x_{nk}=m, z_{nk}=t)} \end{aligned}$$

Resulting in a log-likelihood of:

$$\log(p(D \mid \Theta)) = \sum_{n,k=1}^{N,K} \sum_{m,t=1}^{6} I(x_{nk} = m, z_{nk} = t)[\log(p\left(x_{nk} = m \mid \Theta\right)) + \log(p\left(z_{nk} = t \mid \Theta\right))]$$

Now, if are given an incomplete dataset $D = \{S_{1:N,1:K}\}$ where we can only observe the output of the sum of table's and player's dice then we can compute the expected log-likelihood. We introduce the following variable renaming for convenience:
$\Pi_k^m = p(x_{nk} = m \mid \Theta^*)$, $\Phi_n^t = p(z_{nk} = t \mid \Theta^*)$ and $\Lambda_{n,k}^{m,t} = p(x_{nk} = m, z_{nk} = t \mid s_{nk}, \Theta)$.

$$\begin{aligned}
&\sum_{n=1}^{N} E_{p\left(x_{n,1:K}, z_{n,1:K} \mid s_{n,1:K}, \theta\right)}[\log(p(D \mid \Theta^*))]) = \\
&\sum_{n,k=1}^{N,K} \sum_{m,t=1}^{6} \Lambda_{n,k}^{m,t} \log(\Pi_k^m) + \sum_{n,k=1}^{N,K} \sum_{m,t=1}^{6} \Lambda_{n,k}^{m,t} \log(\Phi_n^t) \\
&= \sum_{k,m=1}^{K,6} [\sum_{n,t=1}^{N,6} \Lambda_{n,k}^{m,t}] \log(\Pi_k^m) + \sum_{n,t=1}^{N,6} [\sum_{k,m=1}^{K,6} \Lambda_{n,k}^{m,t}] \log(\Phi_n^t)
\end{aligned}$$

Therefore, for the E-step, responsibilities calculation:

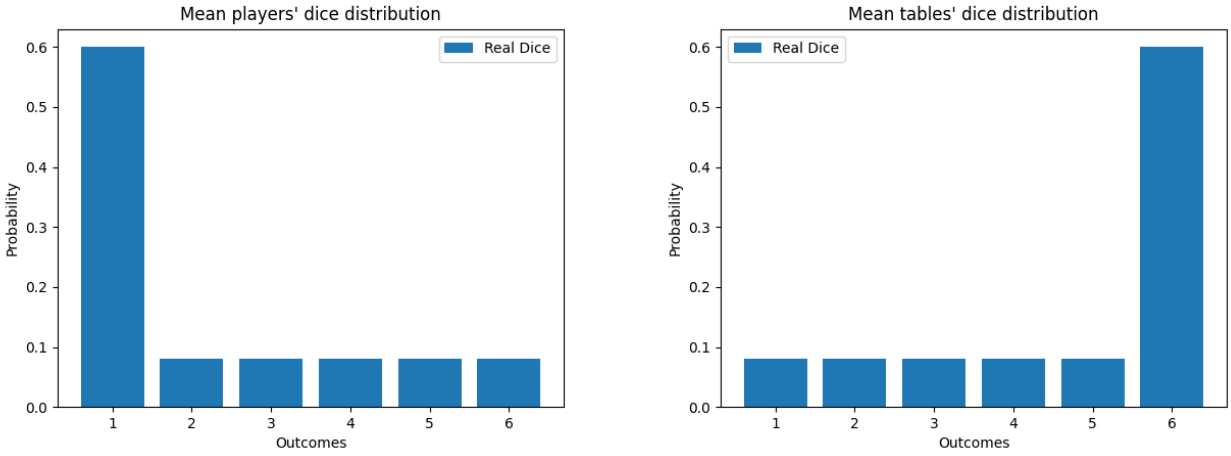
$$\begin{aligned}
p(x_{nk} = m, z_{nk} = t \mid s_{nk}, \Theta) &= \Lambda_{n,k}^{m,t} \\
&= \frac{p(x_{nk} = m, z_{nk} = t \mid s_{nk}, \Theta) p(x_{nk}, z_{nk} \mid \Theta)}{p(s_{nk} \mid \Theta)} \\
&= \frac{p(x_{nk} = m, z_{nk} = t \mid s_{nk}, \Theta) p(x_{nk} \mid \Theta) p(z_{nk} \mid \Theta)}{\sum_i^6 \sum_j^6 p(s_{nk} \mid x_{nk} = i, z_{nk} = j, \Theta) p(x_{nk} \mid \Theta) p(z_{nk} \mid \Theta)} \\
&= \frac{I(m + t = s_{nk}) \Pi_k^m \Phi_n^t}{\sum_i^6 \sum_j^6 I(i + j = s_{nk}) \Pi_k^i \Phi_n^j}
\end{aligned}$$

Finally for the M-step: $(\Pi_k^m)^* = \frac{\sum_{n,t}^{N,6} \Lambda_{n,k}^{m,t}}{\sum_{m,n,t}^{6,N,6} \Lambda_{n,k}^{m,t}}$ and $(\Phi_n^t)^* = \frac{\sum_{k,m}^{K,6} \Lambda_{n,k}^{m,t}}{\sum_{t,k,t}^{6,K,6} \Lambda_{n,k}^{m,t}}$

Question 3.6.12

Implement it and test the implementation with data generated in Task 3.5, and provide graphs or tables of the results of testing it with the data.

We use data generated from Task 3.5 in a setting where 3 players play in 5 tables inside the casino. For simplicity when showing results and for better interpretability we will use only two underlying categorical distributions, one for the players' dice and other for the tables' dice. The players will use a dice heavily biased towards 1 and the tables use a dice heavily biased towards 6 as we can see in figure 4.



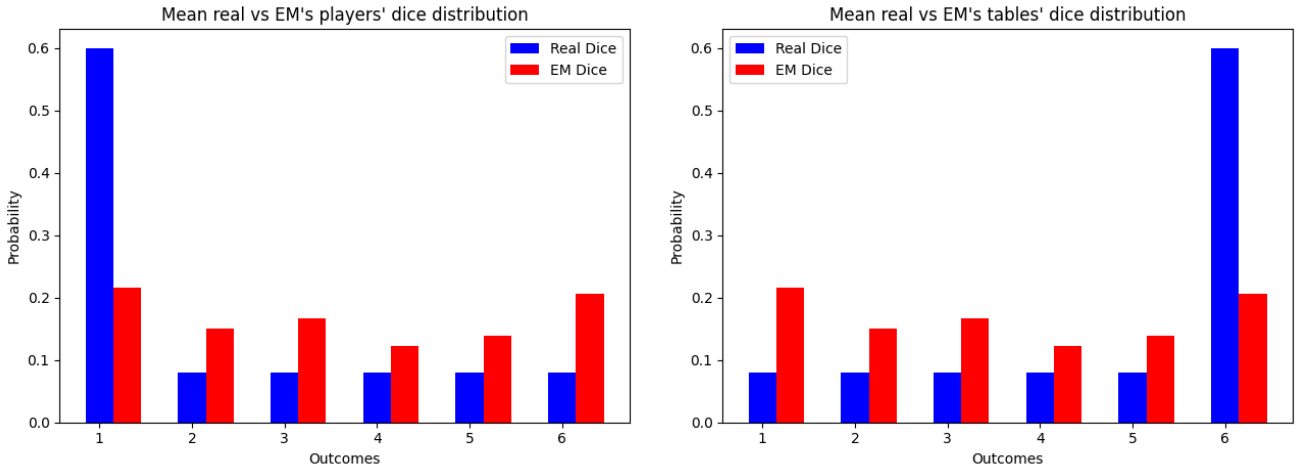
(a) Players' dice distribution

(b) Tables' dice distribution

Figure 4: Real underlying distribution of the dices in the casino

This distribution of dice lead to a sequence of sums of table plus player dice outcomes which is fed into our EM model. This EM model is initialized in two different ways. We will compare how good the EM model fits the real distribution between both initializations. In order to do so we will compare the tables and players dice distribution mean for each output that the EM converges to with the real underlying distribution:

1. **Initialization as uniformly distributed dices:** The EM distributions are initialized in a way where outcomes both types of dice have probability 1/6. After reaching convergence the distributions obtained are very different from the real ones as we can see in figure 5, this can be due to the fact that the EM algorithm is only observing the output of the sum and not the summands. Also, not many tables or players where employed due to computation constraints which leads worse data for the EM to use.



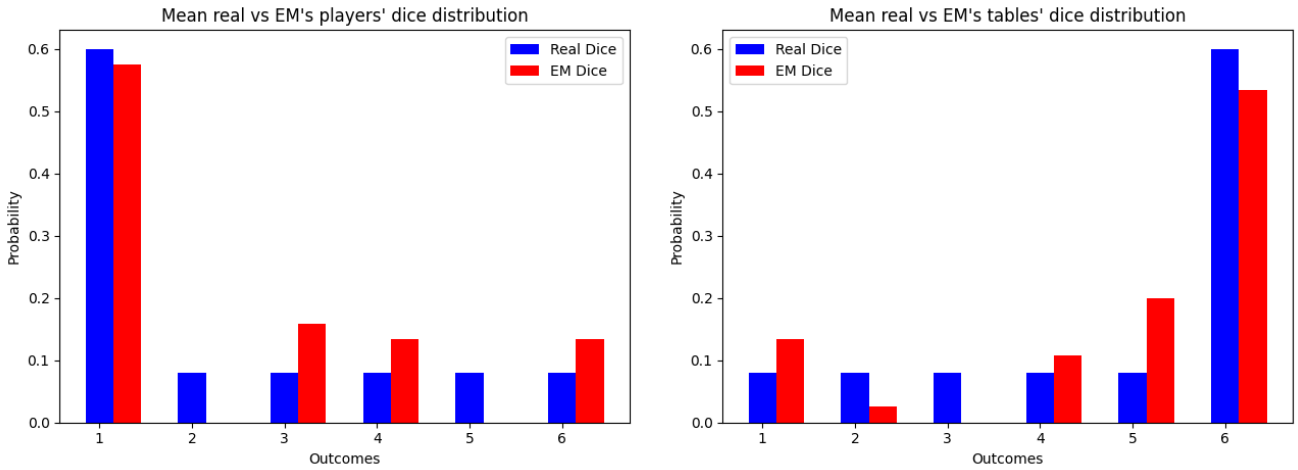
(a) Players' dice distribution

(b) Tables' dice distribution

Figure 5: Distribution of the dices in the casino obtained by the EM model with a uniform initialization

2. **Initialization with a biased distribution:** Now the initial guesses of the distributions of the dice where extremely biased. Players' dice distribution were biased $p(Z_k = 1) = 0.55$ and $p(Z_k = 6) = 0.09$ and the dice distributions whre also biased in the opposite way so: $p(X_k = 6) = 0.55$ and $p(X_k = 1) = 0.09$. This time after convergence we can see in figure

6 how even though the distributions outputted by the EM model are not perfect, they do follow the same tendency as the real ones. We can clearly see how the EM fit of the data shows outcome 1 as the most likely compared to the rest for the players and outcome 6 as the most likely for the dices used in the tables. **This is a clear example of how one can improve the results of an EM model by a good initialization of the parameters, i.e, good domain knowledge or insights about data.**



(a) Players' dice distribution

(b) Tables' dice distribution

Figure 6: Distribution of the dices in the casino obtained by the EM model with a biased initialization

See code in Appendix 1: Casino Code, Appendix 4: EM Code, Appendix 5: 3.6.12 Code.

List of Figures

References

[1] S. Dasgupta and A. Gupta, "An elementary proof of a theorem of johnson and lindenstrauss," *Random Struct. Algorithms*, vol. 22, no. 1, pp. 60–65, Jan. 2003. DOI: 10.1002/rsa.10073. [Online]. Available: https://doi.org/10.1002/rsa.10073.

[2] W. L. Hamilton, R. Ying, and J. Leskovec, *Representation learning on graphs: Methods and applications*, cite arxiv:1709.05584Comment: Published in the IEEE Data Engineering Bulletin, September 2017; version with minor corrections, 2017. [Online]. Available: http://arxiv.org/abs/1709.05584.

[3] C. M. Bishop, *Pattern recognition and machine learning*. Springer, 2016.

Appendix 1: Casino Code

```
import numpy as np
import matplotlib.pyplot as plt



class Casino:
    def __init__(self, nTables, nPlayers, tableProb, primeTableDice, notPrimeT
        self.nTables = nTables
        self.nPlayers = nPlayers
        self.primeTableDice = primeTableDice
        self.notPrimeTableDice = notPrimeTableDice
        self.playerDice = playerDice
        self.tableProb = tableProb
        self.tables = self._init_tables()

    def _init_tables(self):
        tables = []  # Y_k
        firstTable = np.random.binomial(1, self.tableProb['init'], 1)[0]
        tables.append(firstTable)

        for k in range(1, self.nTables):
            if tables[k-1] == 0:  # Y_{k-1} == 0 (not prime)
                t = np.random.binomial(1, self.tableProb['change'], 1)
            else:  # Y_{k-1} == 1 (prime)
                t = np.random.binomial(1, 1-self.tableProb['change'], 1)
            tables.append(t[0])
        return tables

    def _throw_dice(self, dice):
        outcomes = np.random.multinomial(1, dice)
        return np.argmax(outcomes) + 1

    def run(self):
        S_n = []  # S_{k,n} sum of the outcomes from X_{k,n} and Z_{k,n}
        for p in range(self.nPlayers):
            S_k = []
            for k, table in enumerate(self.tables):
                if table == 0:  # Y_k == 0 (not prime table)
                    X_nk = self._throw_dice(self.notPrimeTableDice[k])
                else:  # Y_k == 1 (prime table)
                    X_nk = self._throw_dice(self.primeTableDice[k])

                Z_nk = self._throw_dice(self.playerDice[p])
                S_k.append(X_nk + Z_nk)
            S_n.append(np.asarray(S_k))

        self.S = np.asarray(S_n)
```

Appendix 2: 3.5.8 Code

```
from ..casino import Casino
import matplotlib.pyplot as plt
import numpy as np

def main():
    nTables = 3
    nPlayers = 1000
    tableProb = {
        # 50% chance of being prime and same of not being prime table_1
        'init': 1/2,
        'change': 3/4
    }

    # Case 1:
    # All dice unbiased

    primedTableDice = np.ones((nTables, 6)) * (1/6)
    notPrimedTableDice = np.ones((nTables, 6)) * (1/6)
    playerDice = np.ones((nPlayers, 6)) * (1/6)  # one dice per player

    casino = Casino(nTables, nPlayers, tableProb,
                    primedTableDice, notPrimedTableDice, playerDice)
    casino.run()

    plt.figure()
    plt.hist(casino.S, stacked=True)
    plt.savefig('3_5_8/unbiased.png')

    # Case 2:
    # Player dice biased towards 1

    primedTableDice = np.ones((nTables, 6)) * (1/6)
    notPrimedTableDice = np.ones((nTables, 6)) * (1/6)
    playerDice = np.ones((nPlayers, 6)) * (1/6)  # one dice per player

    playerDice[:, 0] = 1/2
    playerDice[:, 1:5] = 1/10

    casino = Casino(nTables, nPlayers, tableProb,
                    primedTableDice, notPrimedTableDice, playerDice)
    casino.run()

    plt.figure()
    plt.hist(casino.S, stacked=True)
    plt.savefig('3_5_8/player-biased.png')

    # Case 3:
    # Prime and not prime tables' dice biased towards 1 player's  dice biased

    primedTableDice = np.ones((nTables, 6)) * (1/6)
    primedTableDice[:, 0] = 1/2
    primedTableDice[:, 1:5] = 1/10
```

```
    notPrimedTableDice = np.ones((nTables, 6)) * (1/6)
    notPrimedTableDice[:, 0] = 1/2
    notPrimedTableDice[:, 1:5] = 1/10

    playerDice = np.ones((nPlayers, 6)) * (1/6)  # one dice per player
    playerDice[:, 5] = 1/2
    playerDice[:, 0:4] = 1/10

    casino = Casino(nTables, nPlayers, tableProb,
                    primedTableDice, notPrimedTableDice, playerDice)
    casino.run()

    plt.figure()
    plt.hist(casino.S, stacked=True)
    plt.savefig('3_5_8/all-biased.png')


main()
```

Appendix 3: 3.5.10 Code

```
import numpy as np
import os
import sys
import inspect

currentdir = os.path.dirname(os.path.abspath(
    inspect.getfile(inspect.currentframe())))
parentdir = os.path.dirname(currentdir)
sys.path.insert(0, parentdir)
from casino import Casino


def compute_beta(observations, nTypeTables, casino):
    beta_matrix = np.zeros((len(observations), nTypeTables))
    playerDice = casino.playerDice[0]
    for k in range(len(observations)):
        for i in range(0, 6):
            for j in range(0, 6):
                if i+1 + j+1 == observations[k]:
                    beta_matrix[k][0] += playerDice[i] * casino.notPrimeTableD
                    beta_matrix[k][1] += playerDice[i] * casino.primeTableDice

    return beta_matrix


def compute_alpha(observations, casino, nTypeTables, A, beta_matrix):
    # table of alphas
    alpha = np.zeros((len(observations), nTypeTables))
    for i in range(nTypeTables):
        alpha[0][i] = casino.tableProb['init'] * beta_matrix[0][i]
```

```
    for k in range(1, len(observations)):
        for i in range(nTypeTables):
                alpha[k][i] += alpha[k-1][0] * A[0][i] * beta_matrix[k][i]
                alpha[k][i] += alpha[k-1][1] * A[1][i] * beta_matrix[k][i]
    return alpha


def posteriorSampling(obs, nTypeTables, alpha_matrix, A):
    tables = []
    sum = 0
    for type in range(nTypeTables):
        sum += alpha_matrix[len(obs)-1][type]
    tableProb = alpha_matrix[len(obs)-1][1]/sum
    lastTable = np.random.binomial(1, tableProb)

    tables.append(lastTable)

    for i in reversed(range(len(obs)-1)):
        prevTable = []
        sum = 0
        for beforeTable in range(nTypeTables):
            prevTable.append(A[beforeTable][lastTable]*alpha_matrix[i][beforeT
        for type in range(nTypeTables):
            sum += prevTable[type]
        tableProb = prevTable[1]/sum
        tables.append(np.random.binomial(1, tableProb))
        lastTable = tables[-1]

    # put in correct chronological order
    tables = tables[::-1]
    return tables


def main():
    nTables = 15
    nPlayers = 1
    tableProb = {
        # 50% chance of being prime and same of not being prime table_1
        'init': 1/2,
        'change': 3/4
    }
    A = [[1/4, 3/4], [3/4, 1/4]]
    nTypeTables = len(A[0])

    primedTableDice = np.ones((nTables, 6)) * (1/6)
    notPrimedTableDice = np.ones((nTables, 6)) * (1/6)
    playerDice = np.ones((nPlayers, 6)) * (1/6)  # one dice per player

    primedTableDice[:, 0] = 1
    primedTableDice[:, 1:5] = 0
```

```
    casino = Casino(nTables, nPlayers, tableProb,
                    primedTableDice, notPrimedTableDice, playerDice)
    casino.run()
    observations = casino.S[0]
    print(f"Observations: {observations}")
    print(f"Real posterior: {casino.tables}")

    # Sample sequence of table from the given observations
    beta_matrix = compute_beta(observations, nTypeTables, casino)
    alpha_matrix = compute_alpha(observations, casino, nTypeTables, A, beta_ma
    tableSequence = posteriorSampling(
        observations, nTypeTables, alpha_matrix, A)
    print(f"Sampled posterior: {tableSequence}")


main()
```

Appendix 4: EM Code

```
import numpy as np
import math


class CategoricalCategoricalEM:
    def __init__(self, pi_1_shape, pi_2_shape, threshold):
        # pi_1: first categorical dist, pi_2: second cat dist
        self.pi_1_shape = pi_1_shape
        self.pi_2_shape = pi_2_shape
        self.threshold = threshold

    def fit(self, observations):
        self.init_params()
        old_log_likelihood = 0
        iter = 0

        print('\tFitting Expectation Maximization...')
        while True:
            self.E_step(observations)
            self.M_step()
            new_log_likelihood = self.log_likelihood()
            if math.isnan(new_log_likelihood):
                break
            if abs(old_log_likelihood - new_log_likelihood) < self.threshold:
                break
            old_log_likelihood = new_log_likelihood
            iter = iter + 1

            print(new_log_likelihood)

        self.log_likelihood = new_log_likelihood
        print(f'\tIterations= {iter}')
```

```
def init_params(self):
    # initial guesses
    self.pi_1 = np.ones(self.pi_1_shape)*(1/6)  # dice from table
    self.pi_2 = np.ones(self.pi_2_shape)*(1/6)  # dice from players

    self.pi_1[:, 0:4] = 0.09
    self.pi_1[:, 5] = 0.55

    self.pi_2[:, 0] = 0.55
    self.pi_2[:, 1:] = 0.09

    print(
        f"Initial guesses:\n\tTable Dist:\n\t{self.pi_1}\n\tPlayer Dist:{s

def get_params(self):
    return self.pi_1, self.pi_2

def E_step(self, observations):
    # Calculate responsibilities
    K, N = self.pi_1.shape[0], self.pi_2.shape[0]
    n_cat = (self.pi_1.shape[1], self.pi_2.shape[1])

    self.r_nk = np.zeros((K, N, n_cat[0], n_cat[1]))

    den = np.zeros((K, N))
    num = np.zeros((K, N, n_cat[0], n_cat[1]))

    for k in range(K):
        for n in range(N):
            for t in range(n_cat[0]):
                for m in range(n_cat[1]):
                    s = observations[n][k]
                    if (m+1 + t+1 == s):
                        num[k][n][t][m] = self.pi_1[k][m]*self.pi_2[n][t]
                        den[k][n] += self.pi_1[k][m]*self.pi_2[n][t]
    for k in range(K):
        for n in range(N):
            for t in range(6):
                for m in range(6):
                    s = observations[n][k]
                    if (m+1 + t+1 == s):
                        self.r_nk[k][n][t][m] = num[k][n][t][m] / den[k][n

def M_step(self):
    # Update categorical distributions: pi_1 and pi_2
    K, N = self.pi_1.shape[0], self.pi_2.shape[0]
    n_cat = (self.pi_1.shape[1], self.pi_2.shape[1])
    num_1, num_2 = np.zeros((K, n_cat[0])), np.zeros((N, n_cat[1]))
    den_1, den_2 = np.zeros((K)), np.zeros((N))

    # Update pi_1: table_dist, pi_km
    for k in range(K):
```

```
            for m in range(n_cat[0]):
                for n in range(N):
                    for t in range(n_cat[1]):
                        num_1[k][m] += self.r_nk[k][n][t][m]

        for k in range(K):
            for m in range(n_cat[0]):
                den_1[k] += num_1[k][m]

        for k in range(K):
            for m in range(n_cat[0]):
                self.pi_1[k][m] = num_1[k][m]/den_1[k]

        # Update pi_2: player_dist, phi_nt
        for n in range(N):
            for t in range(n_cat[1]):
                for k in range(K):
                    for m in range(n_cat[0]):
                        num_2[n][t] += self.r_nk[k][n][t][m]

        for n in range(N):
            for t in range(n_cat[1]):
                den_2[n] += num_2[n][t]

        for n in range(N):
            for t in range(n_cat[1]):
                self.pi_2[n][t] = num_2[n][t]/den_2[n]

    def log_likelihood(self):
        log_likelihood = 0
        K, N = self.pi_1.shape[0], self.pi_2.shape[0]
        n_cat = (self.pi_1.shape[1], self.pi_2.shape[1])
        for n in range(N):
            for k in range(K):
                for t in range(n_cat[1]):
                    for m in range(n_cat[0]):
                        log_likelihood += self.r_nk[k][n][t][m] * \
                            (np.log(self.pi_1[k][m])+np.log(self.pi_2[n][t]))
        return log_likelihood

    def print_params(self):
        print("Params:")
        print("\tTables distribution:")
        for row in self.pi_1:  # each table
            s = "\t\t"
            for p in row:  # prob of each outcome in that table
                s = s + f"|{p:.4f}"
            print(s)
        print("\tPlayers distribution:")
        for row in self.pi_2:  # each player
            s = "\t\t"
            for p in row:  # prob of each outcome in for that player
```

```
            s = s + f"|{p:.4f}"
        print(s)
```

Appendix 5: 3.6.12 Code

```
from casino import Casino
import numpy as np
from em import CategoricalCategoricalEM
import matplotlib.pyplot as plt


def plot_dice_dist(dice, file, title):
    outcomes = ["1", "2", "3", "4", "5", "6"]
    dice = np.mean(dice, axis=0)
    fig, ax = plt.subplots()
    ax.bar(outcomes, dice, label='Real Dice')
    ax.set_title(title)
    ax.set_ylabel("Probability")
    ax.set_xlabel("Outcomes")
    ax.legend()
    plt.savefig(f'fig/{file}')


def plot_dice_em_dist(realDice, emDice, file, title):
    width = 0.3
    outcomes_real = [1, 2, 3, 4, 5, 6]
    outcomes_em = [o+width for o in outcomes_real]
    realDice = np.mean(realDice, axis=0)
    emDice = np.mean(emDice, axis=0)

    fig, ax = plt.subplots()

    ax.bar(outcomes_real, realDice, width=width,
           color='b', align='center', label='Real Dice')
    ax.bar(outcomes_em, emDice, width=width,
           color='r', align='center', label='EM Dice')
    ax.set_title(title)
    ax.set_ylabel("Probability")
    ax.set_xlabel("Outcomes")
    ax.set_xticks(outcomes_real)

    ax.legend()
    fig.tight_layout()

    plt.savefig(f'fig/{file}')


def main():
    # np.random.seed(1)
    nTables = 5
    nPlayers = 3
    tableProb = {
```

```
        # 50% chance of being prime and same of not being prime table_1
        'init': 1/2,
        'change': 3/4
    }

    primedTableDice = np.ones((nTables, 6)) * (1/6)
    notPrimedTableDice = np.ones((nTables, 6)) * (1/6)
    playerDice = np.ones((nPlayers, 6)) * (1/6)  # one dice per player

    primedTableDice[:, 0] = 0.08  # 0.3
    primedTableDice[:, 1] = 0.08  # 0.05
    primedTableDice[:, 2] = 0.08  # 0.15
    primedTableDice[:, 3] = 0.08  # 0.2
    primedTableDice[:, 4] = 0.08  # 0.1
    primedTableDice[:, 5] = 0.6  # 0.2

    notPrimedTableDice = primedTableDice

    playerDice[:, 0] = 0.6  # 0.1
    playerDice[:, 1] = 0.08  # 0.2
    playerDice[:, 2] = 0.08  # 0.3
    playerDice[:, 3] = 0.08  # 0.2
    playerDice[:, 4] = 0.08  # 0.1
    playerDice[:, 5] = 0.08  # 0.1

    print(f"Tables distribution:\n{notPrimedTableDice}")
    print(f"Players distribution:\n{playerDice}")

    plot_dice_dist(notPrimedTableDice, "table-dice.png",
                   "Mean tables' dice distribution")
    plot_dice_dist(playerDice, "player-dice",
                   "Mean players' dice distribution")

    casino = Casino(nTables, nPlayers, tableProb,
                    primedTableDice, notPrimedTableDice, playerDice)
    casino.run()
    observations = casino.S
    print(f"Observations:\n {observations}")

    em = CategoricalCategoricalEM(
        notPrimedTableDice.shape, playerDice.shape, threshold=1e-4)
    em.fit(observations)
    em.print_params()

    emTableDice, emPlayerDice = em.get_params()
    plot_dice_em_dist(notPrimedTableDice, emTableDice, "em-biased-table.png",
                      "Mean real vs EM's tables' dice distribution")
    plot_dice_em_dist(playerDice, emPlayerDice, "em-biased-player.png",
                      "Mean real vs EM's players' dice distribution")


if __name__ == "__main__":
```

main ( )