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

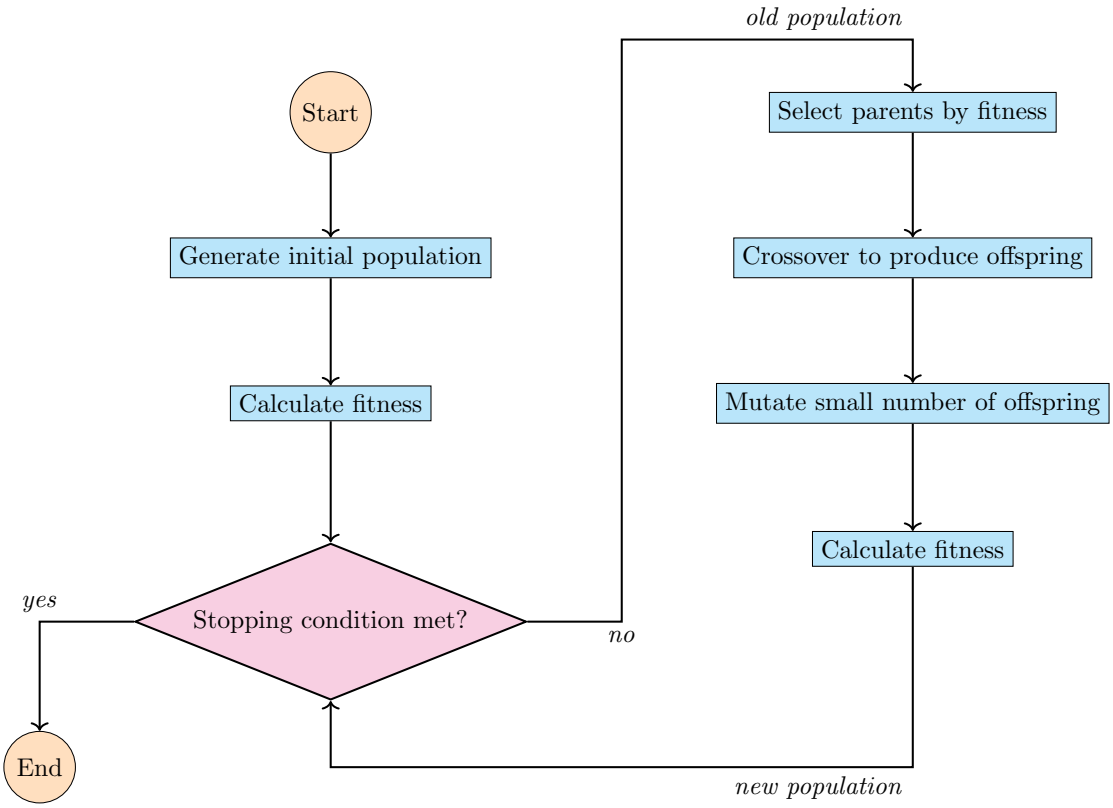

Figure 1: A general schematic for an evolutionary algorithm.

- 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: $f, N, R, C, \mathcal{P}, w, M, b, l, \mu, s$
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 shrinking factor* **then**
      shrink the mutation space based on parents
    end
  end
end

Algorithm 2: Creating an initial population

Data: $N, R, C, \mathcal{P}, w$
Result: A population of datasets of size N
begin
  start with an empty population
  for $i \leftarrow 1$ **to** N **do**
    create an individual using the row/column limits and distribution pool
    add them to the population
  end
end

Algorithm 3: Creating a new population

```
Data: parents, N, R, C, 𝒫, w, μ
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
```

Algorithm 4: Shrinking the mutation space

```
Data: parents, current iteration, 𝒫, M, s
Result: A new mutation space focussed around the parents
begin
    calculate the shrink coefficient
    for distribution in 𝒫 do
        for each parameter of the distribution do
            get current values for parameter in parent columns
            find the mean value
            calculate the shift term using the values and shrink coefficient
            find the new lower and upper bounds around the mean
            set the parameter limits
        end
    end
end
```

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.
- 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

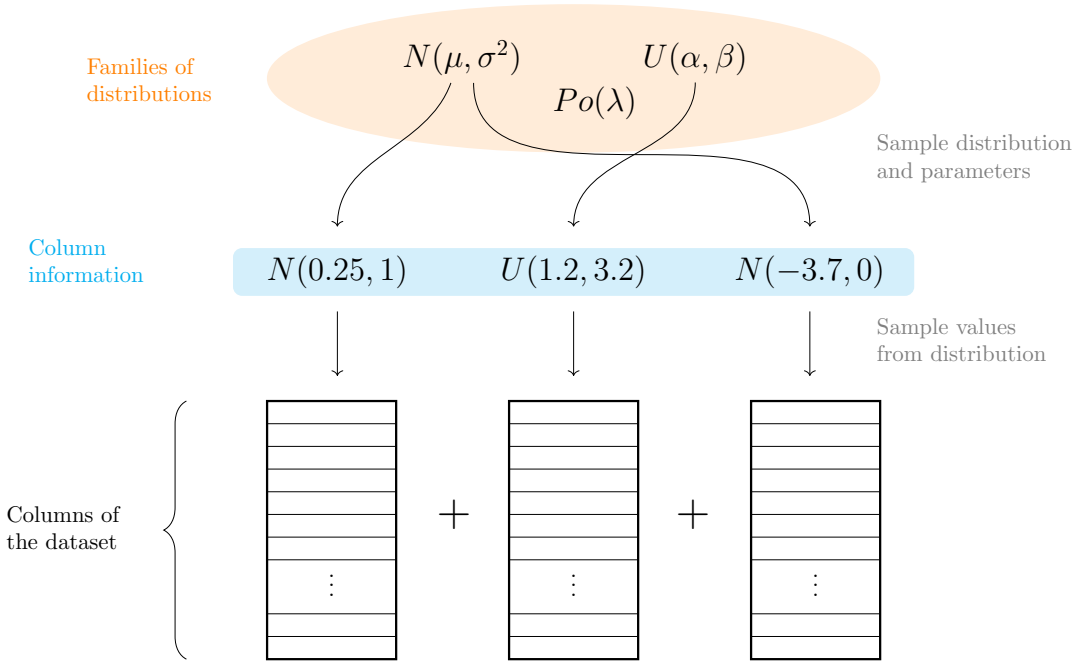

Figure 2: An example of how an individual is first created.

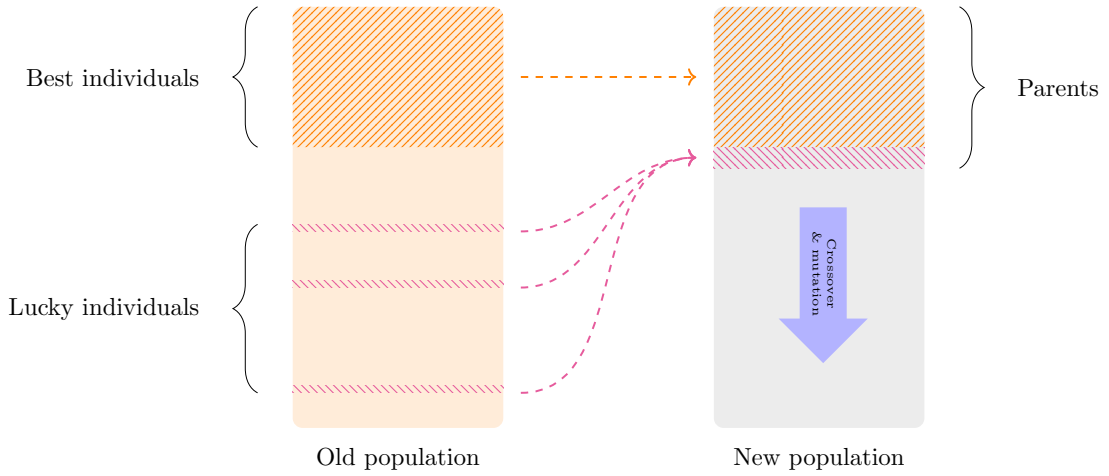

Figure 3: An example of the selection process with the inclusion of some lucky individuals.

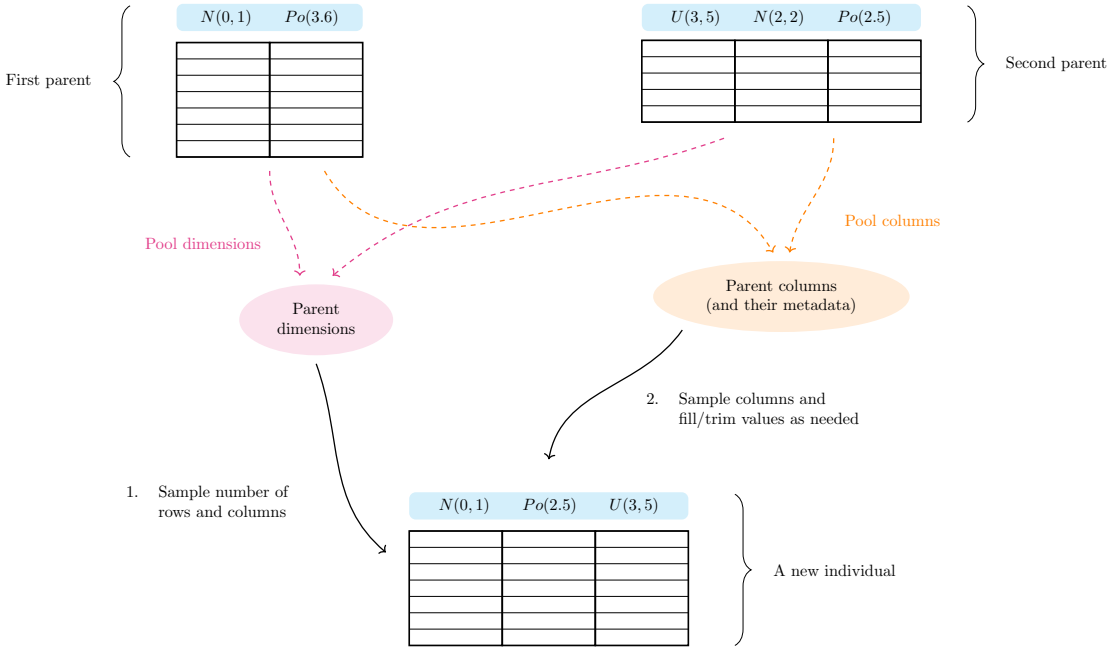

Figure 4: An example of the crossover process.

2.2.2 Selection

2.2.3 Crossover

2.2.4 Mutation

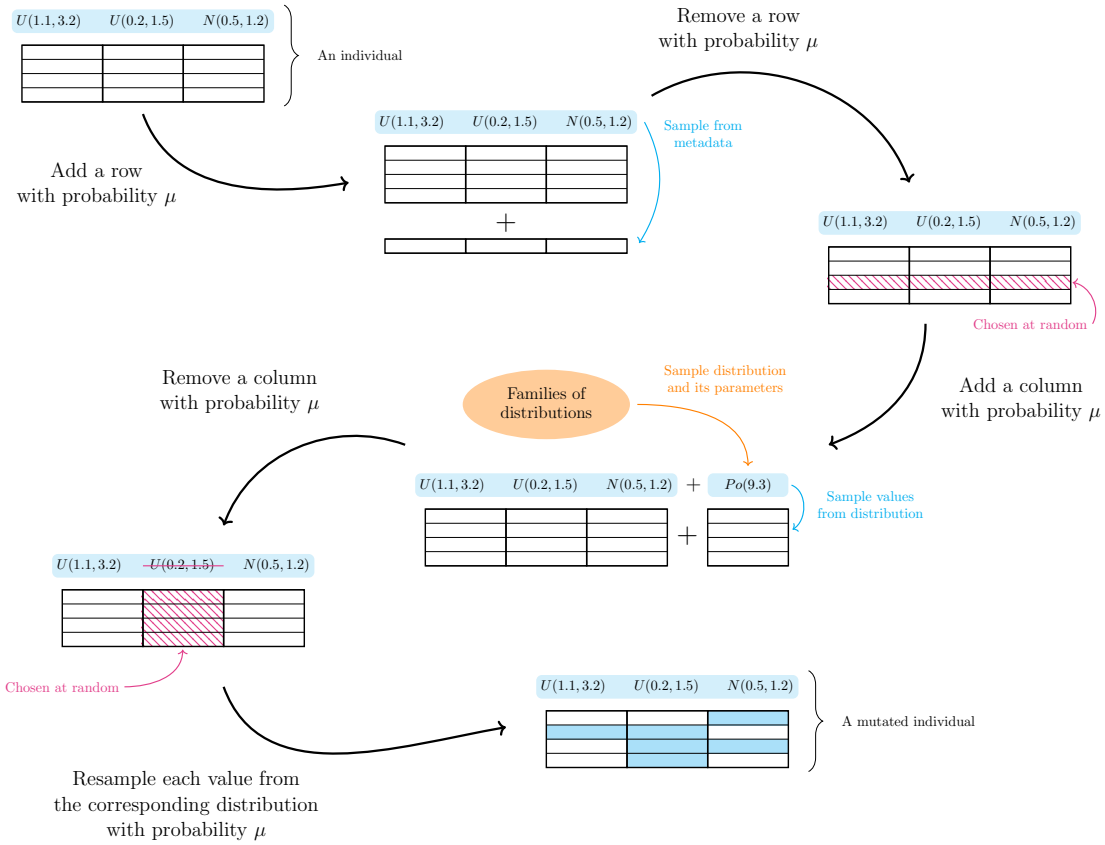

Figure 5: An example of the mutation process.

2.2.5 Shrinking

Figure 6: An diagram of the shrinking process.

2.3 Implementation

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

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.