EP 5 MAP 2212

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

July 8, 2021

Problem Statement

Taken from https://www.youtube.com/watch?v=G5fGaT0RpYo

- Re-submit your 4th Programming exercise,
- Replacing the random sampler you have used, namely, the exact sampler for the Dirichlet distribution based on an Acceptance-Rejection generator for the Gamma distribution...
- by a Markov Chain Monte Carlo sampler based on the knowledge of just a potential for the Dirichlet distribution.
- Use as kernel for the MCMC a Multivariate Normal N(0, Σ) where the covariance matrix is ajusted based on your prior knowledge of the parameters of the Dirichlet andor some initial or trial sampling sequences.

Methodology

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:

$$
\begin{aligned}
Z_{score} &= \frac{mdd}{\sqrt{\frac{\sigma^2}{n}}} \\
Z_{score} \cdot \frac{\sqrt{\sigma^2}}{\sqrt{n}} &= mdd \\
\frac{Z_{score} \cdot \sqrt{\sigma^2}}{mdd} &= \sqrt{n} \\
n &= \frac{Z_{score}^2 \cdot \sigma^2}{mdd^2} \qquad (1)
\end{aligned}
$$

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:

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

$$n = \frac{1.65^2 \cdot 0.25}{0.0005^2} = 2722500 \tag{3}$$

How to run:

1. start by initiating the EP5 class defined in `ep5_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.
Here, we also save the Covariance matrix that is going to be used for the Multivariante normal sampling at the Metropolis method, based on `https://pt.wikipedia.org/wiki/Distribui%C3%A7%C3%A3o_de_Dirichlet#Propriedades`

```
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

      # Covariance matrix to feed Multinormal Distribution for Theta generation
      covariance_matrix = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
      a_0 = sum(self.alpha)
      for i in range(len(self.alpha)):
         for j in range(len(self.alpha)):
            if i == j:
               covariance_matrix[i][j] = (a_0 -
                  self.alpha[i])*self.alpha[i]/((a_0**2)*(a_0 + 1))
            else:
               covariance_matrix[i][j] =
                  -1*self.alpha[i]*self.alpha[j]/((a_0**2)*(a_0 + 1))
      self.covariance_matrix = covariance_matrix
      self.multivariate_mean = [0 for i in range(len(self.alpha))]
```

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`. This Dirichlet sampling is the objective of this EP and was built first by burning 10000 points to heat up the system. Next, we generate the desired n Θ observations.

```
def generate_theta(self):
      #None -> None

      # Initial guess
      x_i = [self.alpha[i]/sum(self.alpha) for i in range(len(self.alpha))]

      # Burning in
      for i in range(10000):
         x_i = self.Metropolis(x_i)

      thetas = [x_i]
      for i in range(self.n):
         x_j = self.Metropolis(thetas[-1])
         thetas.append(x_j)

      self.thetas = thetas
```

For this method, we are using the Metropolis acceptance criteria that involves, based on an current point inside the desired dominion, create a proposed new point that has distance based on another point distributed as a n-dimensional Multivariate Normal distribution with mean 0 and covariance matrix as the one defined on the *init* method. If this covariance matrix has too low values, then the proposed new points will be closer to the current point providing a limited distributed distribution, while a covariance matrix with too high values will return proposed values too distant from the current point. Then, we should accept the proposed point with probability $\alpha(x_i, x_j)$. Therefore, the pseudo-code is as follows: - a current point of the trajectory is given as x_i
- a proposed new point is defined as $x_j = x_i + y\ N(0, \Sigma)$
- if x_j is outside the desired dominion, redo the previous step until a suitable x_j emerges
- accept x_j as the next step with probability $min(1, f(x_j)/f(x_i)$
- store the next step (if approved, x_j, else x_i) and move

```
def Metropolis(self, x_i):
      # Receive x_i current point
      # at the trajectory and returns
      # the next point of the trajectory
      # that may or may not be the same
      # based on the Metropolis acceptance
      # criterion.

      def alpha(x_i, x_j):
```

```
        return min(1, self.f(x_j)/self.f(x_i))

    def is_inside_dominion_fn(x_j):
        # n-d vector -> boolean
        # checks if n-dimensional point
        # x_j is inside Theta dominion
        if sum(x_j) > 1: return False
        for i in range(len(x_j)):
            if x_j[i] <= 0:
                return False
        return True

    # Variable declaration to start loop. If first try
    # obeys dominion limits, it only runs once. Otherwise,
    # it runs until it gets a point inside the dominion.
    is_inside_dominion = False
    # Guarantees that x_j is inside the desired dominion
    while is_inside_dominion == False:
        y = np.random.multivariate_normal(self.multivariate_mean,
            self.covariance_matrix)
        x_j = [x_i[i] + y[i] for i in range(len(x_i))]
        is_inside_dominion = is_inside_dominion_fn(x_j)

    u = np.random.uniform()
    if u > alpha(x_i, x_j):
        # Reject
        x_j = x_i
    return x_j
```

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/EP5 MAP2212 (main *% u=)
   $ python3 ep5.py
U(0) = 0
U(1) = 0.03684591368227732
U(0.5) = 0.0161623507805326
```

```
U(15) = 0.8881590449954087
U(20) = 1
Time taken: 414.6862452030182 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).
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. About the Metropolis implementation, it yielded a much slower runtime (over 400 seconds while EP4 was running below the 15 second threshold) and a empirically less precise one.
While implementing, we found out that the initial guess is much more influential than guessed at first glance and could be an improved opportunity if desired (providing different initial guesses could yield more precise overall results). Also, based on ROBERTS, ROSENTHAL, 2001, we didn't achieved a desired acceptance rate of 23% (ours is close to 57%), meaning that our algorithm is accepting more new proposed points that what the found references says it is optimal. While we didn't succeed in improving this, the algorithm may be precise enough regardless.

References

ROBERTS, G. O.; ROSENTHAL, J. S. Optimal scaling for various metropolis-hastingsalgorithms.Statistica
Science, v. 16, n. 4, p. 351–367, 11 2001