Evolutionary Dataset Optimisation: learning algorithm quality through genetic evolution

Henry Wilde, Vincent Knight, Jonathan Gillard

Abstract

When faced with a problem involving data, it is almost certainly the case that the data is fixed and in order to do something with that data, a researcher must select an algorithm that is appropriate for the problem domain whilst performing well on their data. The value prescribed to an algorithm is often found through a process of surveying the current literature to create a shortlist, then running various trials with the shortlisted algorithms. The winning algorithm is then chosen based on some common objective value. The issue with this process is that it does not necessarily allow (or require) the researcher to consider why certain algorithms perform better on particular datasets and not others, and which characteristics make data "good" for their chosen algorithm.

This paper introduces a novel method for generating artificial data through genetic evolution, the purpose of which is to create populations of datasets for which a particular algorithm performs well. This is done by passing an algorithm's objective function to an evolutionary algorithm. Therein, each individual is a particular dataset defined by its dimensions, entries, and the approximate statistical shape of each of its attributes. In this way, detailed information about each individual is retained throughout the algorithm. Hence, they may be manipulated in a meaningful way during the run, and studied once the algorithm has terminated.

Following this, a number of examples are given to show the performance of the method. These examples are created using a Python implementation of the process which is built to be highly customisable, interpretable and reproducible.

1 Introduction

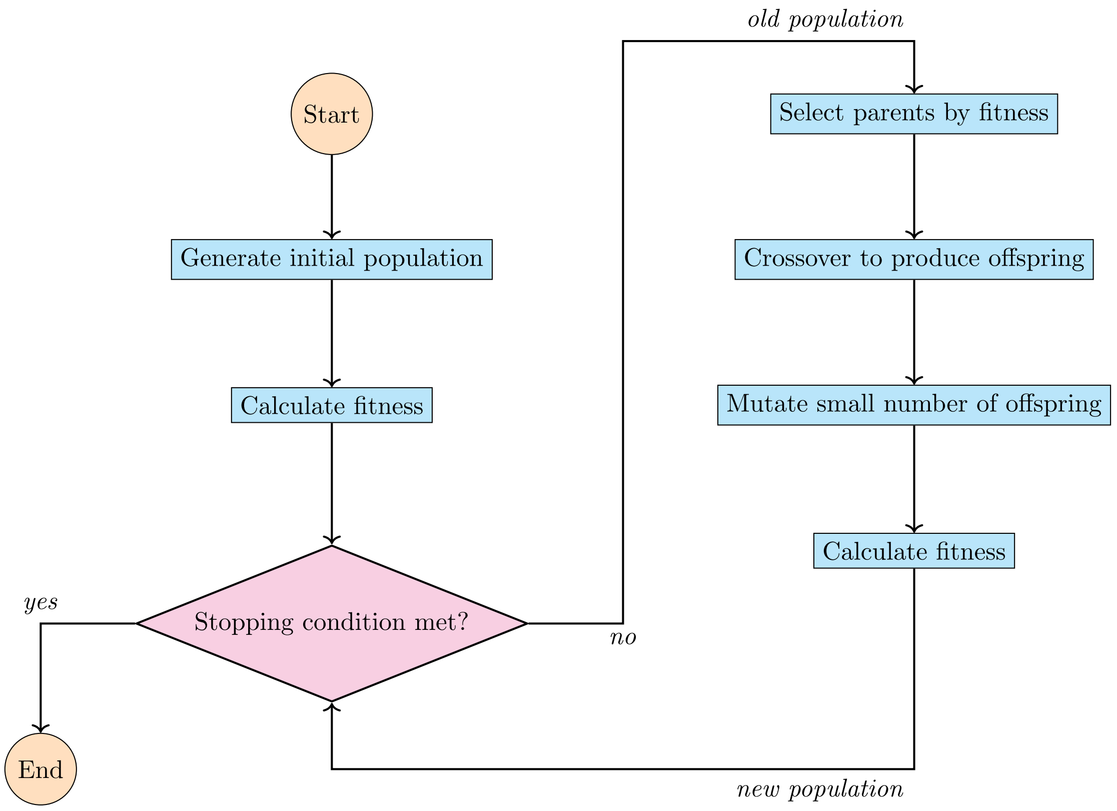
old population
Start
Select parents by fitness
Generate initial population
Crossover to produce offspring
Calculate fitness
Mutate small number of offspring
Calculate fitness
yes
Stopping condition met?
no
End
new population

Figure 1: A general schematic for an evolutionary algorithm.

- What is the motivation?
- What is the problem?
- What is the solution?

1.1 Literature review

- How is artificial data made?
- Why hasn't this been done before?
- Genetic algorithms used to train algorithms for data
- Diagram showing that this is the "reverse" problem

2 The evolutionary algorithm

2.1 Structure

- Algorithm statement with components
- Discussion on the choice to include custom stopping/dwindling conditions, compacting the search space, etc.
- Operators and detailed mechanisms to come later.

Algorithm 1: The evolutionary dataset optimisation algorithm

```
Data: Fitness function, $f : X \to \mathbb{R}$; population size, $N$; row limits, $R = [r_l, r_u]$; column limits, $C = [c_l, c_u]$;
      Column distributions, $P$; relative weights for $P$, $w \in [0,1]^{|P|}$ s.t. $\sum w = 1$; maximum iterations, $M$;
      Best proportion, $\delta$; lucky proportion, $\epsilon$; mutation probability, $\mu$; compaction ratio, $s$.
Result: A full history of the populations and their fitnesses.

initialisation
$population \longleftarrow CreateInitialPopulation(N, R, C, P, w)$
$populationFitness \longleftarrow GetPopulationFitness(population, f)$
$populationHistory \longleftarrow \{(population, populationFitness)\}$
$i \longleftarrow 0$

begin iterative step
while $iteration < M$ and not STOP do
    select parent individuals
    $parents \longleftarrow Selection(population, populationFitness, \delta, \epsilon)$
    create new population
    $population \longleftarrow CreateNewPopulation(parents, N, R, C, P, w, \mu)$
    update fitness and history
    $populationFitness \longleftarrow GetPopulationFitness(population, f)$
    $populationHistory \longleftarrow populationHistory \cup \{(population, populationFitness)\}$
    adjust parameters
    $i \longleftarrow i + 1$
    $P \longleftarrow ReduceMutationSpace(parents, P, i, M, s)$
    $\mu$, STOP $\longleftarrow UpdateParameters$
```

Additional algorithms for the individual creation and operator processes are given in Section 2.2. However, there is no defined process for how to stop the algorithm or adjust the mutation probability, μ. This is deliberate as such conditions are specific to the problem domain. As such, in the Python implementation, a user may define their stopping condition or μ-adjustment to be any function. Some examples include:

- These should be real-world examples used in other EA with references.

Algorithm 2: *CreateInitialPopulation*

```
Data: $N, R, C, P, w$
Result: A random population of datasets of size $N$, $population$

$population \longleftarrow \emptyset$
for $i \leftarrow 1$ to $N$ do
    $population \longleftarrow population \cup \{CreateIndividual(R, C, P, w)\}$
```

Algorithm 3: *GetPopulationFitness*

```
Data: $population, f$
Result: Fitness of each individual in the population, $populationFitness$

$populationFitness \longleftarrow \emptyset$
for $individual \in population$ do
    $populationFitness \longleftarrow populationFitness \cup \{f(individual)\}$
```

Algorithm 4: *CreateNewPopulation*

```
Data: $parents, N, R, C, P, w, \mu$
Result: A new population of size $N$, $newPopulation$

$newPopulation \longleftarrow parents$
while $|newPopulation| < N$ do
    Sample two parents, $parentOne, parentTwo$, from $parents$
    $offspring \longleftarrow Crossover(parentOne, parentTwo, C, P, )$
    $mutant \longleftarrow Mutation(offspring, \mu, R, C, P, w)$
    $newPopulation \longleftarrow newPopulation \cup \{mutant\}$
```

Algorithm 5: *ReduceMutationSpace*

```
Data:
Result:

initialisation
$S \longleftarrow 1 - \frac{i}{sM}$
$currentParameters \longleftarrow GetCurrentParameters(parents, P)$

compute new bounds on distribution parameters
for $parameters \in currentParameters$ do
    for $parameter, parameterValues \in parameters$ do
        $m \longleftarrow Mean(parameterValues)$
        $\Delta \longleftarrow \frac{S}{2} \times (\max parameterValues - \min parameterValues)$
        $L \longleftarrow \max\{parameterLimits[0], \min\{m - \Delta, m + \Delta\}\}$
        $U \longleftarrow \min\{parameterLimits[1], \max\{m - \Delta, m + \Delta\}\}$
        $parameterLimits \longleftarrow [L, U]$
```

Algorithm 6: *GetCurrentParameters*

```
$currentParameters \longleftarrow (p : (\emptyset, \ldots, \emptyset) \mid p \in P)$
for $parent \in parents$ do
    for each column in parent do
        $p \longleftarrow$ distribution class of column
        if no entry for $p$ in $currentParameters$ then
            $currentParameters[p] \longleftarrow \emptyset$
        for $(parameter, parameterValue) \in p$ do
            $currentParameters[p] \longleftarrow currentParameters[p] \cup \{(parameter, parameterValue)\}$
```

- 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 Internal mechanisms

2.2.1 Individuals

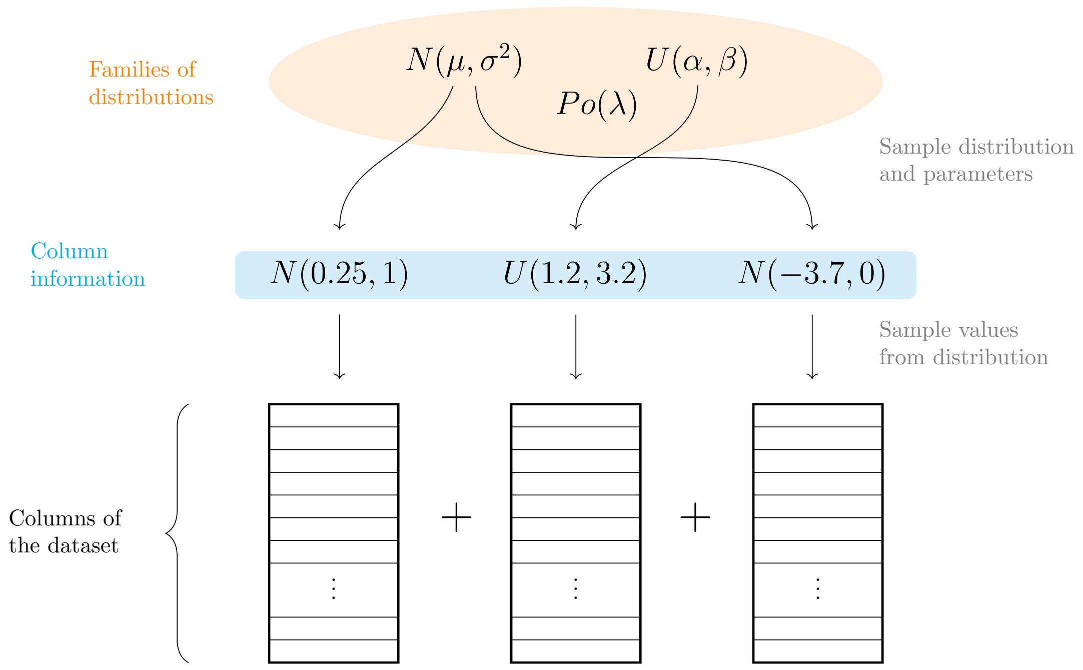

Figure 2: An example of how an individual is first created.

2.2.2 Selection

2.2.3 Crossover

2.2.4 Mutation

2.2.5 Shrinking

2.3 Implementation

- Documentation: `https://edo.readthedocs.io`
- Repo: `https://github.com/daffidwilde/edo`

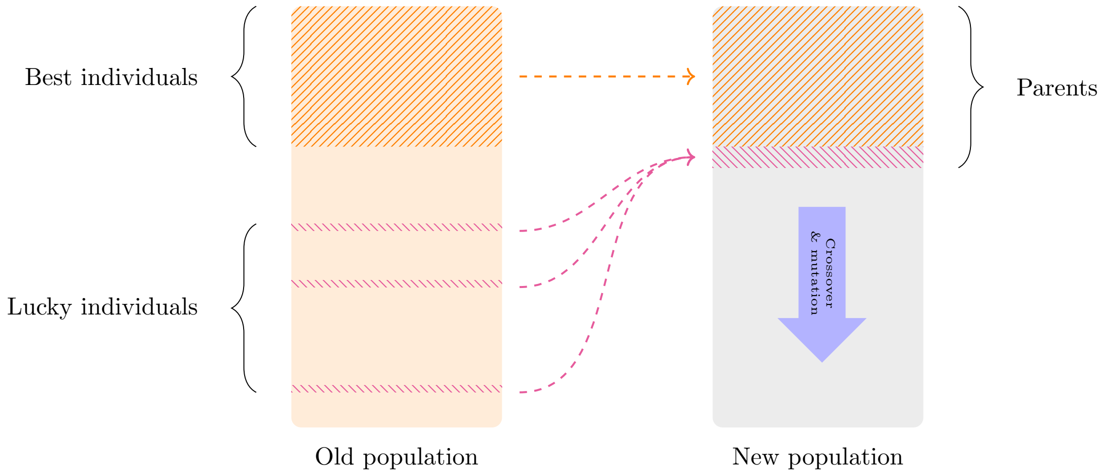

Figure 3: An example of the selection process with the inclusion of some lucky individuals.

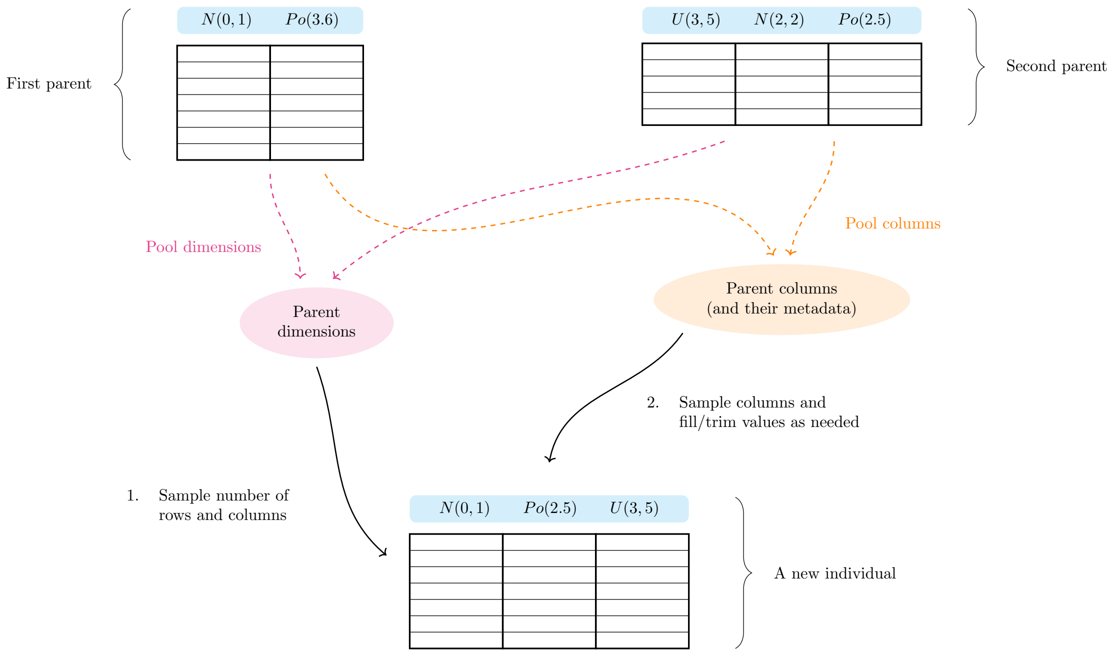
First parent
$N(0,1)$ $Po(3.6)$
$U(3,5)$ $N(2,2)$ $Po(2.5)$
Second parent
Pool dimensions
Pool columns
Parent dimensions
Parent columns (and their metadata)
2. Sample columns and fill/trim values as needed
1. Sample number of rows and columns
$N(0,1)$ $Po(2.5)$ $U(3,5)$
A new individual

Figure 4: An example of the crossover process.

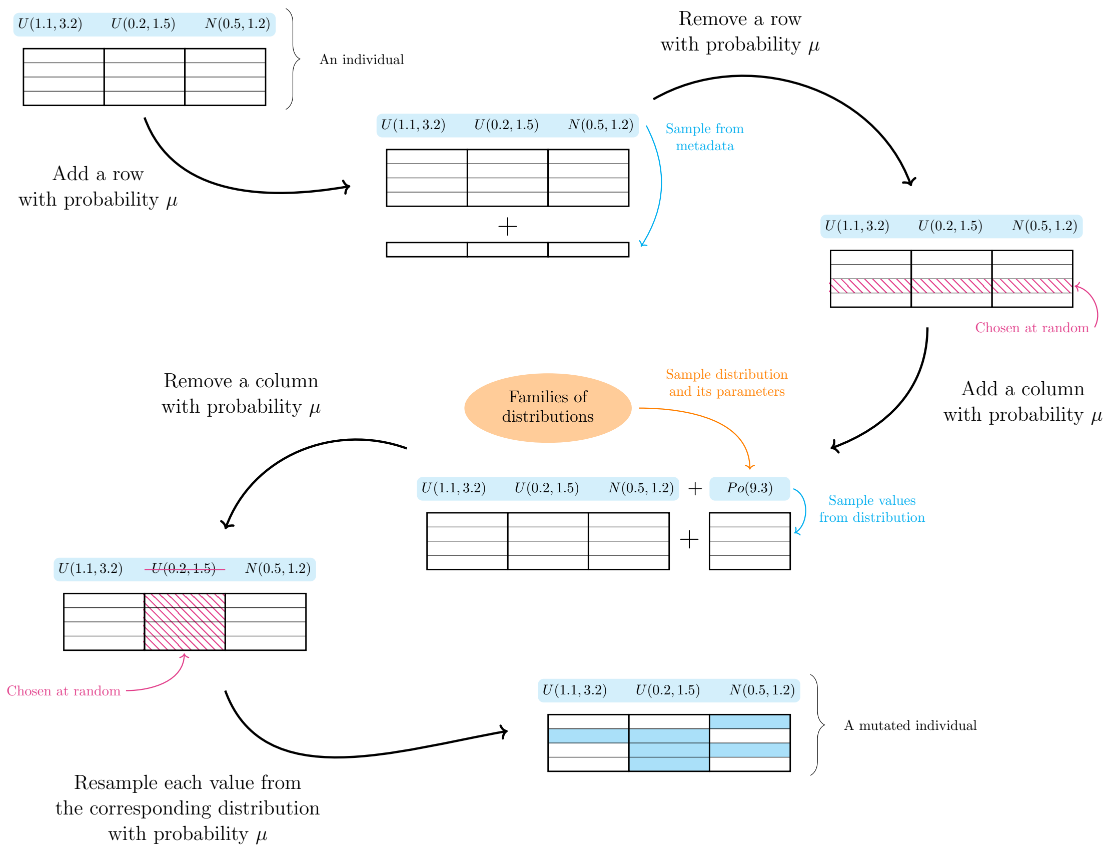

Figure 5: An example of the mutation process.

Figure 6: An diagram of the shrinking process.

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.