walkr

by David Kane, Andy Yao

Abstract The `walkr` package samples points using random walks from the intersection of the N simplex with M hyperplanes. Mathematically, the sampling space is all vectors x that satisfy $Ax = b$, $\sum x = 1$, and $x_i \geq 0$. The sampling algorithms implemented are hit-and-run and Dikin walk, both of which are MCMC (Monte-Carlo Markov Chain) random walks. `walkr` also provide tools to examine and visualize the convergence properties of the random walks.

Introduction

A and b represent the system of equations that we have above. Specifically, A is a $M \times N$ matrix (M variables and N constraints), and b is a $M \times 1$ vector.

Mathematical Background of Sampling Space

In this section, we go through the mathematical background needed to understand the space from which we are sampling – the intersection of the N simplex and hyperplanes. Specifically, we go through a few examples as well as some linear algebra tricks. The reader does not need to read this section in order to use our package or understand our sampling algorithms. However, this section should help the reader understand better what the sample space is both geometrically and mathematically.

Definition: The N-dimensional unit simplex is described by:

$$x_1 + x_2 + x_3 + ... + x_n = 1$$
$$x_i \geq 0$$

Sampling space: simple 3D case

Let's begin with the simplest case – one linear constraint in 3 dimensional space.

$$x_1 + x_3 = 0.5$$

We can express this in terms of matrix equation $Ax = b$, where:

$$A = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}, \quad b = 0.5, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

In addition, we require the solution space to be intersected with the 3D simplex:

$$\sum x_i = 1$$
$$x_i \geq 0$$

In the following graph, we draw the intersection of the two. The orange equilateral triangle represents the 3D simplex, and the blue rectangle represents the plane $w_1 + w_3 = 0.5$. The intersection of the hyperplane (blue) with the simplex (orange) is the red line segment, which is our sampling space.

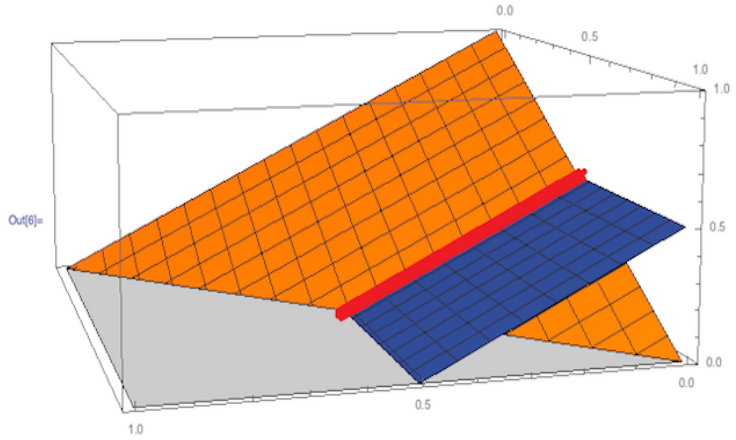

Figure 1: Intersection (red line) of Simplex and 2D hyperplane living in 3D space

Matrix Representation of Hyperplanes

Every hyperplane is described by one linear equation. Thus, a system of linear equations is the intersection of hyperplanes. In general, if we have M linear equations and N variables, then $Ax = b$ would look like:

$$A_{M\times N} = \underbrace{\left[\begin{array}{c} \\ \cdots \\ \\ \end{array}\right]}_{\text{N columns (variables)}} \Big\} \text{M rows (constraints)}$$

$$b = b_{M\times 1}, \quad x = x_{N\times 1}$$

Going from $Ax = b$ and the unit-simplex to $Ax \leq b$

Our sampling space is represented by equalities $Ax = b$, $\sum x = 1$, and non-negativity constraint $x_i \geq 0$. Because of the inequality, our sampling space is bounded (i.e. has finite volume in $\mathbb{R}^N$). More formally, our sampling space known as a **convex-polytope** in $\mathbb{R}^N$, which could be described by a generic $Ax \leq b$. Here, we describe a transformation which takes us from the intersection of $Ax = b$ and the unit-simplex to a large matrix inequality of the form $Ax \leq b$.

First, note that the equality part of the simplex constraint could be added as an extra row in $Ax = b$

$$A = \begin{bmatrix} & \cdots\cdots & \\ & \cdots\cdots & \\ & \cdots\cdots & \\ 1 & 1 \quad \cdots\cdots \quad 1 & 1 \end{bmatrix}, \quad b = \begin{bmatrix} \\ \cdots \\ \\ 1 \end{bmatrix}$$

Second, to find the complete solution to the new $Ax = b$ (i.e. the set of all possible x's that satisfy $Ax = b$), we must find the Null Space Basis of A, the set of all possible x's that satisfy $Ax = 0$, then add on a particular solution to $Ax = b$.

Mathematically, if the original A was $M \times N$, then after adding on the extra row from the simplex, the set of basis vectors which span the Null Space of our new A will be:

$$v_1, \quad v_2, \quad v_3, \quad \cdots\cdots \quad , \quad v_{M-(N+1)}$$

Third, using any particular solution, $v_{particular}$, the complete solution to the new $Ax = b$ will be

$$\left\{ v_{particular} + \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 + ... + \alpha_{M-(N+1)} v_{M-(N+1)} \quad | \quad \alpha_i \in \mathbb{R} \right\}$$

Lastly, we tag on the $x_i \geq 0$ constraints, and with some algebraic manipulations:

$$v_{particular} + \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 + ... + \alpha_{M-(N+1)} v_{M-(N+1)} \quad \geq \begin{bmatrix} 0 \\ 0 \\ ... \\ ... \\ ... \\ 0 \end{bmatrix}$$

$$\alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 + ... + \alpha_{M-(N+1)} v_{M-(N+1)} \quad \geq -v_{particular}$$

$$V\alpha \geq -v_{particular}, \quad \text{where:} \quad V = \begin{bmatrix} v_1 & v_2 & ... & v_{M-(N+1)} \end{bmatrix}, \quad \alpha = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ ... \\ \alpha_{M-(N+1)} \end{bmatrix}$$

And finally, we obtain the mathematical description of our sampling space in the form $Ax \leq b$.

$$-V\alpha \leq v_{particular}$$

Note that we have performed a **transformation** from "x-space" (coordinates described by $x_1, x_2, ..., x_N$) to "α-space" (coordinates described by $\alpha_1, \alpha_2, ...$). However, the geometric object described is still the same one. In fact, in `walkr`, when the user inputs A and b for $Ax = b$, the package internally performs this transformation, samples the α's, maps them back to "x-space, and then returns the sampled points.

The reader need not be concerned with this transformation affecting the uniformity or mixing properties of our MCMC sampling algorithms. This is because the transformation above is an affine transformation, which preserves uniformity. Simply put, sampling in either space is equivalent.

Random Walks

Now that we've understood that the intersection of $Ax = b$ with the unit-simplex is a convex polytope, and that we could describe the polytope with $Ax \leq b$, we are ready to dive into the core of `walkr` – MCMC random walks.

Starting Points

MCMC random walks need a starting point, x_0, in the interior of the convex polytope. `walkr` currently generates starting points using a linear programming method implemented by `limSolve`. Specifically, the `lsei` function of `limSolve` finds x which:

$$\begin{aligned} \text{minimizes} \quad & |Cx - d|^2 \\ \text{subject to} \quad & Ax \leq b \end{aligned}$$

Thus, we randomly generate C and d obtaining x which satisfy $Ax \leq b$. We discovered that the x's generated this way fall randomly on the boundaries of our convex polytope, due to the minimizing property of linear programming. Thus, we repeat this for say, 10 times, and then take an average of the x's generated. This averaged point is x_0, our starting point.

Hit-and-run

The hit-and-run algorithm is as follows:

1. Set starting point x_0 as current point
2. Generate a random direction $\vec{d}$. If we are in N dimensions, then this random direction will be a vector of N components. Specifically, this random direction is a uniformly generated vector on the N dimensional unit-sphere
3. Find the chord S through x_0 along the directions $\vec{d}$ and $-\vec{d}$. Specifically, we find can find end points s_1 and s_2 of the chord by going through the rows of $Ax_0 \leq b$ individually, setting the inequality to equality (so we hit the surface). Then, we can parametrize the chord along x_0 by $s_1 + t(s_2 - s_1)$, where $t \in [0,1]$
4. Pick a random point x_1 along the chord S by generating t from `Uniform[0,1]`
5. Set x_1 as current point
6. Repeat algorithm until number of desired points sampled

Here is a picture of the hit-and-run algorithm:

Dikin Walk

Preliminary Definitions

Recall, our solution space is a convex polytope. Let's call this polytope K, which we know can be described in the form $Ax \leq b$.

For the definitions below, let a_i represent a row in A, x_i, b_i represent the i^{th} element of x and b. Also recall that A is a $M \times N$ matrix.

Log Barrier Function ϕ:

$$\phi(x) = \sum -\log(b_i - a_i^T x)$$

With some calculus and algebraic tricks, we can compute and simplify the Hessian of the Log Barrier:

Hessian of Log Barrier H_x:

$$H_x = \nabla^2 \phi(x) = ... = A^T D^2 A$$
$$D = diag(\frac{1}{b_i - a_i^T x})$$

For theoretical reasons we won't get into (and don't fully understand), ϕ and $H(x)$ is very important, as they have nice properties in the context of constrained sampling and optimization. **Note:** H_x is a linear operator. Specifically, it is a $N \times N$ matrix. D is a diagonal matrix.

Definition - Dikin Ellipsoid $D^r_{x_0}$

$D^r_{x_0}$, the Dikin Ellipsoid centered at x_0 with radius r is defined as:

$$\{y \quad | \quad (y-x)^T H_{x_0}(y-x) \leq r^2\} \text{ , where } y \text{ is a point in the Ellipsoid.}$$

A important point to note here is that the shape of the Dikin Ellipsoid is a function of A, b, and x_0. In other words, if we think in terms of running a MCMC chain within our polytope K, the Dikin Ellipsoid is able to reshape itself accordingly as it surveys through the polytope.

Algorithm `Dikin`

1. Begin with a point $x_0 \in K$. This starting point must be in the polytope.
2. Construct D_{x_0}, the Dikin Ellipsoid centered at x_0
3. Pick a random point y from D_{x_0}
4. If $x_0 \notin D_y$, then reject y (this condition is counter-intuitive, read it closely)
5. If $x_0 \in D_y$, then accept y with probability $\min(1, \sqrt{\frac{det(H_y)}{det(H_{x_0})}})$ (the big picture is that the ratio of the determinants are equal to the ratio of volumes of the ellipsoids centered at x_0 and y. Thus, the geometric argument would be that this way the Dikin walk can avoid extreme corners of the region)
6. repeat until obtained number of desired points

How to pick a random point uniformly from a Dikin Ellipsoid?

Let's say, we now have D_x^r, the Dikin Ellipsoid centered at x with radius r. Sampling from the ellipsoid is mathematically equivalent to generating a Standard Gaussian vector, projecting that vector on the unit ball, and then solving for a matrix equation with an unique solution. The exact algorithm is as follows:

1. generate ζ from the n dimensional Standard Gaussian (i.e. `zeta = rnorm(n,0,1)`)
2. normalize ζ to be on the n dimensional ball with radius r, that is:
$$\zeta =< x_1, x_2, ..., x_n > \quad \rightarrow \quad < \frac{rx_1}{\sqrt{x_1^2+x_2^2+...+x_n^2}}, \frac{rx_2}{\sqrt{x_1^2+x_2^2+...+x_n^2}}, ..., \frac{rx_n}{\sqrt{x_1^2+x_2^2+...+x_n^2}}$$
3. Solve for d in the matrix equation $H_x d = A^T D\zeta$ (note, as long as x_0 is not on the boundary of our polytope K, H_x will be non-singular, thus, d will always be unique)
4. $y = x_0 + d$ is our randomly sampled point from D_x^r

Important Theorem and Findings

With the algorithm, there is no mention of what happens if the point y we accept is outside of our polytope K. Luckily, there is no need to worry about that because of this theorem:

Theorem – If $x_0 \in K$, then $D_{x_0}^1 \subseteq K$. That is, if our starting point x_0 is in our polytope K, then the Dikin Ellipsoid centered at x_0 with radius 1 will always be contained in K.

This is important because this way, we know for sure that if we set $r = 1$, then our algorithm will never sample points from outside the polytope K. Although there are still two rejection components to the algorithm (see above), the rejection rate is much higher than expected because of this theorem.

Key points about the Dikin Algorithm

- Dikin algorithm does not produce an uniform sample asymptotically. According to the authors of the original paper, the sampling is "nearly uniform" in K. It is unclear to us on how "nearly" they mean. We are looking for papers that quantify this. Either way, it doesn't look that bad in a lower dimensional case (see plot in section below)
- We cannot start Dikin at $x_0 \notin K$ or on the boundary of K. This will cause the Hessian, log-barrier, and determinants to blow to infinity.
- Dikin mixes "strongly in polynomial start" only if we begin at an "analytical center". It is not very clear to us what exactly the authors mean by an "analytical center", but so far we've tried to compute an average of the vertices. However, this problem doesn't scale well. This is because according to the author who came up with the implementation to find the vertices, the run-time complexity is $O(n^2dv)$, where d is the dimension of the polytope, n is the number of hyperplanes, and v is the number of vertices. We have some evidence that the number of vertices grow combinatorially as we increase d.
- we know any $r \leq 1$ guarantees that our ellipsoids are bounded within K. However, from trials it seems like we could increase r up to a certain value for each polytope. The virtue of doing so would be to "cover more space", thus probably mixing even better

Using `walkr`

Examining/Visualizing Results

Conclusion

Authors

David Kane
Managing Director
Hutchin Hill Capital
101 Federal Street, Boston, USA
dave.kane@gmail.com

Andy Yao
Mathematics and Physics
Williams College
Williamstown, MA, USA
ay3@williams.edu