PROJECT 3: BINARY TABLE COUNTING

May 19, 2017

Liam Walsh

Contents

0.1 INTRODUCTION

The problem of counting binary tables is well studied, with applications across disciplines, from pure mathematics to computer science, and beyond. In this project, I consider a specific kind of counting table problem. Here, I am given an mxn table, with certain sum constraints for both the rows $\mathbf{r} = (r_1, r_2, ...r_m)$ and the columns $\mathbf{c} = (c_1, c_2, ..., c_n)$, with $\sum_{i=1}^{m} r_i = \sum_{j=1}^{n} c_j$. I want to count the number of tables such that, for a table X, where $x_{ij} \in \{0, 1\}$ is a particular entry in the table,

$$\sum_{j=1}^{n} x_{ij} = r_i \tag{1}$$

$$\sum_{i=1}^{m} x_{ij} = c_j \tag{2}$$

This problem of counting, while combinatoric in nature, can be easily converted into a probabilistic framework. Let **B** denote the set of all binary tables. Clearly, it has size 2^{nm}. Further, let **B*** denote the set of all binary tables which satisfy given constraints **r,c** as given above. Then, the probability that some binary table $\mathbf{X}^T$ satisfies the constraints is given by:

$$P(\mathbf{X}^T \in \mathbf{B^*}) = \frac{|\mathbf{B^*}|}{|\mathbf{B}|} \tag{3}$$

In estimating the probability in (3), I can find the fraction of tables that satisfy the given constraints. Further, I can estimate how many tables satisfy the constraints, simply by multiplying the estimated probability by the size of **B**

0.2 NAIVE IMPLEMENTATION

In this problem, the scaling parameter is n, which is the number of column constraints (and which is the upper bound for the number of row constraints). For smaller n, it is easy enough to simply just enumerate the number of tables by hand, and hence, calculate directly the fraction of tables satisfying the given constraints. For larger n, when direct calculation becomes infeasible, simulation methods, like the one proposed below, become a way of enumerating the number of tables. In this section, I will give some results of a single simple simulation method, as well as precisely detail at what size of n this method becomes unusable due to the desired state space becoming too small.

0.2.1 Method Details

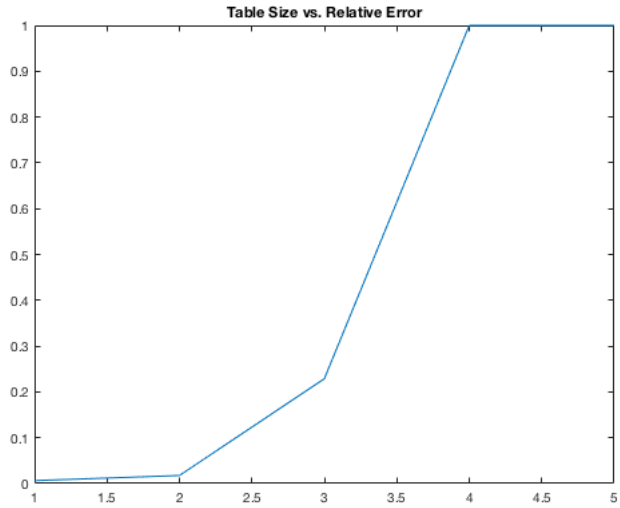
A fairly simple way of generating small tables that satisfy the constraints is through simple MCMC computation. Here, I can generate a Markov Chain whose state space is all of the binary tables, and whose stationary distribution is uniform over these state spaces. Once I have constructed this Markov Chain, I can generate a large number of samples, and use the ergodic theorem to estimate the probability of the chain being in the desired **B***, denoted by $\hat{l}$. The pseudocode for this algorithm is below:

Algorithm 1 Naive MCMC Estimation

$N_T \leftarrow 0$
for $r = 1 : s$ **do** ▷ s is the desired number of samples
    $X \leftarrow rand$ ▷ Randomly assign 0s and 1s
    if $X \in$ **B*** **then**
        $N_T \leftarrow N_T + 1$
return $\hat{l} = \frac{N_T}{s}$

0.2.2 Results

For very small n (like 2 or 3), this method works well, even for rare tables. However, at a table size of 4x4 and beyond, this method doesn't work for rare tables anymore. See the graph below for the comparison of the relative error of the method as n increases.



This figure was obtained by generating 10000 samples, and using the Naive MCMC algorithm to estimate the probability of a rare table. In this case, the rare table was a table of all 0s, meaning r and c were just the 0 vectors. I selected this as a starting point, since the probability was far easier to calculate exactly than the probabilities of other rare events. Overall, then, this method is severely limited by table size. It is no longer effective on rare tables for $n \geq 4$. At $n = 4$, the state space of all tables is a large 65536 different tables, and trying to just hit one specific table out of those thousands is difficult. This issue is not resolved when increasing n to larger values. Even at $n = 10000000$, the naive method is unable to generate a single table that fits the constraints, and gives the probability of a rare event (like a table of all 0s) to be 0. This is also not fixed by selecting slightly less rare tables. For tables that appear with slightly higher probabilities (specifically, tables with all row and sum constraints to be 2), this method is able to handle a table size of 4x4, but not 5x5. Overall, then, it is clear that for rare tables (which are often the most interesting), this naive method scales very poorly.

0.2.3 Note on Table Sizes

As an aside, for all of the analysis done above (and in the next section), I will be discussing square tables alone, that is, tables whose dimensions are nxn. The main reason for this is that it provides an upper bound on the power of an algorithm. Consider, for example, a long and skinny table (perhaps $m = 2$ but $n = 5$). In this case, while the naive method typically has problems with $n = 5$, in reality, it would not have an issue correctly estimating the probability. This is because the state space here is of size 1024, which is only a moderate step up from the $3x3$ table. This still applies to tables that have a moderate difference in dimensions (say, a 14

by 8 table). Evidence of this can be seen in the example 5x2 table in my code.
On the other end of the spectrum, consider a table where m is close to n. Here, the state space will be similar enough in size to the size of the full square table, enough so that it makes sense just to consider the full square table instead. Thus, to simplify analysis, and to get an upper bound on computational abilities, I only consider square tables.

0.3 SPLITTING

For estimating rare events, splitting is a powerful MCMC tool. In constructing the splitting, there are 2 things to consider. First, I need to devise a performance function which can be used to determine when to split. Then, I need to construct an algorithm to perform Gibbs sampling, so that it can be used in the splitting algorithm.

0.3.1 Performance Function & Gibbs Move

A useful performance function can be:

$$S(\mathbf{X}) = -\sum_{i=1}^{m} |\sum_{j=1}^{n} x_{ij} - r_i| \tag{4}$$

This performance function is a measure of how many row sums are correct, given that all the column sums are correct
As for the Gibbs move, the following algorithm works well:

Algorithm 2 Gibbs Move

$Y \leftarrow X$
for $j = 1 : n$ **do**
  $I \leftarrow \{i : Y_{ij} = 1\}$
  for $i \in I$ **do**
    $K \leftarrow \{k : Y_{kj} = 0\} \cup \{i\}$
    $M \leftarrow \varnothing$
    for $k \in K$ **do**
      $Y' \leftarrow Y$; $Y'_{ij} \leftarrow 0$; $Y'_{kj} \leftarrow 1$
      if $S(Y') \geq \gamma$ **then** $M \leftarrow M \cup \{k\}$
      Choose M uniformly from M
      $Y_{ij} \leftarrow 0$; $Y_{Mj} \leftarrow 1$
return Y

As for the splitting, there are typically 3 different ways to go about it. I selected to do fixed

effort splitting. The exact specifics of the fixed effort algorithm can be seen in Rubinstein & Kroese, or in my supplementary code.

0.3.2 Fixed Effort Parameter Selection

For the splitting problem, there are 2 main parameters that can be tweaked for performance - the number of initial samples (N), and the sequence of performance levels $\boldsymbol{\gamma} = (\gamma_1, \gamma_2, ...\gamma_T)$, where $\gamma_T = 0$, which is the final level. For each individual problem, I hand selected performance levels based on what the maximum error could be based on the performance function. I also found that as samples progressed through the levels, it was unnecessary to have levels that were 1 apart, and instead, levels that were 2 apart (like [-8,-6,-4,-2,0]) functioned suitably. This is by nature of the performance function, as it only ever returns even numbers. If a table is wrong, at the very least, it means there is a 1 where there should be a 0, and a 0 where there should be a 1, resulting in an error of -2.
I chose $N = 20000$ because the book (Rubinstein & Kroese) and paper (Chen et al.) use this number, and for my purposes, it worked well. Further, increasing N did not significantly affect performance. In moving from $N = 20000$ to $N = 50000$, there was no performance change for small tables, and for the $12x12$ table presented in the book, there was a very minimal decrease in relative error.

0.3.3 Splitting Results

As expected, this method performs quite well, even for rare events on larger tables that the naive method could not handle. Below is a table of results for simulating the event of all the rows and columns summing to 2, from a 3x3 table to a 7x7 table, with the exact values given by Wang and Zhang (1998). Note that the 2x2 table has been excluded since the method doesn't correctly work. This is due to how the initial samples are generated. They are generated uniformly from a space where all of the column constraints have been satisfied, and in the 2x2 case, this means every sample initial sample would satisfy the all the constraints (since there is only one way for a 2x2 table to have the column constraints satisfied), and the estimated probability would be 1.
For even larger tables, this method was still able to accurately determine probabilities. The book presents a sample table, which is $12x12$, with all row and column constraints equal to 2. The true number of tables that satisfy these constraints is 21,959,547,410,077,200, which results in a probability of $9.847 * 10^{-28}$. Using 100 iterations, I found the probability to be $9.344 * 10^{-28}$, with 5.11% relative error. This is in line with the findings in the book. Overall, then, it is clear that the splitting algorithm is able to properly estimate the probabilities of

Table Size	True Prob	Est. Prob	Relative Err.
3x3	0.0117	0.0117	0
4x4	0.0014	0.0014	0
5x5	$6.1095 * 10^{-5}$	$6.1070 * 10^{-5}$	$4.0920 * 10^{-4}$
6x6	$9.888 * 10^{-7}$	$9.8713 * 10^{-7}$	0.0017
7x7	$5.5261 * 10^{-9}$	$5.5389 * 10^{-9}$	0.0023

rare events, even for large tables.

0.4 SOURCES

Chen, Yuguo, et al. "Sequential Monte Carlo Methods for Statistical Analysis of Tables". *Journal of the American Statistical Association.* 100, 109-120. 2005. statweb.stanford.edu. Web. 18 May 2017.

Rubinstein, Reuven Y., and Dirk P. Kroese. *Simulation and the Monte Carlo Method.* Hoboken, NJ: Wiley, 2017. Print.

Wang, B. Y., and Zhang, F., "On the Precise Number of (0, 1)-Matrices in U(R, S)," *Discrete Mathematics*, 187, 211-220. 1998. sciencedirect.com. Web. 18 May 2017.

Naive MCMC

Table of Contents

Simple Base Case - Sanity Check

```
r = 1;
c = 1;
s = 1000;
N = 0;
for i = 1:s
    X = rand >= .5;
    if checkSumConst(X,r,c)
        N = N + 1;
    end
end
l = N/s; % Should be .5
```

Simple Case - Rectangular Table

```
r = [1;1];
c = 2;
s = 10000;
N = 0;
for i = 1:s
    X = rand(2,1) >= .5;
    if checkSumConst(X,r,c)
        N = N + 1;
    end
end
l = N/s; % Should be .25
```

Simple Case - Square Table

```
r = [1;1];
```

```
c = [2,0];
s = 10000;
N = 0;
for i = 1:s
    X = rand(2) >= .5;
    if checkSumConst(X,r,c)
        N = N + 1;
    end
end
l = N/s; % Should be .0625
```

Rare Event - 3x3 Table

```
n = 3;
m = 3;
r = [0;0;0];
c = [0,0,0];
s = 10000;
N = 0;
for i = 1:s
    X = rand(3) >= .5;
    if checkSumConst(X,r,c)
        N = N + 1;
    end
end
l = N/s; % Should be .0020
```

5x5 Table - Rare events no longer simluated

```
r = [0;0;0;0;0];
c = [0,0,0,0,0];
s = 10000;
N = 0;
for i = 1:s
    X = (rand(5) >= .5);
    if checkSumConst(X,r,c)
        N = N + 1;
    end
end
l = N/s; % Should be 2.9802*10^-8
```

Rectangular for Test - 5x2 Table

```
r = [1;0;0;1;0];
c = [1,1];
X = rand(5,2) >= .5;
s = 10000;
N = 0;
for i = 1:s
    X = rand(5,2) >= .5;
    if checkSumConst(X,r,c)
```

```
        N = N + 1;
    end
end
l = N/s; % Should be .0020
         % Shows comparable performance to 3x3 and smaller - same
 state space size
```

Large Table From Paper

```
c = 2*ones(1,12);
r = 2*ones(12,1);
s = 50000;
N = 0;
for i = 1:s
    X = rand(12) >= .5;
    if checkSumConst(X,r,c)
        N = N + 1;
    end
end
l = N/s;
```

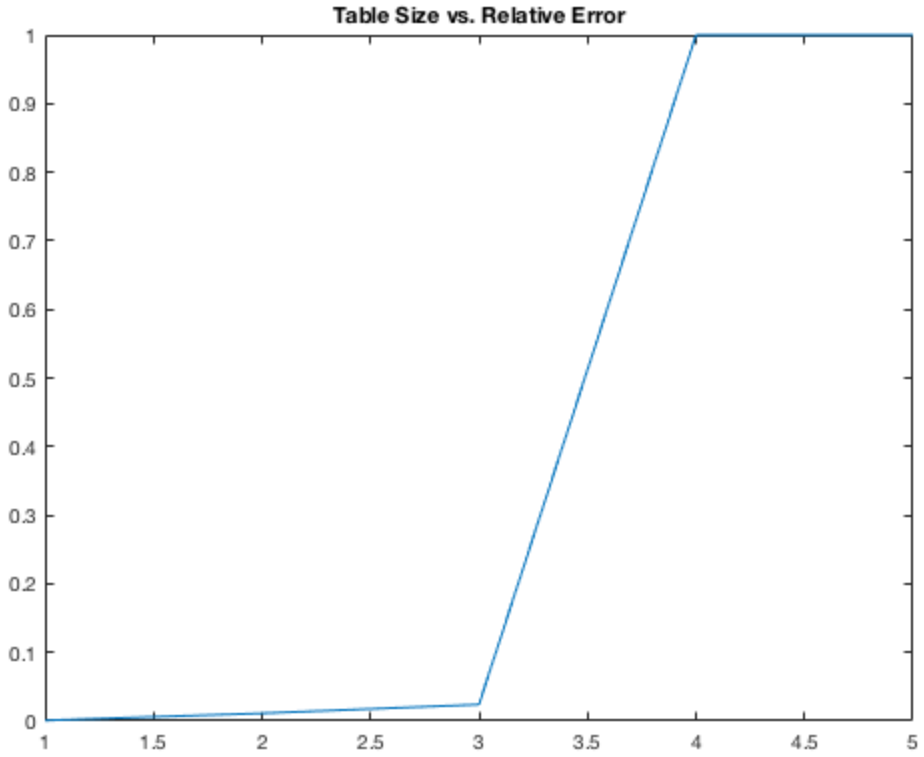
Comparison of Rare Events

Here, rare events mean row and column constraints that are satisfied by a single table configuration

```
estimates = zeros(1,5);
true_probs = [1/2, 1/16, 1/512, 1/(2^16), 1/(2^25)];
s = 10000;

for i = 1:5
    r = zeros(i,1);
    c = zeros(1,i);
    N = 0;

    for j = 1:s
        X = rand(i) >= .5;
        if checkSumConst(X,r,c)
            N = N + 1;
        end
    end
    estimates(i) = N/s;
end
rel_err = abs(1-estimates./true_probs);
plot(1:5,rel_err)
title('Table Size vs. Relative Error')
```



Rare Event - 4x4 Table

```
r = 2*ones(4,1);
c = 2*ones(1,4);
s = 10000;
N = 0;
for i = 1:s
    X = rand(4) >= .5;
    if checkSumConst(X,r,c)
        N = N + 1;
    end
end
l = N/s; % .0014
```

Rare Event - 5x5 Table

```
r = 2*ones(5,1);
c = 2*ones(1,5);
s = 10000;
N = 0;
for i = 1:s
    X = rand(5) >= .5;
    if checkSumConst(X,r,c)
        N = N + 1;
    end
```

```
end
l = N/s; %6.0797 * 10^-5
```

Functions Used:

```
dbtype checkSumConst.m
```

```
1     function bool = checkSumConst(X,r,c)
2     % A function which checks if the given column  (c) and row (r)
 sum
3     % constraints are satisfied by the input table, and returns a
 boolean
4     % indicating so
5     %
6     % INPUT:
7     %       X - An n x m binary table, whose values are to be
 checked
8     %       r - The row constraints, a 1 x n vector
9     %       c - The column constraints, a 1 x m vector
10    %
11    % OUTPUT:
12    %       bool - A boolean (1 or 0) indicating if the given X
 satisfies the
13    %       desired row and column constraints
14
15    bool = (isequal(sum(X,1),c) && isequal(sum(X,2),r));
```

Splitting MCMC

Table of Contents

3x3 to 7x7 Table - Rare Events

```
est = zeros(1,5);
rel_err = zeros(1,5);
num_tables = [6,90,2040,67950,3110940];
for i = 3:7
    r = 2*ones(i,1);
    c = 2*ones(1,i);
    g = -i:2:0;
    est(i-2) = AFESplitting(r,c,g);
    real = num_tables(i-2)/(2^(i^2));
    rel_err(i-2) = abs(1-est(i-2)/real);
end
```

Large Table From Paper

```
num_samps = 100;
lhat = zeros(1,num_samps);

c = 2*ones(1,12);
r = 2*ones(12,1);
g = -10:2:0;
for i = 1:num_samps
    lhat(i) = AFESplitting(r,c,g);
end
l = mean(lhat);
std = sqrt(var(lhat));
bounds = [l+1.96*std/sqrt(num_samps),1-1.96*std/sqrt(num_samps)];
```

Functions Used:

```
dbtype AFESplitting.m
dbtype AFESSampler.m
dbtype gibbsMove.m
```

```
1     function lhat = AFESplitting(r,c,g)
2     %
3     % A function which performs fixed effort splitting to estimate
 the
4     % probability that a binary table satisfies the given row and
 column sum
5     % constraints. Here, N = 20000
6     %
7     % INPUT:
8     %       r - An mx1 matrix of row sum constraints
9     %       c - A 1xn matrix of column sum constraints
10    %       g - A vector of performance levels, the last value of
 which is 0
11    %       (or -1, which are functionally equivalent, since the
 performance
12    %       function error will always be even)
13    %
14    % OUTPUT:
15    %       lhat - The estimated probability that a binary table
 satisfies the
16    %       given constraints
17
18    N = 20000;
19    pf = @(x) (-sum(abs(sum(x,2)-r))); % Performance function
20
21    total_N = [];
22    Y = {};
23    for i = 1:N
24        samp = AFESSampler(r,c);
25        if pf(samp) >= g(1)
26            Y{end+1} = samp;
27        end
28    end
29
30    Nt = numel(Y);
31    for gt = 1:numel(g)-1
32        total_N = [Nt, total_N];
33        Yt = {};
34        B = zeros(1,Nt);
35        B(randperm(Nt,mod(N,Nt))) = 1;
36
37        for i = 1:Nt
38            R = floor(N/Nt) + B(i);
39            Y_samp = Y{i};
40            for j = 1:R
41                Y_samp = gibbsMove(Y_samp,g(gt),r);
42                if pf(Y_samp) >= g(gt+1)
43                    Yt{end+1} = Y_samp;
```

```
44                end
45            end
46        end
47        Nt = numel(Yt);
48        Y = Yt;
49    end
50    total_N = [Nt,total_N];
51    lhat = cumprod(total_N/N);
52    lhat = lhat(end);
```

```
1     function samp = AFESSampler(r,c)
2     %
3     % A function to uniformly sample from the tables that satisfy
 the given
4     % column constraints c
5     %
6     % INPUT:
7     %       r - An mx1 vector of row sum constraints
8     %       c - A 1xn vector of column sum constraints
9     %
10    % OUTPUT:
11    %       samp - An mxn binary table with all of the column
 constraints c
12    %       satisfied
13
14    m = numel(r);
15    n = numel(c);
16    samp = zeros(m,n);
17
18    for i = 1:n
19        toSwap = randperm(n,c(i));
20        samp(toSwap,i) = 1;
21    end
22
23    end
```

```
1     function Y = gibbsMove(X,g,r)
2     %
3     % A function that performs Gibbs sampling on an input binary
 table, and
4     % outputs a newly sampled table which still satisfies the
 performance
5     % function to at least level g
6     %
7     % INPUT:
8     %       X - The initial binary table sample
9     %       g - The current performance level value, which the input
 table must
10    %       satisfy given the performance function
11    %       r - An mx1 vector of row sum constraints
12    %
13    % OUTPUT:
14    %       Y - A new binary table obtained via Gibbs sampling from
 X. It
```

```
    %       satisfies S(Y) >= g

    Y = X;
    n = size(X,2);

    for j = 1:n
        I = find(Y(:,j)==1);
        for i = 1:numel(I)
            K = find(Y(:,j)==0);
            M = [];
            for k = 1:numel(K)
                Y_prime = Y;
                Y_prime(I(i),j) = 0;
                Y_prime(K(k),j) = 1;
                if (-sum(abs(sum(Y_prime,2)-r))) >= g
                    M = [M,K(k)];
                end
            end
            if ~isempty(M)
                m = M(randi(numel(M)));
                Y(I(i),j) = 0;
                Y(m,j) = 1;
            end
        end
    end
```

Published with MATLAB® R2016b