STAT 535A - Assignment

Gong (Archer) Zhang, Student ID: #86311164

April 5, 2018

This pdf file serves as a supplementary material to the Java and Blang code in my Github repo (click here for link) for this assignment. It contains some theoretical details and explanation of how the code works.

1.1 Basic sampler for permutations

Design an invariant and irreducible sampler for permutations. Describe formally the proposed moved and establish irreducibility and invariance with respect to the uniform distribution on the space of perfect bipartite matchings (equivalently, permutations).

Solution: For any perfect Bipartite matching graph G of size n where each side (left and right) of the graph has n vertices, we represent it as a tuple of length n. For example, when $n = 3$, a tuple $(1, 0, 2)$ represents a perfect matching graph, in which first vertex on the left side is connected to the second vertex on the right side, and the second/third vertex on the left is connected to the first/third vertex on the right, respectively.

To propose the next state from the current one in the Markov Chain, we first uniformly pick two elements (can be two identical elements) in the current tuple, and then swap them. For example, the proposed state can be $x' = (0, 1, 2)$ by swapping the first and second element in the current tuple $x = (1, 0, 2)$. In this way of proposal, the ratio of two proposal probabilities is given by

$$\frac{q(x|x')}{q(x'|x)} = 1,$$

because the swap is same from x to x' and from x' to x.

Therefore, we can generate Markov Chains by the Metropolis-Hastings (MH) algorithm. Let $\pi(\cdot)$ to be our target distribution over our combinatorial space. Then, we accept the proposed state x' with probability

$$\alpha = \frac{\pi(x')}{\pi(x)} \frac{q(x|x')}{q(x'|x)} = \frac{\pi(x')}{\pi(x)}.$$

Theoretical results tell us that transition kernel of the Markov Chain generated from this MH algorithm is π-invariant. Moreover, the constructed Markov Chain is also irreducible because clearly we can jump to any state from the current in finite times. As an example, from state $(1, 0, 2)$ we can move to state $(2, 1, 0)$ in three steps as follows:

$$(1, 0, 2) \to (2, 0, 1) \to (2, 1, 0).$$

The Java code (in `PermutationSampler.execute`) for implementing this sampling method is given in my github repo.

1.2 Understanding the test

Use `Open Type` in eclipse to open and read the comments in the following files:

- `DiscreteMCTest.java`
- `ExhaustiveDebugRandom.java`

Study how these object work. For example, you may want to try using ExhaustiveDebugRandom to automatically construct the decision tree induced by a simple random process such as a Polya urn.

Summarize these algorithms with a description and pseudo-code. Explain why the kernels are tested separately for invariance but mixed together along with an identify matrix for testing irreducibility.

Solution:

(i) Summary and Description of `DiscreteMCTest.java`:

This Java file is used to test for invariance and irreducibility of the Markov Chain that is generated from our sampler. Description of the main class `DiscreteMCTest` and two important functions `checkInvariance()` and `checkIrreducibility()` is given as below.

- Main class `DiscreteMCTest`: A pre-processing step to build target distribution and list of kernels.

Algorithm 1 class DiscreteMCTest

```
1: Let targetDistribution be the target distribution.
2: Let transitionMatrices be the empty list, whose length is the number of samplers we want to study.
3: Exhaustively enumerate all the possible configurations, and find the corresponding unnormalized and
   exact normalized probabilities.
4: Construct the target distribution as a vector based on the probabilities that are calculated previously.
5: for each kernel we study do
6:     Build the transition matrix for this kernel, whose $(i, j)$-th entry is the addition of the probabilities
   (available from step 3) of all possible transitions from state $i$ to state $j$ in the exhaustive list of
   configurations.
7:     Store the built transition matrix in the list transitionMatrices.
```

- Function `checkInvariance()`: check if the transition matrix for each MCMC individual kernel is π-invariant.

Algorithm 2 Check Invariance

```
1: for every transition matrix $K_i$ in the list transitionMatrices do
2:     Compute the one step probability $\pi^\top K_i$.
3:     if $\pi^\top K_i = \pi^\top$ then;
4:         Print Invariance of the kernel $K$ is established successfully;
5:     else
6:         Return failure of the test: The $i$-th kernel is not $\pi$-invariant.
```

- Function `checkIrreducibility()`: check the irreducibility of the mixture of the kernels.

Algorithm 3 Check Irreducibility

```
1: Denote by $m$ the number of transition matrices in the list transitionMatrices.
2: Denote by $n$ the state space size, which is equal to nrow and ncol of each transition matrix.
3: Define the mixing proportion to be $w = 1/(1+m)$.
4: Define the mixture matrix $MIX = wI_n + \sum_{i=1}^{m} wK_i$, where $K_i$ is the $i$-th kernel in the list
   transitionMatrices, and $I_n$ is the identity matrix of size $n$.
5: Initialize: let $v = (1, 0, \cdots, 0)$ be a row vector of length $n$.
6: for $i$ from 1 to $n$ do
7:     Set $v = vMIX$;
8:     Count the number of non-zero elements in the updated row vector $v$, call it $b$;
9:     if $b = n$ then
10:        Print Irreducibility is achieved at the $i$-th step.;
11: if nothing is printed at all then
12:     Return Failure of the test: The mixture kernel is not irreducible (in $n$ steps).
```

Explanation: From the line 6-10 in the pseudo code algorithm, what we check is whether we are able to reach every state in the the full state space starting from the initial state in at most n steps. If not, we declare the test to be failed.

(ii) Summary and Description of `ExhaustiveDebugRandom.java`:

I will try to summarize and explain what the three important functions do in the `ExhaustiveDebugRandom`, namely: `nextCategorical`, `nextBernoulli` and `nextInt`.

- `nextCategorical` samples an index from a discrete distribution according to some probability, by looping over branches of the "tree".
- `nextBernoulli` generates a binary variable, whose probability of being 1 is specified by a given parameter p. It uses the function `nextCategorical` to generate binary samples by specifying the distribution over two categories $(p, 1-p)$.
- `nextInt` generates a uniform random variable among a list of size n, each of which has probability $1/n$ to be chosen. It also uses the function `nextCategorical` to generate samples by specifying the distribution over n categories $(1/n, 1/n, \cdots, 1/n)$.

(iii) Explanation of why the kernels are tested separately for invariance but mixed together along with an identify matrix for testing irreducibility.

For a list of kernels $\{K_i\}_{i=1}^{m}$, each of which has size $n \times n$, we define a mixture as

$$MIX = \frac{1}{m+1} I_n + \sum_{i=1}^{m} \frac{1}{m+1} K_i,$$

where I_n is the identity matrix of size $n \times n$.

Since we put equal weights $w = 1/(1+m)$ on each of the kernel we study, it is clear that, if every kernel $K_i, 1 \leq i \leq m$ is π-invariant, then the mixture kernel MIX is also π-invariant because:

$$\pi^\top MIX = \frac{1}{m+1}[\pi^\top I_n + \sum_{i=1}^{m} \pi^\top K_i] = \frac{1}{m+1}[\pi^\top + m\pi^\top] = \pi^\top.$$

Therefore, if we can test that whether each kernel K_i is π-invariant, then the mixture MIX automatically satisfies the invariance property. However, to make the test more informative, we also want to keep track of which kernel is not π-invariant if the mixture kernel failed to be π-invariant.

In terms of test of irreducibility for the mixture kernel MIX, it does not make practical sense to check irreducibility for each kernels because the mixture MIX is irreducible if any kernel K_i is irreducible. Therefore, if the irreducibility test for the mixture fails, then automatically know that all of the kernels K_i are not irreducible. On the other hand, if the mixture passes the irreducibility test, then we are free to use the mixture kernel MIX to make inference (e.g. LLN and CLT).

1.3 Bipartite matching (non-perfect)

To make things a bit more interesting, consider now the problem of sampling bipartite matchings, i.e. we do not require the matching to be perfect anymore (vertices do not have to be covered).

The datastructure is implemented for you in `BipartiteMatching.xtend`, which you should have a look at. Your job is to implement a sampler for this combinatorial space in `BipartiteMatchingSampler.java`, which you can test via `TestBipartiteMatching.xtend`.

Solution:

Same as solution for **1.1**, we represent the non-perfect Bipartite matching graph by a tuple of length n, where n is the number of vertices on each side (left and right) of the matching. However, we use -1 to represent that the corresponding node is not connected to any vertex. For example, when $n = 3$, the tuple $(-1, 0, 2)$ corresponds to the non-perfect matching graph where the first vertex on the left side is connected to nothing on the right side, and the second/third vertex on the left is connected to the first/third vertex on the right, respectively.

To propose the next state from the current one in the Markov Chain, we first uniformly pick one element in the current tuple, and then propose to replace it by either -1 or all the other elements not in the current tuple. For example, if the current state is $x = (-1, 0, 2)$, we can propose the next state x' by replacing the second element 0 in x with either -1 or 3. In this way of proposal, the ratio of two proposal probabilities is given by

$$\frac{q(x|x')}{q(x'|x)} = 1,$$

because the replacements from x to x' and from x' to x have equal probability to occur.

Therefore, we can generate Markov Chains by the Metropolis-Hastings (MH) algorithm. Let $\pi(\cdot)$ to be our target distribution over our combinatorial space. Then, we accept the proposed state x' with probability

$$\alpha = \frac{\pi(x')}{\pi(x)} \frac{q(x|x')}{q(x'|x)} = \frac{\pi(x')}{\pi(x)}.$$

Theoretical results tell us that transition kernel of the Markov Chain generated from this MH algorithm is π-invariant. Moreover, the constructed Markov Chain is also irreducible because clearly we can move to any state from the current in finite number of moves. For example, it is possible to move from state $(-1, 0, 2)$ to state $(2, 1, 0)$ in the following way:

$$(-1, 0, 2) \overset{\text{pick 3rd}}{\longrightarrow} (-1, 0, -1) \overset{\text{pick 1st}}{\longrightarrow} (2, 0, -1) \overset{\text{pick 2nd}}{\longrightarrow} (2, 1, -1) \overset{\text{pick 3rd}}{\longrightarrow} (2, 1, 0).$$

The Java code (in `BipartiteMatchingSampler.java`) for implementing this sampling method is given in my github repo.

2.1 A statistical model involving a combinatorial space

Implement in `PermutedClustering.bl` a model in Blang for the problem as described in the github repo.

Solution: See my github repo for implementation of the model in Blang.

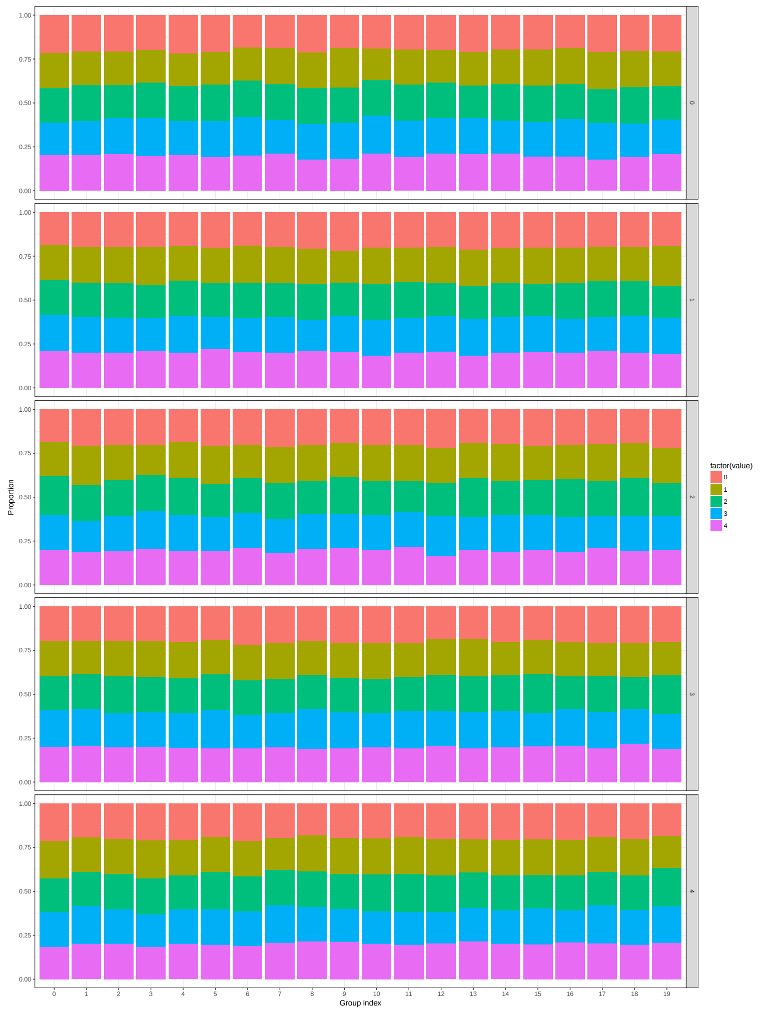

Figure 1: Plot: permutations-posterior.pdf

After testing the model on some synthetic data, the plot for posterior statistics on inferred permutations produced by nextflow is attached in Figure 1.

Note: Since I was not familiar with Java before, students in this class Weining Hu, Wayne Wang and Qiong Zhang helped me a lot in coding in our discussion.