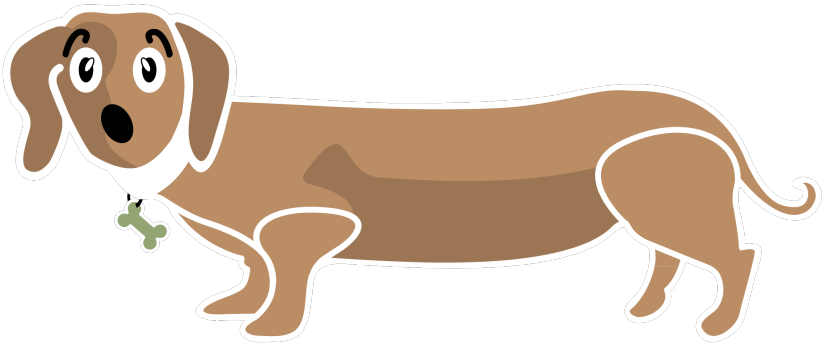
D/oxy

Symbolic regression for the browser.



What is it?

D/oxy is a **for-fun**, free as in beer, homemade symbolic regression program that runs on your own puny local CPU. You give D/oxy a set of data (like a spreadsheet with a bunch of columns) and it tries to randomly evolve expressions that can accurately predict one column by looking at the others.

D/oxy is written in Javascript and is, therefore, much slower to arrive at answers than programs with similar aims like Eureqa or TuringBot, which now run in the cloud. On the other hand, D/oxy is free, and surprisingly quick.

History: Why did I write this?

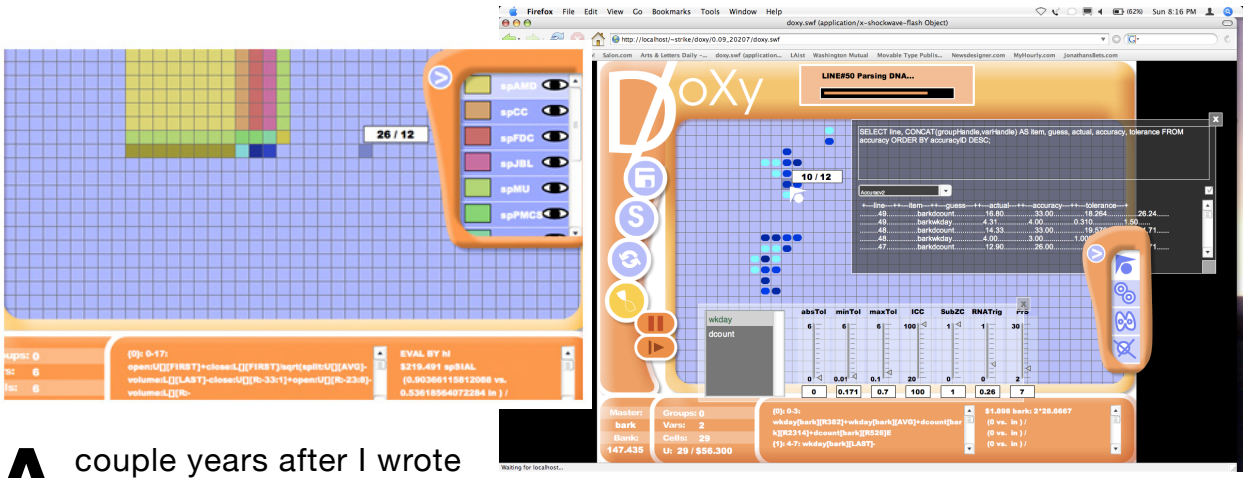
Silly rabbit. I wrote it to crack the stock market, predict earthquakes, and improve my sportsbetting average. (It only became clear that it doesn't *really* work for any of those things after I wrote it. See: Overfitting. Mainly now it is just for fun. Or maybe that's just what I want you to believe.)

The real history is, back in 2004, years before I ever heard the term *symbolic regression*, I was fascinated with building procedurally generated games (like my favorite game as a kid) and creating simple rules on paper that could evolve into complicated, emergent behaviors. I was making most of my games and business apps in Flash and Flex at the time, so I wrote the first version of this thing called D/oxy in Actionscript 2 and PHP. The PHP code did most of the heavy lifting. It fetched data out of a mysql database, and used SOAP calls to distribute the evolutionary algorithms to multiple servers, evaluate them, and then return the data at each iteration to the Flash front-end. I only had two actual servers at the

time, but I imagined that D/oxy would need *thousands of servers*, so writing a whole SOAP architecture seemed pretty awesome. Yeah, I was young and dumb.

Looking back, D/oxy1 actually had some very interesting features I'd like to try again. It was very visual. Although arguably we would say now that its logic was a bit too bound to a certain graphical/spatial construct to be flexible or effective as a scientific tool, maybe that wasn't a bad thing. It did lead to some neat output.

The action in D/oxy1 took place on a large grid, like Conway's Game of Life. The grid could be layered with colorful substrates of "sugar" - each layer represented a dataset over time, such as a different stock with its volume and price and other daily data. Then "bacteria" would spawn on the grid. Each bacterium would have some "DNA" that was a set of expressions it would run to try to accurately model and consume the data under it, at each tick of the clock. The best DNA would remain activated, while DNA that failed would be ignored and eventually discarded. Bacteria gained and lost "energy" points based on how ***accurate*** they were in getting the right result for the ***target column*** in the data at each row. If they ran out of energy, they died. But if they gained enough points, they could clone themselves into an adjoining grid cell, if one was empty, or trade DNA strands with the bacteria in the next cell if it wasn't. The adjoining grid cell might have a totally, or slightly different "sugar" substrate of data; perhaps a different set of stocks from the same market sector, blending into a different market sector a little further over. A bacterium would have to adapt to the new conditions or die.



Some screenshots from the original D/oxy, sometime around 2007. One of the weirdest things I was able to get out of the it, that may have had to do with its spatial nature, was a prediction a week in advance of a major earthquake in Chile based only on the positions of other recent ones around the Pacific Rim. I'm not kidding.

A couple years after I wrote D/oxy1 and had been messing around with earthquake data and stocks, mostly pointlessly, a program called Eureqa was published free-to-use by a group at Cornell University. I was blown

away by it. It did essentially the same thing, but much, much faster (even in Windows emulation on a Mac), and with the much more direct approach of simply having a huge list of expressions competing to model one big piece of data. I hadn't thought of trying that, but I was impressed at how well it seemed to work. This was clearly built by and for serious people, which I am not. I had mostly abandoned D/oxy by that point, but I spent awhile messing around with Eureqa and returned to it from time to time, to test out ideas. I figured the problem had been solved. Unfortunately, (or fortunately for them!) that college project became a for-profit company, killed off the offline shareware version of their software, and put the computation into the cloud. Then they started charging absurd sums of money to use it. Who could be using this? I wondered. Is this bizarre little trick of evaluating random math actually worth millions of dollars to someone? Because ...frankly, I couldn't see it.

About 15 years went by and I mostly forgot about the D/oxy experiment, and about genetic algorithms / symbolic regression in general. One day though, during the pandemic, I was looking through piles of old Flash programs and lamenting that I could never compile them again. At the same time, I'd been playing with writing a common worker pool library for sharding big computations to Nodejs workers and webworkers, and I decided it was worth resurrecting D/oxy in a quicker and dirtier way with Typescript. The first of these, D/oxy2, had basically no graphical UI. What you see here is D/oxy3. Graphical and free.

The essence of the original Flash and PHP-based, D/oxy1 recipe for evolving polynomials is mostly preserved in this (and enhanced with logical functions), but I took the Eureqa approach for now and just made it a single stack of competing expressions.

Now that I think the tools and horsepower exist on the desktop to do what I intended with the original D/oxy, I'd like to keep (de)volving it back toward the original idea with all its odd spatial directions. But for now, here it is.

Basic usage.

Import a CSV file with some columns of data. Your CSV must have a top row with column names. One of these will be your ***target column*** that you're trying to predict; each expression will test its results against that column. No expression is allowed to *use* the target column to *figure out* the target column (although, strange things will happen if you switch the target column in the middle of a run).

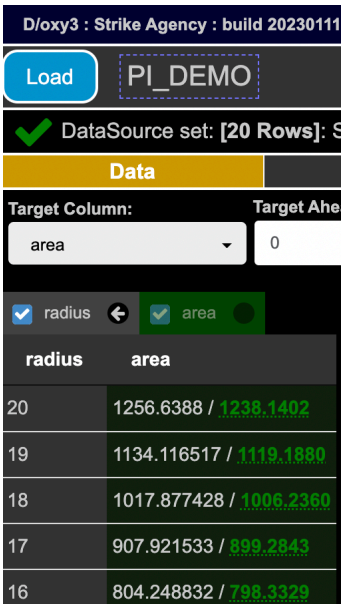


Image: This dataset shows the area of a circle with a given radius. The ***target column*** is set to "area" and you can see it's green and its checkbox is lit and grayed out. The blue checkboxes tell D/oxy whether or not to use the column at all. Note the small white arrow in a circle pointing at "radius". That means ***lookbacks are available***. More on that in a second.

The numbers in white under *radius* and *area* are the basic data from the CSV we imported. The numbers in green beside *area* are D/oxy's best approximation, as it's running, based on the most accurate polynomial it's evolved from the non-target columns (in this case, from *radius*). When it's not running, you can roll over those green numbers (with a mouse, on a computer - you're not reading this on a phone are you?) and see which expressions scored highest. You can then go into the **Monitor** section and pin them, if you like 'em so much.

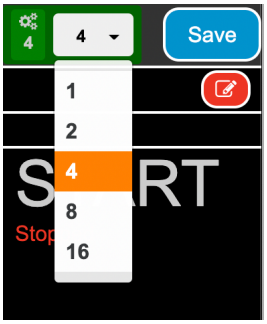
Next to the **Target Column** dropdown, there's a field for **Target Ahead**. If you set ***Target Ahead*** to 3, D/oxy will create 3 fake rows at the end of your data, to magically predict the future. Instead of evaluating the current row, it will evolve expressions that take whatever row it's looking at and try to guess the target column of the row 3 after it. Doing this also means that the first 3 rows of your data at the bottom are being used to generate results, but don't *get* results themeslves. They are to be ignored. Which brings us to two important special values in D/oxy: ***N*** and ***F***.

D/oxy can't predict this row because the best expression it's found requires 3 rows of prior data to compare with, and since this is only row 3 it doesn't have enough rows before it to make a prediction.

N means a row may be mined for its data, but it's not being evaluated. That's either because you set ***target ahead*** or because the expression in question has evolved to ***look back*** and this particular row doesn't have enough rows before it to look back at. Either way, ***N*** means not enough information to evaluate.

F means a row is in the future. If you want to leave gaps in your target column data, don't leave them blank; write the letter F in them. If D/oxy sees a null or blank in the target column, it might evaluate it as meaning zero. It will then think its prediction was way far off. Write ***F*** in an otherwise numerical column to

indicate that you want D/oxy to *not test itself against this particular result*, but simply to predict it.



Okay, I added a CSV.

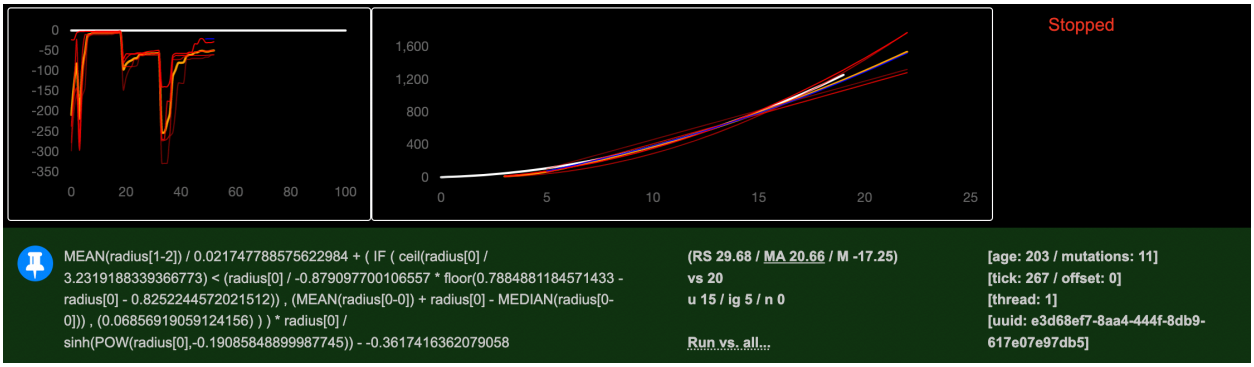
Cool. Now hit that big ugly thing that says **START** and have a go. The dropdown box above it shows how many threads/workers you're going to use. ***Please don't set it to more than the number of cores you have or your computer will go boom.*** The number to the left of the dropdown shows how many workers your file *has previously been sharded to*. If you add more workers than that, it'll clone your best results out to the extra workers. If you reduce the number of workers below it, it'll eliminate some of your results and round-robin the best ones to the available threads. Best results from one thread are also passed around between workers as you run, on a rolling basis. **See that save button?** The ***Load*** and ***Save*** buttons let you keep all your results. They also save a copy of your original data inside the JSON formatted file, because we're not running this on a server.

There's a NodeJS-based version of this, that takes SQL queries and sqlite files. It's not what you're looking at. So this version saves all your big data in each file with the solutions instead of fetching it on the fly. Sorry. But you don't want to upload all that proprietary data to my server, and I sure as hell don't want to host it, so we'll have to live with this arrangement.

Monitoring and pruning.

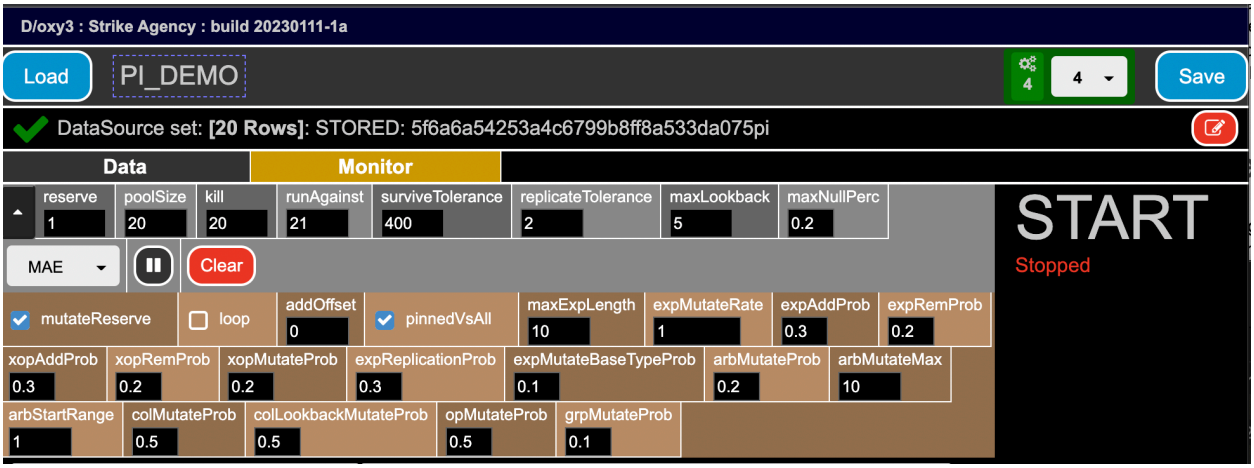
There are probably a lot of ways I could optimize this thing by *automatically* tweaking survival tolerances, throttling it if your CPU is hot enough to fry eggs on, etc. I haven't done any of those. Instead, you have the **Monitor** section**.**

The **Monitor** section shows all the polynomials you've evolved, and lets you interact with the worker threads in semi-realtime. That is to say, it *broadcasts* your changes when you make them, but the workers might already be working on the next line of the solution.

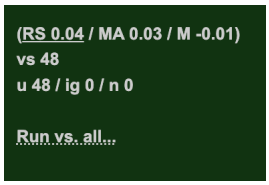


*Red lines are the best individual solutions. Their brightness is stronger the better they are. Blue lines are solutions you have pinned, which may suck or not; it's your prerogative. The yellow line is an average. The first polynomial solution - the green area with white text under the graphs - is (1) **pinned**, because you clicked the blue pin while it wasn't running (you can't pin them while D/oxy is running, it runs too fast and asynchronously; you have to stop it to pin things). (2) its background is green which means it's protected as part of the **reserve on its thread,** which is thread 1 based on the data to the right.*

Also, if its text turns green for a second, that's because its solution just improved a little bit.

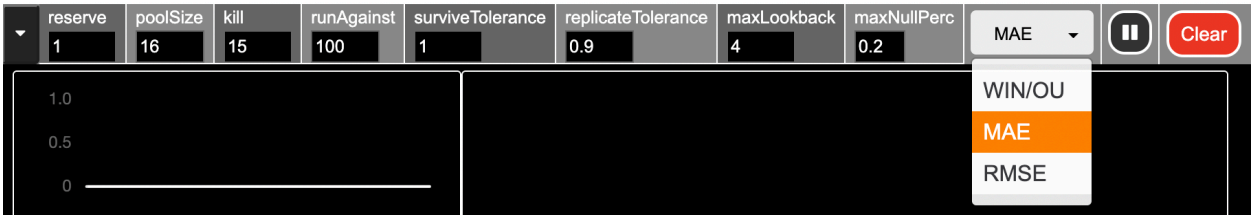


Make sure to hit the tiny arrow to the left of "reserve" so you can see all the options.



Left: Each solution shows its last accuracy for root mean square error (RS), mean absolute error (MA), and the non-absolute average mean difference (M) between predictions and desired results. The one underlined is the one that was used to evaluate. **vs. 48** means it was **run against** 48 rows. **u 48 / ig 0 / n 0** means that 48 rows were **utilized** (and averaged into the MAE), 0 rows were **ignored** as a result of lookbacks or target-ahead, and 0 rows produced **null results** (which includes division by zero / Infinity / undefined and any error cases).

Most importantly for USING D/OXY: All those fields with little numbers have to broadcast their changes to the workers. They do that when they **lose focus (click off them *after* inputing a new number)**, and also do it **every time you use the up/down arrows to change their value.** Which is to say that nothing gets updated if you type a new number and leave your cursor sitting there. That's on purpose, because we all know how easy it is to accidentally type the wrong



Changes to your preferences are sent to the running workers, which update asynchronously.

number. Please read this paragraph again at least twice before you lose the answer to the meaning of life. You've been warned.

Use the controls in the **Monitor** section to adjust how D/oxy evolves its expressions. I'm planning to add more help icons, but for now here are the most crucial bits:

> *Wait. Timeout. Hang on. Am I expected to be adjusting these values while it runs, like I'm playing a video game as I try to evolve my solution?*
>
> Yes, yes you are. You'll end up with better results more quickly if you take an active hand and guide the garden's growth. ***If you come up with a reliable algorithm for automatically adjusting these values based on other values over time, you just created an automatic gardener that I'd love to include in a future version.***

Basic Options

Let's take it from the left. I'm going to stop saying "polynomial" now and just call D/oxy's solutions "expressionss", because it's easier to type, and I dropped out of art school.

RESERVE: For every whole number, you are telling D/oxy it *must save* an expression in each thread, no matter how bad it sucks, *reserving* it when it kills off the ones that don't meet its survival criteria. The reserve expressions have a

green background in your list. You can choose to allow reserves to mutate or not in the secondary options. (You should allow it. Generally, less death and more mutation breeds better results than the opposite).

POOL SIZE: How many expressions do you want to keep for each thread? This has a huge impact on performance, obviously. Every expression in your pool will have to run against every line in your data (see ***run against*** below) before a tick can be completed.

KILL: The maximum number of expressions that can be killed on each tick. Up to this many will be killed if they're **(a)** not reserved, **(b)** not pinned, and **(c)** not within the ***survival tolerance.*** Note that lowering the pool size will also kill the worst performers, and expressions with too many null-ish/imaginary results are killed even if kill is set to zero. Setting the kill to 1 less than the pool size when nothing meets survival tolerance will still preserve the top expression, and is effectively the same as setting the reserve to 1.

RUN AGAINST: How many rows should each expression test itself against, on each tick. Note that ticks occur on workers so they're not synchronous. But each expression checks against X number of rows, averages out its success rate, and there's your accuracy score. Earlyish rows that can't be calculated because lookbacks don't count, and future rows don't get judged; ***run against*** takes these into account and doesn't penalize you for rows that are ignored; it just ignores them. Generally for a short data set, just set ***run against*** to a number as big as the number of rows, or larger (you can also ***loop***). But sometimes you want your expressions to crawl across the data and evolve over time. This gives you the option. One thing that's particularly effective is setting ***run against*** to 80% of your number of rows and then slowly increasing it. ***Pinned expressions run against the entire dataset by default***.

SURVIVE TOLERANCE / REPLICATE TOLERANCE: The concept of **accuracy** in D/oxy is that *perfect accuracy is zero*. The higher the accuracy number, the less accurate the expression is. An expression will be killed if its **accuracy** is higher than the **survive tolerance** and you have "kill" set to something above zero. In MAE or RMSE modes, accuracy is literally the mean absolute error or root mean squared error of that error, averaged over the set that the expression was just ***run against***. In "WIN/OU" mode, which means "over-under", the expression is considered a simple win or loss based on whether *its result and the target column share a plus or minus sign*. So in "WIN/OU" mode, accuracy is the inverse of the win percentage. If you have an expression "winning" 90% of the time, you need to set survive tolerance to below 0.1 to kill it.

Replication happens randomly, and only within the same thread. All replicants get at least one mutation. Every expression below the ***replicate tolerance*** is eligible to clone itself on every tick. ***expReplicationProb*** is the relevant control to how likely it is to actually replicate if it meets that tolerance.

MAXLOOKBACK: This is the global control to allow columns to compare and run group functions on their previous data. See below for more about lookbacks. Please note: Changing this to a lower value *does not modify or kill expressions in the pool that already rely on longer lookbacks.* They will still be around unless you **clear** them.

MAXNULLPERC: We talked about **N** and **F**. But what if an expression spits out *Infinity* or *NaN?* This tells us what percentage of the potentially evaluable **run against** we are willing to tolerate being imaginary numbers or absurdities before we throw out the whole expression. Why are the Lakers going to get infinity points next game? I don't know. Fuckit. We have to keep moving.

WIN/OU / MAE / RMSE: How should we evaluate how close the expression is to the predicting the target column? Mean average error or root of same are standard. **Win/OU** is a D/oxy thing that can be thought of as "yes or no": If the column is a positive number and the solution is positive, it's a win. If the column is negative and the solution is negative, it's a win. If zero, it's a push. I told you I was trying to improve my sportsbetting average...

In nature, red indicates danger.

CLEAR: Nukes all your expressions.

PAUSE: Maybe unintuitively, this doesn't stop the run. It tells D/oxy to keep running but just **stops all mutations and clones.** This is useful. Here's why. If you load a new CSV file into D/oxy that has the exact same fields as the old one, D/oxy will keep your lookback preferences on all the columns, and will not clear out your expressions. So suppose you trained a bunch of expressions on one season of the NBA. You're winning 100% of your games. You just want to see how those expressions stack up against the next season... but the problem is, when you load the next season, your expressions will start mutating right away to get better at predicting the *new* data. You don't want them to mutate, you just want to know how well *these* expressions work. So, aha. You stop D/oxy, hit the pause button, load in the new data and start D/oxy again. No mutations or replications will happen; you just run your old expressions across the new dataset and see how it performs.

Most settings with `prob` in the name...

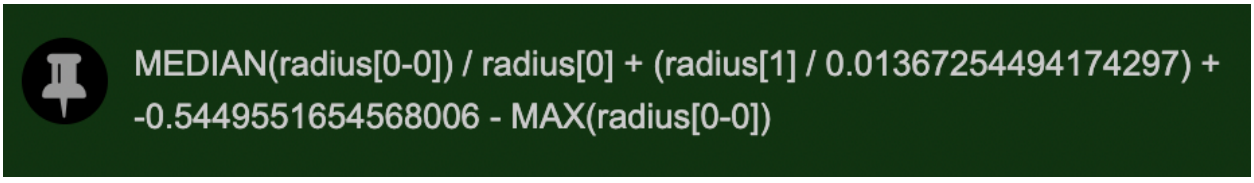
These are probabilities of certain types of mutations *being attempted* on each tick. If it's a fraction of 1, like say 0.5, a mutation will be attempted randomly 50% of the time. If you want it to happen more than once per tick, set it to a whole number larger than 1, and that's how many times this type of mutation *will be attempted* on each tick. These fields increment by 0.05 when you use the arrow keys, but decimal places on numbers larger than 1 are ignored.

How Mutations Work

When an expression mutates in D/oxy, it is guaranteed to be an improvement. For every mutation, a temporary clone is made and mutated and then compared to the original against the same **run against** dataset. If the clone doesn't perform as well as the original, it's discarded. If it improves on the original, it replaces the original expression and the original expression text ***flashes green*** in the **Monitor** section. So just because something tries to mutate on every tick doesn't mean it actually replaces itself. It only replaces itself if the mutation performs better. ***On the other, other hand, your mutation rate also affects how new clones are born, and they will survive for one tick no matter what.***

Lookbacks. (Under the Data tab).

Imagine your ***target column*** is the closing price of a stock, and the only input columns you have are the opening price and the volume. Maybe you'd generate another column to show yesterday's last closing price, and another for the moving average, and another for volatility, etc. D/oxy has the concept of ***lookbacks*** which let the genetic algorithms access historical data for your columns. D/oxy's evolved expressions are always denoted this way:



Notice the **[**square brackets**]**. They imply *how many rows back in the data we're talking about*. So he're we're trying to guess a circle's area from the radius. **radius[1]** is the radius *one row back in the data*. You can see how a group function like MEDIAN[0-3] can take a range backwards from the current point in time.

Every column in your data can be specified to have:

DEFAULT LOOKBACK. Defaults to the ***maxLookback*** specified in the monitor section.

FORCE LOOKBACK. Expressions are not allowed to use the column's *current* value to predict the target column, but they may evolve to use prior values from this column (up to the range set by ***maxLookback***). This is incompatible with setting maxLookback to zero, and the sim will protest if you do.

NO LOOKBACK. The column is prevented from ever seeking its own history as a source of guessing.

MATCH LOOKBACK. AFAIK this is unique to D/oxy, and very powerful. When you set this on a column, all the historical data is filtered to match the current value of this column. Only *then* are other columns' lookbacks applied. A good example would be a list of all the Laker games for a year. When you get to a row where the Lakers were playing the Nets, you want all your other lookbacks to point to the last time those two teams played. So you set **match lookback** on the home and away teams. Now, POINTS_HOME[1] will not refer to the last game in the Lakers' season, but the last game where the Lakers played the Nets (at home). Note that this incurs a penalty with a lot of ***N*** (not evaluated) results if you're looking back a few games and the Lakers haven't played the Nets enough times yet.

Match lookbacks are additive; *every* column you set them on must match for a row to be a valid lookback row. They themselves are often words (strings) so they're not doing anything. They control and filter out what all *other* columns are able to see when they look backwards.

Final thoughts

There are a lot of quirks and features to D/oxy I'm not covering in this document. This was never really intended to be seen or used by the public. Whenever I see people offering to sell me something that'll make me rich and cure my problems I just think, if this person could cure their own problems they wouldn't be making infomercials to sell me their system for $49.99. Y'know? Like, if I claimed D/oxy actually *solved* the NBA or the stock market, and I tried to sell it to you...

Well, so, it doesn't. And I'm not. Here you go. If you become a billionaire from it, send me a tip. I'm rooting for ya.

I do hope to get people a little more interested in these kinds of solutions. In the current excitement around AI diffusers, I think it's been forgotten how interesting genetic algorithms can be. You don't really *learn* anything from the output of diffusers. Informing you of how a solution was derived is, unfortunately, not a feature of modern AI. **Genetic algorithms are interesting not because they magically spit out the solution to a big dataset, but because if you read the expressions closely, you can sometimes infer correlations you wouldn't have otherwise noticed, suggesting new ways to think about complex systems.**

Do with it what you will.

****** If you would like to contribute to this project or talk about this stuff, or work on something incredibly fun & interesting (not for a big tech company), like a procedural world full of vicious little animals that evolve to eat other little animals for no reason at all, or you just want to say hi, my email is:***

Josh@TheStrikeAgency.com

Attributions

D/oxy utilizes the following code packages:

Math.js, Bootstrap (with tether.js), jQuery, DataTables, FileSaver.js, moment.js, tooltipster, require.js, and pretty-checkbox.css.