Darts

Max Caragozian

April 21, 2025

The tap-room fire is alight again
and a new stock of darts laid in;
serviceable and well-feathered,
that fly true and will see us into
another spring.

"A Chiltern Autumn" [4]

1 Introcudction

When I moved to Rochester this past January, I bought a dartboard and hung it in the basement of my college house. It ended up being a worthwhile purchase, both as a way to get to know my new roommates and as time killer for what ended up being a long, snowy winter. I started reading into the mathematics of darts and came across the paper "A Statistician Plays Darts"[2], which quantifies how a dart thrower's skill affects where he/she should aim.

In the simplest model, the authors assume that a person throws with a symmetrical bivariate normal distribution. They estimate the function $\mu^*(\sigma)$ that takes in a standard deviation σ and outputs a point μ that maximizes the expected score of a dart thrown with distribution $\mathcal{N}(\mu, \sigma^2)$.

Bad sentence -¿ Unsurprisingly, the authors found that a person with a very small standard deviation should aim for the triple 20 (T20), as it has the highest point value on the board. The optimal aiming location stays in the T20 until approximately $\sigma = 16.9$mm, where the optimal location jumps to the T19, before eventually moving towards the center of the board.

At the outset of this project, my aim was to replicate some of the results of "A Statistician Plays Darts."[2] Along the way, I found that there is more than one way to get there and learned much about global optimization algorithms. This paper documents the several methods I tried, how I implemented them, and how each method 111

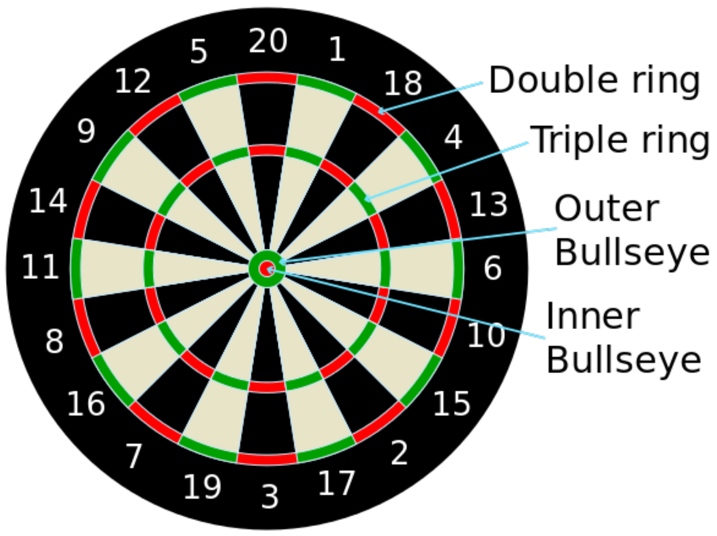

Figure 1: Standard dartboard with point values [3]

2 Integration Brute Force

Before trying to find $\mu^*(\sigma)$, I wrote code that would find the expected score for someone throwing a dart with a given random distribution. We define a *sector* as a continguous area of the dartboard with a single point value. Examples are the triple 20, the double bullseye, or the portion of single 19 that lies between the outer bullseye ring and the inner triple ring. Let S be the set of all sectors and X be a distribution with PDF f_x. To find the expected score of a distribution we compute:

$$E[score(X)] = \sum_{s \in S} \iint_s score_s \cdot f_x(x, y) dxdy. \tag{1}$$

I used SciPy's *nquad* function to perform the integration. Originally, I used the *dblquad* method, which is specific to double integral, but I ran into a rounding issues when f_x was very small. The *nquad* function allows finer control and fixed the issue.

Turning now to the specific case of symmetrical bivariate normal distributions, let us define the function that takes in a point μ and returns the expected score of a dart thrown with distribution $\mathcal{N}(\mu, \sigma^2)$ as

$$F(\mu|\sigma). \tag{2}$$

With the code implementation of Equation 1, I could calculate $F(\mu|\sigma)$for a grid of possible μ values across the dartboard. I could then both plot the grid as a heatmap and return the maximum calculated value as a numerical approximation for $\mu^*(\sigma)$.

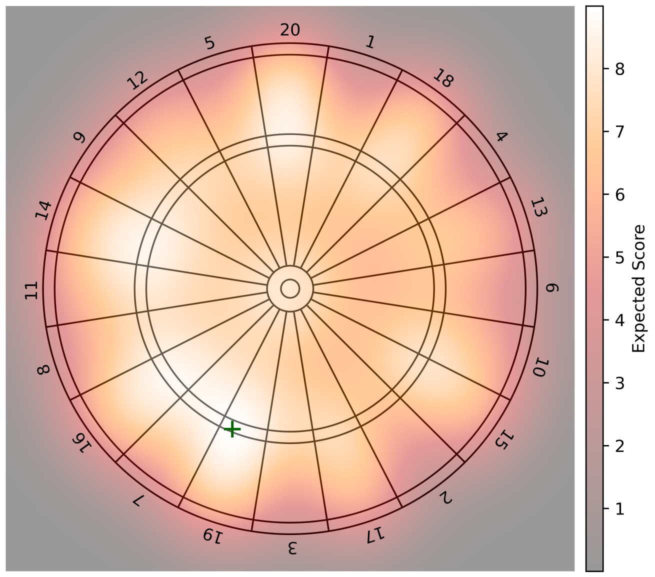

Figure 2: Heatmap for $\sigma = 26.9$. The green + is the location of μ^*

For each pixel in the heat map (i.e. each evaluation of $F(\mu|\sigma)$), I needed to compute 82 double integrals: one for each sector of the dartboard. This was painfully slow, and I spent a lot of time trying to optimize the code by testing the value of the PDF at certain values across the sector so I could skip integrating over sectors that would contribute very little to the final expected score. After a lot of time tinkering with the threshhold for rejecting a sector and how to distribute test points, etc., I came to the conclusion it was easier just to compute every integral. The speedup from my optimization was minimal, and I couldnt escape the fear that by skipping certain integrals I might materially affect the final calculation.

3 Basin-Hopping

I looked for faster ways to compute $\mu^*$for a fixed σ. We can think about the computation as a global maximization problem for a function of function of two inputs (the x and y coordinates of μ), that is

$$\mu^*(\sigma) = \max[F(\mu|\sigma)] \tag{3}$$

By convention, global optimization algorithms find the *minimum* of a function, so in practice we compute

$$\min[-F(\mu|\sigma)]. \tag{4}$$

SciPy has bindings for several different global minimization algorithms. I ended up settling on a stochastic algorithm called *basin-hopping.*

Each iteration of basin-hopping starts with a point (the "currently accepted point"). The algorithm then takes a jump of a random size in a random direction. After landing at a new point, the algorithm moves "downhill" to a local minimum and compares the new local minimum with the currently accepted point. If the new minimum is lower than the previous one, it is always accepted. If it is not lower, there is a random process to decide whether or not to accept it, with lower minima being more likely to be accepted. The process then repeats, with another random perturbation, a local minimization, and a comparison with the most recently accepted point. Over time, random jumps get smaller, and the algorithm ends when a point has remained unchallenged for some number of cycles. The distribution of jump sizes, the local minimization algorithm, and the acceptance criteria for new points can all be controlled by parameters [1].

Figure 3 shows a visualization for one run of the basin hopping algorithm We should note that basin-hopping is not guaranteed to find the global minimum. If the random jumps never find the right "basin", the algorithm will return the wrong answer. 2.

While the dartboard problem proved a good way to illustrate basin-hopping, basin-hopping was not a good way to solve my dartboard problem. There were two main issues, the of which first I have already noted: basin hopping is not guaranteed to find the true global minimum. The other problem is that basin-hopping still relies on the slow process of integration for each point that it tests. To be sure, it is still faster than brute force. The basin-hopping run in Figure 3 needed only 134 evaluations of $F(\mu|\sigma)$, compared to 9000 for a 300×300 heatmap. But it was still slow and sometimes gives the wrong answer, so I decided to go back to square one.

4 Don't Let Ten Minutes of Reading Save You From Ten Hours of Coding

I reread "A Statistician Playes Darts"[2] and realized the authors came up with a much faster way to compute $F(\mu|\sigma)$ that I missed the first time I read the paper. The authors derived that

$$F(\mu|\sigma) = (f_{0,\sigma^2} * score)(\mu), \tag{5}$$

that is the convolution between the PDF of $\mathcal{N}(0, \sigma^2)$ and the score value for each point on the dartboard[2]. I confess I can't quite wrap my head around how this works and for the scope of this paper I will accept it as magic. Even more magical is that one can compute the convolution extremely quickly with

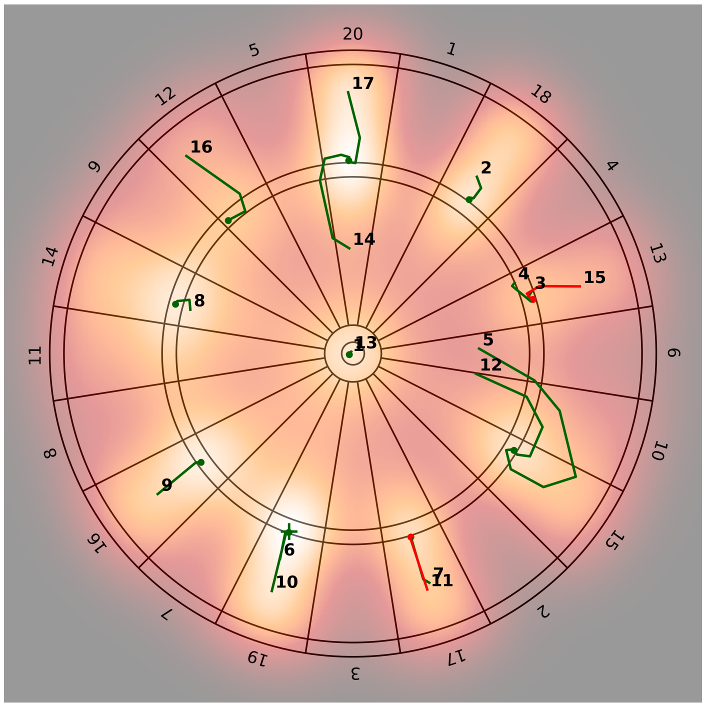

Figure 3: Each run of the local optimizer for a basin-hopping run to find $\mu^*(\sigma = 19mm)$. Points that were *accepted* by the algorithm are green while those that were *rejected* are red. The green + is the true location of μ^*.

an algorithm that uses a Fast Fourier Transform. Generating the data for a 300 by 300 heatmap such as that in Figure 2 took over an hour by integration, but SciPy's *fftconvolve* function gets the job done in seconds.

The paper [2] also notes that previous researchers have approximated the same results using a monte-carlo method. To round off my survey of methods for computing μ^*, I implemented the monte-carlo method. It ended up being considerably faster than direct integration even when taking many samples for each function evaluation.

5 Method Comparison and Timing

We can sum up the methods for computing in a table:

	Deterministic	**Stochastic**
Compute Whole Grid	Convolution, Integration	Monte Carlo
Compute Single Point	-	Basin-Hopping

For

References

[1] The Scipy Community. *scipy.optimize.basinhopping*. URL: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.basinhopping.html#scipy.optimize.basinhopping`.

[2] Andrew Price Ryan Tibshirani and Jonathan Taylor. "A Statistician Plays Darts". In: *Journal of the Royal Statistical Society Series A* 174 (2011), pp. 213–226.

[3] Tijmen Stam. *Dartboard diagram.svg*. 2014. URL: `https://commons.wikimedia.org/wiki/File:Dartboard_diagram.svg`.

[4] *The Spectator*. v. 143, nos. 5271-5296. The Spectator, 1929. URL: `https://books.google.com/books?id=mniapACANGEC`.