EP 4 MAP 2212

Nicholas Gialluca Domene
N USP 8543417 - IME
Felipe de Moura Ferreira
N USP 98674702 - IME
Universidade de São Paulo

June 27, 2021

Problem Statement

Taken from https://www.youtube.com/watch?v=ZmaYDTLR8yw

- Consider the m-dimensional Multinomial statistical model, with observations, x, prior information, y, and parameter Theta;

$$-x, y \in N^m, \Theta \in Sm = \{\Theta \in R^{+m} | \Theta' 1 = 1\}, m = 3; \tag{1}$$

The statistical model comprises:
- Posterior density potential,

$$f(\theta|x,y) = \frac{1}{B(x+y)} \Pi_{I_i=1}^{m} \theta_i^{x_i+y_i-1} \tag{2}$$

- Cut-off set,

$$T(v) = \{\Theta \in Sm | f(\theta|x,y) \leq v\}, v \geq 0 \tag{3}$$

- Truth function,

$$W(v) = \int_{T(v)} f(\Theta|x,y) d\Theta \tag{4}$$

W(v) is the posterior probability mass inside T(v), that is, the probability mass where the posterior potential, $f(\Theta|x,y)$, does not exceed the threshold level v

$$-Dirichlet(\theta|a) = \frac{\Pi_{i=1}^{m} \theta_i^{a_i-1}}{B(a)}, \tag{5}$$

$$B(a) = \frac{\Pi_{i=1}^{m} \Gamma(a_i)}{\Gamma(\sum_{i=1}^{m} a_i)} \tag{6}$$

B(a) is the multivariate Beta function
- Set k cut-off points, $0 = v_0 < v_1 < v_2 < ... < v_k = sup f(\Theta)$
- Use a Gamma RNG to generate n points in Sm, $\Theta_1, ..., \Theta_n$, distributed according to the posterior density function
- Use the fraction of simulated points, Θ_t, inside each bin;, $v_{j-1} \leq f(\Theta_t) < v_j$, as an approximation of $W(v_j) - W(v_{j-1})$
- Dynamically adjust the bin's borders, v_j, to get bins of approximately equal weight, i.e., $W(t_j) - W(t_{j-1}) \approx \frac{1}{k}$

Implementation

The algorithms were implemented using Python and the usage instructions can be found below.
The overall strategy was treated the implementation as an experiment that using the proposed conditions by the problem statement and a sufficiently large n (defined below), should yield an estimate to the integral W(v) with a Confidence Interval lesser or equal to 0.0005 with 95% confidence.
We defined n through an algebraic manipulation of the Normal approximation of the Binomial distribution:

$$\begin{aligned} Z_{score} &= \frac{mdd}{\sqrt{\frac{\sigma^2}{n}}} \\ Z_{score} \cdot \frac{\sqrt{\sigma^2}}{\sqrt{n}} &= mdd \\ \frac{Z_{score} \cdot \sqrt{\sigma^2}}{mdd} &= \sqrt{n} \\ n &= \frac{Z_{score}^2 \cdot \sigma^2}{mdd^2} \end{aligned} \tag{7}$$

where mdd is the "*Minimal Detectable Difference*" which in this case is 0.0005, meaning that in the worst case scenario it will 0.0005 (if $\int_0^1 f(x)dx = 1$). Considering the worst case scenario:

$$mmd = 0.0005, \quad Z_{score} = 1.65, \quad \sigma^2 = 0.25 \tag{8}$$

$$n = \frac{1.65^2 \cdot 0.25}{0.0005^2} = 2722500 \tag{9}$$

How to run:

1. start by initiating the EP4 class defined in `ep4_class.py` inputting the vector `x` and `y`, both of dimension 3. In the `__init__` method, we will store both x and y vectors, a `alpha` vector that is the sum of each x_i and y_i for each $alpha_i$, the constant $B(x + y)$ of the denominator of the evaluating function $f(\Theta|x, y)$ and the `n` for generating `n` Θ observations of the simplex generated by the Dirichlet distribution.

```
def __init__(self, x, y):
      self.x = x
      self.y = y
      self.n = 2722500
      self.alpha = [x[i] + y[i] for i in range(len(x))]
```

```
    numerator = 1
    for i in range(len(self.alpha)):
      numerator *= math.gamma(self.alpha[i])
    B_x_y = numerator / math.gamma(sum(self.alpha))

    self.denominator_constant = B_x_y
```

2 run the method `generate_theta` that will generate the `n` Θ observations through the Dirichlet distribution with parameters `alpha` and store then into the variable `thetas`.

```
def generate_theta(self):
    thetas = np.random.dirichlet(self.alpha, self.n)
    self.thetas = thetas
```

3 execute the method `order_f_thetas` that will create a list `f_thetas` with the values of $f(\Theta, y)$ for each Θ_i and sort that list in ascending manner. The method will store the ordered list of $f(\Theta)$ values, the minimum and the maximum value for $f(\Theta)$ in the generated observations. This runs in $O(n \log n)$ time, n being the number of Θ observations generated.

```
def order_f_thetas(self):
    f_thetas = [self.f(theta) for theta in self.thetas]
    f_thetas.sort()

    self.ordered_f_thetas = f_thetas
    self.min_f = f_thetas[0] #min value of f_thetas since it is ordered
    self.sup_f = f_thetas[-1] #max value of f_thetas since it is ordered
```

4 run the method `U(v)` passing `v` as the parameter to the desired output function `U`. The idea behind this method is:
Since we have an ordered list the the values of $f(\theta)$ for each θ in our sample space, then we need to find out how many theta observations have $f(\theta)$ below certain v. By finding the index where we would insert a new observation of value v in our ordered list of $f(\theta)$, we use that index to determine how many observations are to the left (lower values). The number of observations whose $f(\theta)$ values are below v divided by the total number of observations (n) is the estimate for W(v). This decreases exponentially the running time and gives a better estimate of W(v) than using bins since in the worst case scenario, it is exactly the same as using the proposed bins given that the is runs in $O(\log n)$ time.
So we are not using bins or the k-varible mentioned in the problem statement because this implementation is both faster and more accurate.

```
  def U(self, v):

    if v > self.sup_f:
      return 1
    if v < self.min_f:
        return 0

    n = self.n
    i = bisect.bisect_left(self.ordered_f_thetas, v)
    return (i + 1)/n #index divided by total n points
```

In the `ep4.py` file, you will find a example of usage passing the `x` and `y` vectors and a few `test_cases`, ready to be tested in the evaluation process:

```
from ep4_class import EP4
x = [4, 6, 4]
y = [1, 2, 3]

test_cases = [0, 1, 0.5, 15, 20]

ep4 = EP4(x, y)

ep4.generate_theta()

ep4.order_f_thetas()

for test in test_cases:
   print("U(%s) = "%(test), ep4.U(test))
```

Results

```
[09:35:01] nicholas.domene ~/Documents/map2212/map2212/EP4 MAP2212 (main *% u=)
   $ python3 ep4.py
Initiating...
U(0) = 0
U(1) = 0.040963820018365474
U(0.5) = 0.018735353535353536
U(15) = 0.8910321395775941
U(20) = 1
Time taken: 8.587098836898804 seconds
```

Conclusion

We managed to create an algorithm that yields the desired accuracy (absolute error smaller than 0.0005 with 95% confidence), an empirical precision until the 3rd decimal (many different runs yield results with that same precision, being different from the 4th decimal place and beyond) and with the fastest run time we heard of during benchmarks in the class discussion at the forum (Whatsapp) with an average of less than 10 seconds per test case given

x and y vectors.
We attribute this results to the usage of the sorted list of $f(\theta)$ values and the binary search to figure out how many θ values yield $f(\theta)$ smaller or equal to the given threshold v (fed into the U(v) evaluating function) and believe this is a good option to go forward.

Next steps

To achieve greater precision, we believe it only takes a greater sample size of θ generated observations which should increase run time (approx) linearly. To test this, we ran an experiment with an n of 10,000,000 observations and it yielded the following results and run time:

```
[09:35:14] nicholas.domene ~/Documents/map2212/map2212/EP4 MAP2212 (main *% u=)
    $ python3 ep4.py
Initiating...
U(0) = 0
U(1) = 0.0411287
U(0.5) = 0.018692
U(15) = 0.8909363
U(20) = 1
Time taken: 33.3142511844635 seconds
```

Which, as expected, shows that by increasing the sample size by an order of 4, also increases the run time by approximately 4 times and greatly improves precision.