Evolutionary Dataset Optimisation: learning algorithm quality through evolution

Henry Wilde, Vincent Knight, Jonathan Gillard

Abstract

In this paper we propose a new method for learning how algorithms perform. Classically, algorithms are compared on a finite number of existing (or newly simulated) benchmark data sets based on some fixed metric. The algorithm(s) with the smallest value of this metric are chosen to be the 'best performing'. We offer a new approach to flip this paradigm. We instead aim to gain a richer picture of the performance of an algorithm by generating artificial data through genetic evolution, the purpose of which is to create populations of datasets for which a particular algorithm performs well. These data sets can be studied to learn as to what attributes lead to a particular progress of a given algorithm.

Following a detailed description of the algorithm as well as a brief description of an open source implementation, a number of numeric experiments are presented to show the performance of the method.

1 Introduction

When an algorithm is developed to solve a given problem, the developer is presented with questions about the performance of their proposed method, and its relative performance against existing methods. The standard response to these situations is to fix some benchmark datasets and a common metric amongst the proposed method and its competitors. The algorithm is then assessed based on this metric. This paradigm has a number of problems:

1. How were these benchmark examples selected? In some domains and disciplines there are well-established benchmarks so those found through literature can be reliable, in others less so.

2. Sometimes, when there is a lack of benchmark examples, a 'new' dataset is simulated to assess the algorithm. This begs the question as to how and why that simulation is created.

3. In disciplines where there are established benchmarks, there may still be underlying problems:

   (i) As an example, work by Torralba and Efros [13] showed that image classifiers trained and evaluated on a particular dataset, or datasets, did not perform reliably when evaluated using other benchmark datasets.

   (ii) The amount of learning one can gain as to the characteristics of data which lead to good (or bad) performance of an algorithm is constrained to the finite set of attributes present in the benchmark data chosen in the first place.

This work presents a novel approach that flips this paradigm on its head. Instead of fixing data and evaluating an algorithm on some metric, the proposed method will generate data for which the algorithm performs well (or poorly) through evolution. These datasets can then be studied to learn which attributes and characteristics lead to the success (or failure) of a given algorithm.

Evolutionary algorithms (EAs) have been applied successfully to solve a wide array of problems in recent history - particularly where the complexity of the problem or its domain are significant. These methods are highly adaptive and their population-based construction allows for the efficient solving of problems that are otherwise beyond the scope of traditional search and optimisation methods. It should be clear that the study of algorithm quality falls into this category of problems.

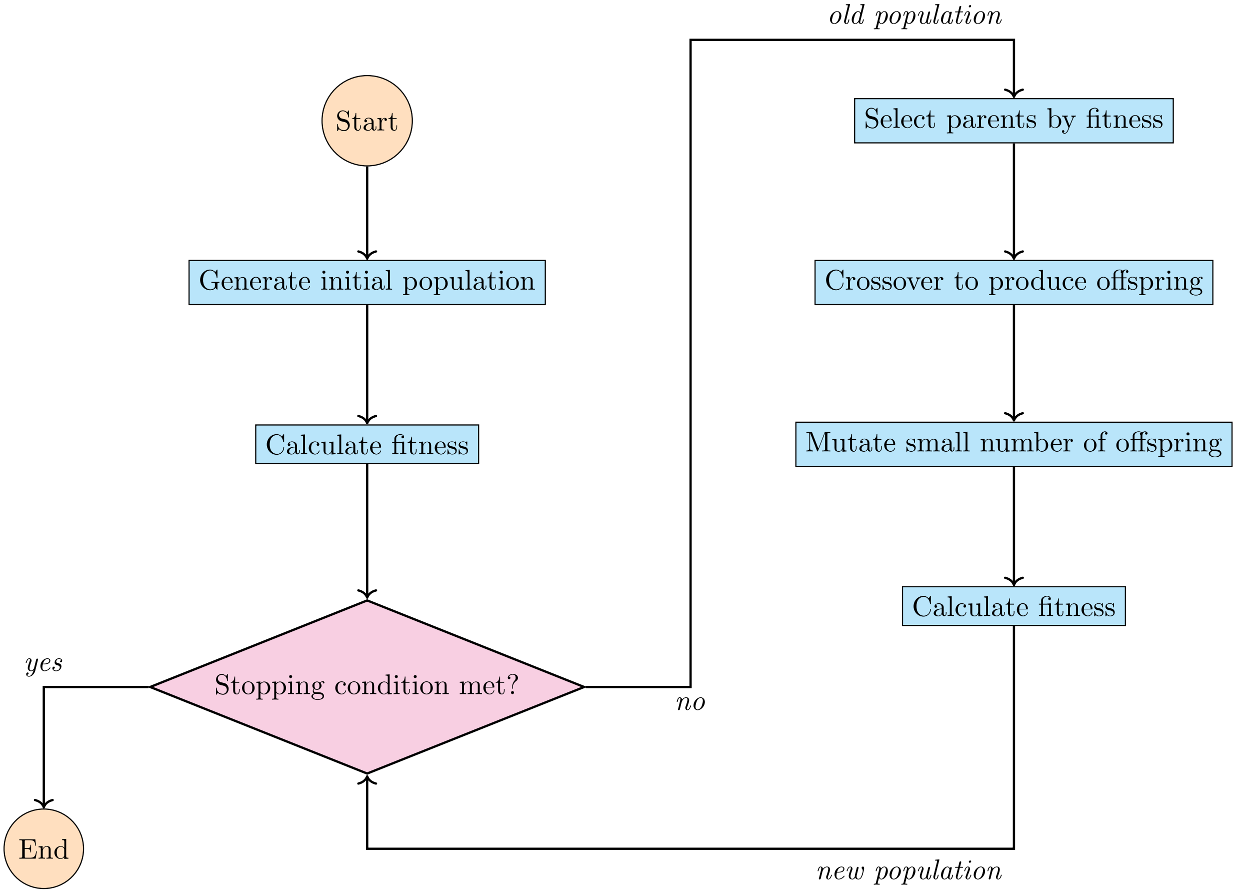
old population
Start
Select parents by fitness
Generate initial population
Crossover to produce offspring
Calculate fitness
Mutate small number of offspring
Calculate fitness
yes
Stopping condition met?
no
End
new population

Figure 1: A general schematic for an evolutionary algorithm.

The use of EAs to generate artificial data, however, is not a new concept; these methods have been used to develop techniques for the automated testing of software with a goal of achieve maximal coverage or to support structural tests [5, 7, 10]. Such methods also have a long history in the parameter optimisation of algorithms, and recently in the automated design of convolutional neural network architecture [11, 12].

2 The evolutionary algorithm

2.1 Structure

On the surface, the EDO method is built as an evolutionary algorithm which follows a traditional (generic) schema with some additional features that keep the objective of artificial data generation in mind. With that, there are a number of parameters passed to the algorithm, including several that are not typical:

- A real-valued fitness function, $f : X \to \mathbb{R}$, which acts on a dataset to return a single fitness score.

- A population size, $N \in \mathbb{N}$.

- Limits on the number of rows a dataset can have, $R \in \left\{(r_{\min}, r_{\max}) \in \mathbb{N}^2 \mid r_{\min} \leq r_{\max}\right\}$.

- Limits on the number of columns a dataset can have

$$C := \left(C_1, \ldots, C_{|\mathcal{P}|}\right) \text{ where } C_j \in \left\{(c_{\min}, c_{\max}) \in (\mathbb{N} \cup \{\infty\})^2 \mid c_{\min} \leq c_{\max}\right\}$$

  for each $j = 1, \ldots, |\mathcal{P}|$. That is, C is defined as a tuple detailing the minimum and maximum number of columns a dataset can have for each distribution passed in $\mathcal{P}$.

- A set of probability distribution families, $\mathcal{P}$. Each family in this set has some parameter limits which form a part of the overall search space. For instance, the normal distribution family, denoted by $N(\mu, \sigma^2)$, would have limits on values for the mean, μ, and the standard deviation, σ.

- A probability vector to sample distributions from $\mathcal{P}$, $w = \left(w_1, \ldots, w_{|\mathcal{P}|}\right)$ such that $\sum_{i=1}^{|\mathcal{P}|} w_i = 1$ and $w_i \in [0, 1], \ \forall \ i = 1, \ldots, |\mathcal{P}|$.

- A maximum number of iterations, $M \in \mathbb{N}$.

- Two selection parameters: one to indicate the proportion of the fittest individuals to carry forward, $b \in [0, 1]$, and to allow for a small proportion of "lucky" individuals in the next generation, $l \in [0, 1]$.

- A mutation probability, $p_m \in [0, 1]$, used to indicate the intensity and likelihood of the mutation a new individual might undergo.

- A shrink factor, $s \in [0, 1]$. The relative size of a component of the search space to be retained after adjustment.

The concepts discussed in this section form the mechanisms of the evolutionary dataset optimisation algorithm. To use the algorithm practically, these components have been implemented in Python as a library built on the scientific Python stack [6, 9]. The library is fully tested and documented (at `https://edo.readthedocs.io`) and is freely available online under the MIT license [3]. The EDO implementation was developed to be consistent with the best practices of open source software development.

The statement of the EDO algorithm is deliberately vague. This is, in part, because of the customisable nature of its build but also to lay out its general structure from a high level perspective. Lower level

Algorithm 1: The evolutionary dataset optimisation algorithm

```
Data: $f, N, R, C, \mathcal{P}, w, M, b, l, p_m, s$
Result: A full history of the populations and their fitnesses.
begin
    create initial population of individuals
    find fitness of each individual
    record population and its fitness
    while current iteration less than the maximum and stopping condition not met do
        select parents based on fitness and selection proportions
        use parents to create new population through crossover and mutation
        find fitness of each individual
        update population and fitness histories
        if adjusting the mutation probability then
            update mutation probability
        end
        if using a shrink factor then
            shrink the mutation space based on parents
        end
    end
end
```

Algorithm 2: Creating a new population

```
Data: parents, $N, R, C, \mathcal{P}, w, p_m$
Result: A new population of size $N$
begin
    add parents to the new population
    while the size of the new population is less than $N$ do
        sample two parents at random
        create an offspring by crossing over the two parents
        mutate the offspring according to the mutation probability
        add the mutated offspring to the population
    end
end
```

discussion is provided below where additional algorithms for the individual creation, evolutionary operator and shrinkage processes are given along with diagrams (where appropriate).

Note that there are no defined processes for how to stop the algorithm or adjust the mutation probability, p_m. These form two key areas of potential customisation since such conditions can be specific to the problem domain. With that, as is detailed in the Python implementation, a user may define their stopping condition or p_m-adjustment to be any reasonable function. Some examples include:

- These should be real-world examples used in other EA with references.
- Stopping when the difference in the variance of the current and last population fitnesses is below some tolerance.
- Halving the mutation probability every 100 iterations.

2.2 Individuals

Evolutionary algorithms operate on succeeding populations of individuals often coined as “generations”. Typically, an individual would be encoded as a bit string of a fixed length and treated as a chromosome-like object to be manipulated. In EDO, as the objective is to generate datasets, there is no encoding process. Instead, the datasets are manipulated directly so that the biological operators can behave and be interpreted in a more meaningful way.

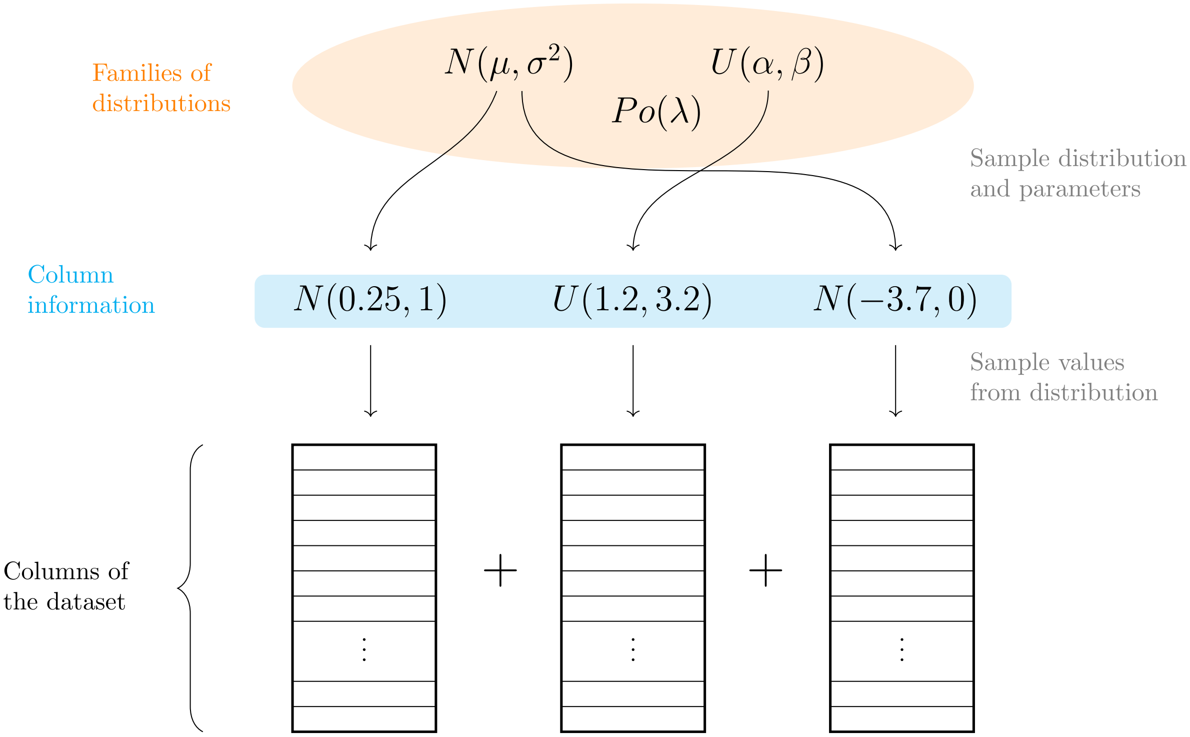

Figure 2: An example of how an individual is first created.

In a sense, a dataset is treated similarly to a classical bit string as the primary components (loci) of the dataset are considered to be its columns. As is seen in Figure 2, an individual's creation is defined by the random generation of its columns. A set of instructions on how to sample new values for that column are recorded in the form of a probability distribution. These distributions are sampled from a pool of distribution families which is passed to the evolutionary algorithm along with the other parameters.

Obviously, users and interpreters of EDO should not be so quick to assume that these pairs of distributions and columns are typical of their partner. That is, that the columns are a reliable representative of the distribution associated with them, or vice versa. A caveat to this statement: this is particularly true of "shorter" datasets with a small number of rows, whereas confidence in the pair could be given more liberally for "longer" datasets. In the case of the latter, the column metadata can become more useful for casually analysing the data which is generated. In any case, however, more direct and sophisticated methods should be employed to understand the structure and characteristics of the data before formal conclusions are made.

Algorithm 3: Creating an individual

```
Data: $R, C, \mathcal{P}, w$
Result: An individual defined by a dataset and some metadata
begin
    sample a number of rows and columns
    create an empty dataset
    for each column in the dataset do
        sample a distribution from $\mathcal{P}$
        create an instance of the distribution
        fill in the column by sampling from this instance
        record the instance in the metadata
    end
end
```

2.3 Selection

The selection operator describes the process by which individuals are chosen from the current population to generate the next. Almost always, the likelihood of an individual being selected is determined by their fitness. This is because the purpose of selection is to preserve favourable qualities and encourage some homogeneity within future generations.

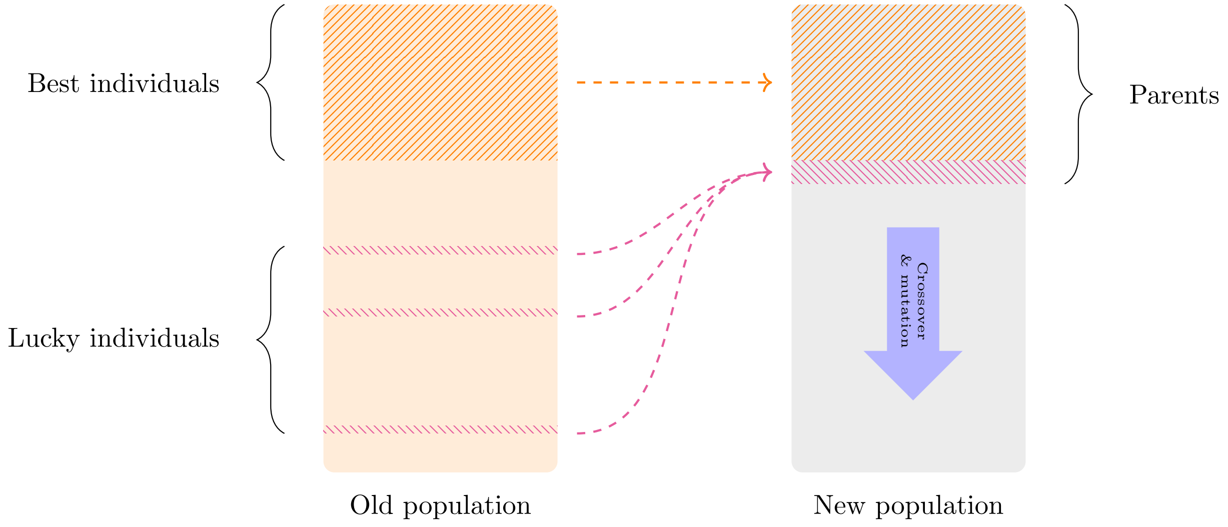

Figure 3: The selection process with the inclusion of some lucky individuals.

In EDO, a modified truncation selection method is used as can be seen in Figure 3. Standard truncation selection takes a fixed number, $n_b = \lceil bN \rceil$, of the fittest individuals in a population and makes them the "parents" of the next. The modification is an optional stage after the best individuals have been chosen. By passing some small l to the evolutionary algorithm, a number of random individuals can be selected to be carried forward. This number is given by $n_l = \lceil lN \rceil$. It should be noted that even with this modification, no individual may be selected more than once in a single iteration but could potentially be present throughout the entire run of the algorithm.

Algorithm 4: The selection process

```
Data: population, population fitness, $b$, $l$
Result: A set of parent individuals
begin
    calculate $n_b$ and $n_l$
    sort the population by the fitness of its individuals
    take the first $n_b$ individuals and make them parents
    if there are any individuals left then
        take the next $n_l$ individuals and make them parents
    end
end
```

It has been found that, despite its efficiency as a selection operator, truncation selection can lead an evolutionary algorithm to converge prematurely to local optima [4]. Hence, the ability to include some randomly selected individuals is included to encourage diversity throughout the run of the algorithm. This feature can be particularly useful in more complex optimisation scenarios - or for larger populations where a greater loss of diversity is seen in the selection process simply due to the nature of truncation selection [8]. As such, it should be used sparingly so as not to dominate the selection process with unwanted randomness.

2.4 Crossover

Crossover is the operation of combining two individuals in order to create at least one offspring. Often in classical evolutionary algorithms, the term “crossover” is taken literally: two bit strings are crossed at a point to produce two new bit strings. Another popular method is uniform crossover where the loci of a new individual are sampled uniformly from either parent individual. This method is adapted to support dataset manipulation here by sampling first a set of dimensions from the parents, and then inheriting entire columns.

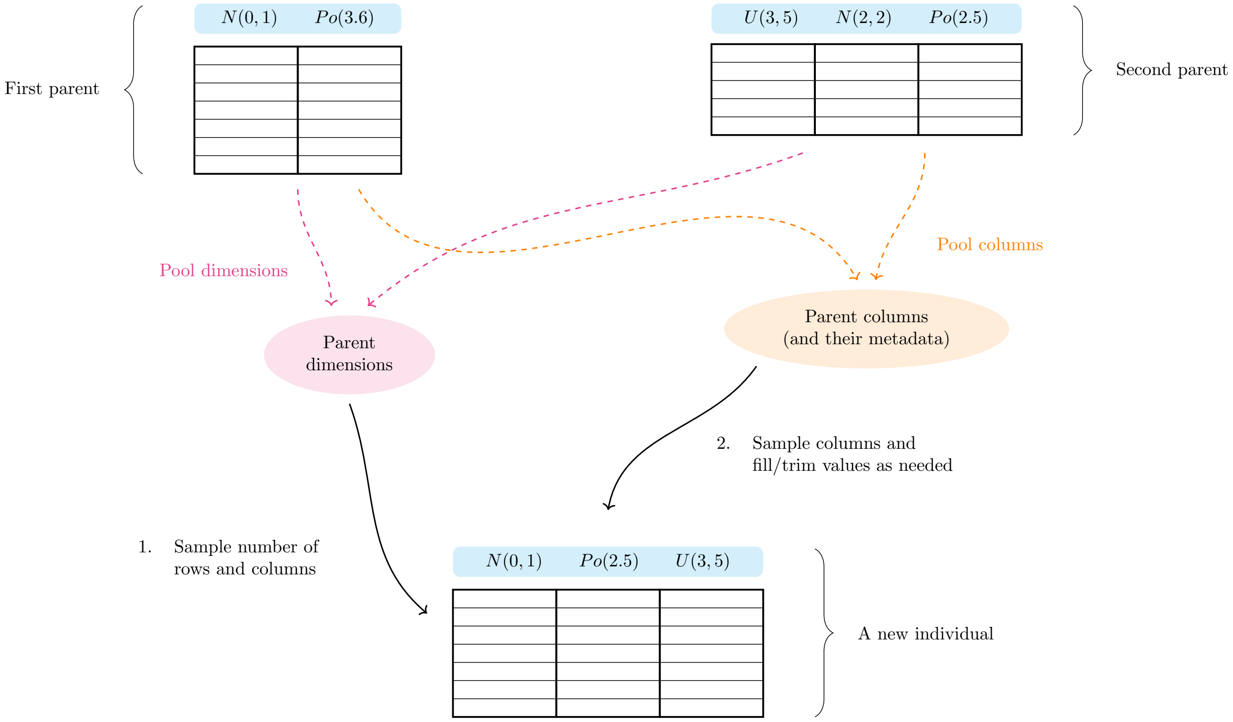

Figure 4: The crossover process.

```
Algorithm 5: The crossover process

Data: Two parents
Result: An offspring made from the parents ready for mutation
begin
    collate the columns and metadata from each parent in a pool
    sample each dimension from between the parents uniformly
    form an empty dataset with these dimensions
    for each column in the dataset do
        sample a column (and its corresponding metadata) from the pool
        if this column is longer than required then
            randomly select entries and delete them as needed
        end
        if this column is shorter than required then
            sample new values from the metadata and append them to the column as needed
        end
        add this column to the dataset and record its metadata
    end
end
```

2.5 Mutation

Mutation is used in evolutionary algorithms to encourage a broader exploration of the search space. Traditional applications with chromosome representations would mutate by run along the loci of an individual and "switching" the binary value with some constant probability. Under this framework, as has been discussed, an individual's columns are similar to traditional loci. However, in the mutation phase multiple characteristics of an individual are susceptible to being mutated beyond the columns themselves such as the dimensions of the individual and their column metadata.

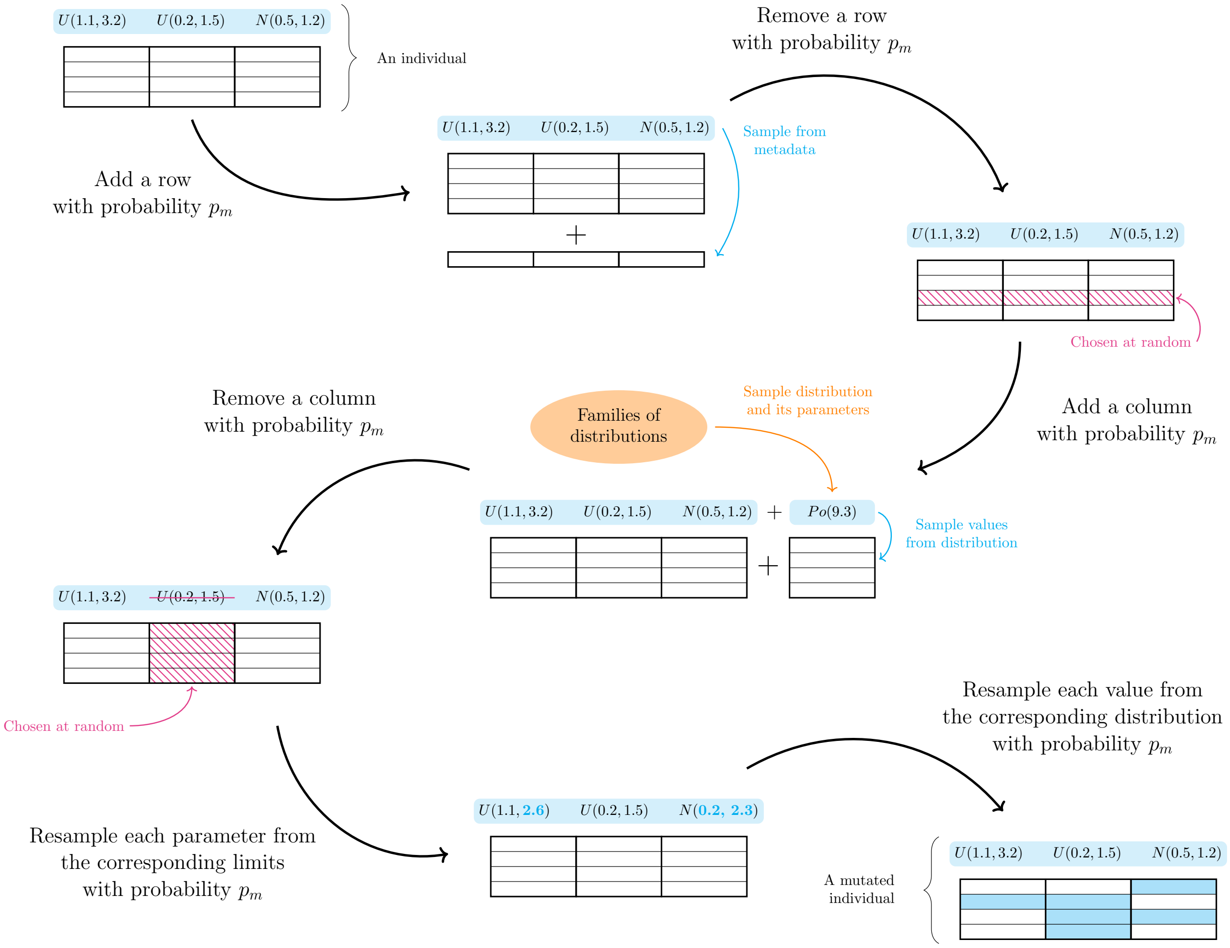
$U(1.1, 3.2)$ $U(0.2, 1.5)$ $N(0.5, 1.2)$
An individual
Add a row with probability p_m
Remove a row with probability p_m
Sample from metadata
+
Chosen at random
Add a column with probability p_m
Families of distributions
Sample distribution and its parameters
$Po(9.3)$
Sample values from distribution
Remove a column with probability p_m
~~$U(0.2, 1.5)$~~
Chosen at random
Resample each parameter from the corresponding limits with probability p_m
$U(1.1, 2.6)$ $U(0.2, 1.5)$ $N(0.2, 2.3)$
Resample each value from the corresponding distribution with probability p_m
A mutated individual

Figure 5: The mutation process.

The mutation process, seen in Figure 5 and defined in Algorithm 6, is deliberately fine so that all aspects of an individual can be changed in a meaningful yet fluid way. Each of the potential mutations occur with the same probability, p_m, but the way in which columns are maintained assure that (assuming appropriate choices for f and $\mathcal{P}$) many mutations in the metadata and the dataset itself will only result in some incremental change in the individual's genetics and fitness overall - at least relatively so to, say, a completely new individual.

2.6 Shrinking

The potential benefits of adapting the search space of an EA has been well-discussed in the domain of complex optimisation problems. During the development of EDO, two methods were considered to do this. Each of these methods relied on a law relating a successive generation's search space with its predecessor. It should be noted that in both cases the methods were adapted to conform with the choice to represent individuals as they are rather than as bit strings.

The first was developed to be a two-part process based on a linear law with two parameters: a shrink factor, $s \in (0, 1)$, and the maximum number of iterations $M \in \mathbb{N}$ [2]. The adapted method would be as follows. For all iterations, $t \geq sM$, no shrinking would take place. However, for all previous iterations, the suggested method would take the parents found during selection and act on each component of the search space. That is, for each iteration, $t < sM$, every component would have its lower and upper limits, denoted by l_t and u_t respectively, adjusted so that they are centred about the mean parent value, μ, and be such that:

$$u_{t+1} - l_{t+1} = (u_t - l_t)\left(1 - \frac{t}{sM}\right)$$

More specifically, the adjusted limits would be calculated as follows:

$$\begin{aligned} l_{t+1} &= \max\left\{l_t,\ \mu - \frac{1}{2}(u_t - l_t)\left(1 - \frac{t}{sM}\right)\right\} \\ u_{t+1} &= \min\left\{u_t,\ \mu + \frac{1}{2}(u_t - l_t)\left(1 - \frac{t}{sM}\right)\right\} \end{aligned}$$

The second method was for use in a genetic algorithm which mapped weighted bit strings to some predefined interval with lower and upper bounds [1]. The proposed method would rely on a power law with a single parameter: some shrink factor, $s \in [0, 1]$. Again, at each iteration, the parents would be taken and every component's limits would be adjusted so that they were centred about the mean parent value, μ, such that:

$$u_{t+1} - l_{t+1} = (u_t - l_t)s^t$$

The process by which the values of l_{t+1} and u_{t+1} would be found are equivalent to the above but with a

Algorithm 6: The mutation process

```
Data: An individual, $p_m$, $R$, $C$, $\mathcal{P}$, $w$
Result: A mutated individual
begin
    sample a random number $r \in [0, 1]$
    if $r < p_m$ and adding a row would not violate $R$ then
        sample a value from each distribution in the metadata
        append these values as a row to the end of the dataset
    end
    sample a new $r \in [0, 1]$
    if $r < p_m$ and removing a row would not violate $R$ then
        remove a row at random from the dataset
    end
    sample a new $r \in [0, 1]$
    if $r < p_m$ and adding a new column would not violate $C$ then
        create a new column using $\mathcal{P}$ and $w$
        append this column to the end of the dataset
    end
    sample a new $r \in [0, 1]$
    if $r < p_m$ and removing a column would not violate $C$ then
        remove a column (and its associated metadata) at random from the dataset
    end
    for each distribution in the metadata do
        for each parameter of the distribution do
            sample a random number $r \in [0, 1]$
            if $r < p_m$ then
                sample a new value from within the distribution parameter limits
                update the parameter value with this new value
            end
        end
    end
    for each entry in the dataset do
        sample a random number $r \in [0, 1]$
        if $r < p_m$ then
            sample a new value from the associated column distribution
            update the entry with this new value
        end
    end
end
```

different shift term:

$$l_{t+1} = \max\left\{l_t,\ \mu - \frac{1}{2}(u_t - l_t)s^t\right\} \tag{1}$$

$$u_{t+1} = \max\left\{u_t,\ \mu + \frac{1}{2}(u_t - l_t)s^t\right\} \tag{2}$$

From these brief definitions alone, these methods appear to be largely indistinguishable. However, following a wider discussion around how EDO would work for the general user, it was decided that the first method be rejected in favour of the second. It was found that the second method having fewer parameters was a particularly redeeming feature; this removed any hidden or otherwise unwanted interactions between a parameter dedicated to the shrink process, s, and the maximum number of iterations, M, which is often used as a fallback stopping criterion in complex problem domains.

Algorithm 7: Shrinking the mutation space

```
Data: parents, current iteration, $\mathcal{P}, M, s$
Result: A new mutation space focussed around the parents
begin
    for each distribution in $\mathcal{P}$ do
        for each parameter of the distribution do
            get the current values for parameter over all parent columns
            find the mean of the current values
            find the new lower (1) and upper (2) bounds around the mean
            set the parameter limits
        end
    end
end
```

3 Numerical examples

- The x^2 example from the docs is a nice easy one to illustrate things
- Something stochastic
- Make use of the moving parts (linear focus of the mutation space, dwindling mutation probability, stopping conditions)
- If not the k-modes initialisation example, then another clustering one. Maybe k-means versus DBSCAN to show DBSCAN works better on non-convex clusters.

References

[1] Adil Amirjanov. “Modeling the Dynamics of a Changing Range Genetic Algorithm”. In: *Procedia Computer Science* 102 (2016), pp. 570 –577. DOI: https://doi.org/10.1016/j.procs.2016.09.444.

[2] Adil Amirjanov and Fahreddin Sadikoglu. “Linear adjustment of a search space in genetic algorithm”. In: *Procedia Computer Science* 120 (2017), pp. 953 –960. DOI: https://doi.org/10.1016/j.procs.2017.11.331.

[3] The EDO library developers. *EDO: v0.1.* 2018. DOI: 10.5281/zenodo.2557597. URL: http://dx.doi.org/10.5281/zenodo.2557597.

[4] Khalid Jebari. “Selection Methods for Genetic Algorithms”. In: *International Journal of Emerging Sciences* 3 (Dec. 2013), pp. 333–344.

[5] Chahine Koleejan, Bing Xue, and Mengjie Zhang. “Code coverage optimisation in genetic algorithms and particle swarm optimisation for automatic software test data generation”. In: *2015 IEEE Congress on Evolutionary Computation (CEC)* (2015), pp. 1204–1211.

[6] Wes McKinney. *Data Structures for Statistical Computing in Python.* Proceedings of the 9th Python in Science Conference. [Online; accessed 2019-03-01]. 2010–. URL: https://pandas.pydata.org/.

[7] Christoph C. Michael, Gary McGraw, and Michael Schatz. “Generating Software Test Data by Evolution”. In: *IEEE Trans. Software Eng.* 27 (2001), pp. 1085–1110.

[8] Tatsuya Motoki. “Calculating the Expected Loss of Diversity of Selection Schemes”. In: *Evolutionary Computation* 10.4 (2002), pp. 397–422. DOI: 10.1162/106365602760972776.

[9] Travis Oliphant. *NumPy: A guide to NumPy.* USA: Trelgol Publishing. [Online; accessed 2019-03-01]. 2006–. URL: http://www.numpy.org/.

[10] Hossein Sharifipour, Mojtaba Shakeri, and Hassan Haghighi. “Structural test data generation using a memetic ant colony optimization based on evolution strategies”. In: *Swarm and Evolutionary Computation* 40 (2018), pp. 76 –91. DOI: https://doi.org/10.1016/j.swevo.2017.12.009.

[11] Masanori Suganuma, Shinichi Shirakawa, and Tomoharu Nagao. “A genetic programming approach to designing convolutional neural network architectures”. In: *GECCO.* 2017.

[12] Yanan Sun et al. “Automatically Designing CNN Architectures Using Genetic Algorithm for Image Classification”. In: *CoRR* abs/1808.03818 (2018).

[13] A. Torralba and A. A. Efros. *Unbiased Look at Dataset Bias*. Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition. 2011. DOI: `10.1109/CVPR.2011.5995347`. URL: `http://dx.doi.org/10.1109/CVPR.2011.5995347`.