edo Documentation

Release 0.3.6

Henry Wilde

Jan 20, 2021

CONTENTS

The `edo` library provides an evolutionary algorithm that optimises any real-valued function over a subset of the space of all possible datasets that we call *Evolutionary Dataset Optimisation*. The output of the algorithm is a bank of effective datasets for which the provided function performs well that can then be studied.

The applications of this method are varied but an important and relevant one is in learning an algorithm's strengths and weaknesses.

When determining the quality of an algorithm, the standard route is to run the comparable algorithms on a finite set of existing (or newly simulated) datasets and calculating some metric. The algorithm(s) with the smallest value of this metric are chosen to be the best performing.

An issue with this approach is that it pays little regard to the reliability and quality of the datasets being used, which begs the question: what makes a dataset "good" for an algorithm? Or, why is it that an algorithm performs well on some datasets but not others?

By passing the objective function of the algorithm to the `edo.DataOptimiser` class, questions like these can be answered by studying the properties of the resultant datasets. Beyond that, a combination of objective functions could be used to determine how an algorithm performs against any number of other algorithms. A comprehensive description of the evolutionary algorithm and an examplar case study is available at https://doi.org/10.1007/s10489-019-01592-4.

CHAPTER

ONE

TUTORIALS

In these tutorials, we will make use of a few small examples to demonstrate how `edo` is used and can be applied to various mathematical problems. Before that, however, the library needs to be installed.

1.1 Installation

The `edo` library requires Python 3.6+ and is `pip`-installable:

```
$ python -m pip install edo
```

To install from source then clone the GitHub repo:

```
$ git clone https://github.com/daffidwilde/edo.git
$ cd edo
$ python setup.py install
```

1.2 Optimising a simple function

Suppose we want to optimise the function $f(x) = x^2$ across some part of the whole real line.

We can consider each x to be a dataset with exactly one row and one column like so:

	column 0
row 0	x

For the sake of this example, let us assume our initial population has 100 individuals in it, each of whom are uniformly distributed. Further, let us assume these uniform distributions randomly sample their bounds from between -1 and 1.

1.2.1 Formulation

To formulate this in `edo` we will need the library and the `Uniform` distribution class:

[1]:
```
import edo
from edo.distributions import Uniform
```

Our fitness function takes an individual and returns the square of its only element:

[2]:
```
def xsquared(individual):

    return individual.dataframe.iloc[0, 0] ** 2
```

We configure the `Uniform` class as needed and then create a `Family` instance for it:

[3]:
```
Uniform.param_limits["bounds"] = [-1, 1]
families = [edo.Family(Uniform)]
```

Note: The `Family` class is used to handle the various instances of the distribution classes used in a run of the evolutionary algorithm (EA).

With that, we're ready to run the EA with the `DataOptimiser` class:

[4]:
```
opt = edo.DataOptimiser(
    fitness=xsquared,
    size=100,
    row_limits=[1, 1],
    col_limits=[1, 1],
    families=families,
    max_iter=5,
)

pop_history, fit_history = opt.run(random_state=0)
```

The `edo.DataOptimiser.run` method returns two things:

- `pop_history`: a nested list of all the `edo.Individual` instances organised by generation
- `fit_history`: a `pandas.DataFrame` containing the fitness scores of all the individuals

With these, we can see how close we got to the true minimum and what that individual looked like:

[5]:
```
idx = fit_history["fitness"].idxmin()
best_fitness = fit_history["fitness"].min()
generation, individual = fit_history[["generation", "individual"]].iloc[idx]

best_fitness, generation, individual
```

[5]:
```
(1.0230389458133027e-06, 0, 56)
```

[6]:
```
best = pop_history[generation][individual]
best
```

[6]:
```
Individual(dataframe=          0
0  0.001011, metadata=[Uniform(bounds=[-0.15, 0.83])])
```

So, we are definitely heading in the right the direction but we might want to take a closer look at the output of the EA.

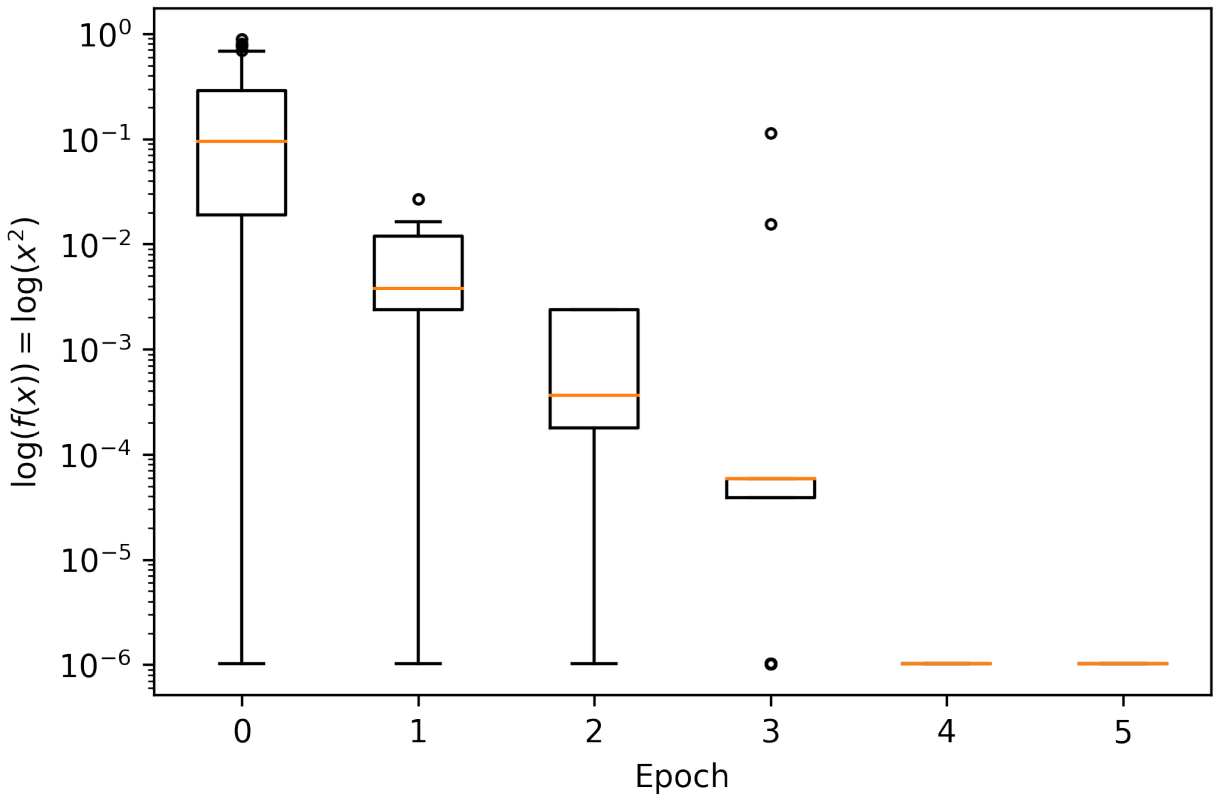
1.2.2 Visualising the results

To get a better picture of what has come out of the EA, we can plot the fitness progression and some of the individuals.

[7]:
```
import matplotlib.pyplot as plt
import numpy as np
```

[8]:
```
_, ax = plt.subplots(dpi=300)

for gen, data in fit_history.groupby("generation"):
    ax.boxplot(data["fitness"], positions=[gen], widths=0.5, sym=".")

ax.set(xlabel="Epoch", ylabel="$\log (f(x)) = \log (x^2) $", yscale="log")
```

[8]:
```
[Text(0.5, 0, 'Epoch'), Text(0, 0.5, '$\\log (f(x)) = \\log (x^2) $'), None]
```

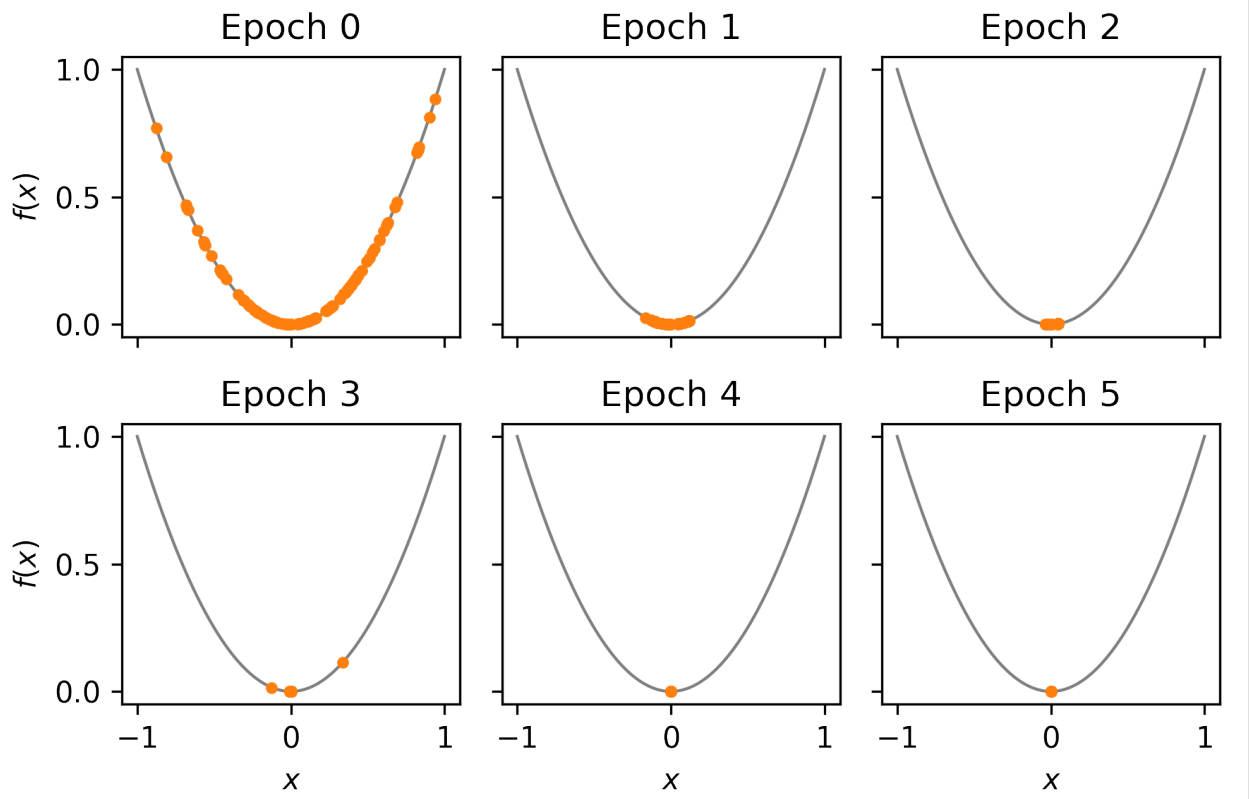


[9]:
```
_, axes = plt.subplots(2, 3, dpi=300, sharex=True, sharey=True)

axes = np.reshape(axes, 6)

for i, (generation, ax) in enumerate(zip(pop_history, axes)):

    xs = np.linspace(-1, 1, 300)
    ax.plot(xs, xs ** 2, color="tab:gray", lw=1, zorder=-1)

    xs = np.array([ind.dataframe.iloc[0, 0] for ind in generation])
    ax.scatter(xs, xs ** 2, color="tab:orange", marker=".")

    xlabel = "$x$" if i > 2 else None
```

(continues on next page)

(continued from previous page)

```
    ylabel = "$f(x)$" if i % 3 == 0 else None

    ax.set(title=f"Epoch {i}", xlabel=xlabel, ylabel=ylabel)

plt.tight_layout()
```



This looks good! The EA appears to be converging somewhere near* the optimal value.

---

* NB: *near* could be considered a little loose but this sort of simple optimisation task is not really what `edo` is for.

Generated by nbsphinx from a Jupyter notebook. Formatted using Blackbook.

1.3 Simulating the unit circle

Consider the following scenario. Let X be a dataset with two columns denoted by X_r and X_a respectively, each containing $n \in [50, 100]$ values. The values of X_a are drawn from the interval $[-2\pi, 2\pi]$ while those in X_r may take values from $[0, 2]$.

Our aim is to find a dataset X that maximises the following function:

$$f(X) = \frac{\text{Var}\,(X_a)}{\max_{x \in X_r} |x - 1|}$$

That is, a dataset which maximises the variance of one column and minimises the maximal distance from one of the other.

Such a dataset would describe the polar coordinates of some set of points along the unit circle where the points in X_r corresponds to the radii and those in X_a correspond to the angle from the origin in radians.

1.3.1 Formulation

For the sake of ease, we will assume that each column of X can be uniformly distributed between the prescribed bounds.

Then, to formulate this scenario in `edo`, we will have to create some copies of the `Uniform` class.

[1]:
```
import edo
import numpy as np
import pandas as pd
from edo.distributions import Uniform
```

[2]:
```
class RadiusUniform(Uniform):

    name = "RadiusUniform"
    param_limits = {"bounds": [0, 2]}


class AngleUniform(Uniform):

    name = "AngleUniform"
    param_limits = {"bounds": [-2 * np.pi, 2 * np.pi]}
```

To keep track of which column is which when calculating the fitness of an individual, we will use a function to extract that information from the metadata.

[3]:
```
def split_individual(individual):
    """ Separate the columns of an individual's dataframe. """

    df, metadata = individual
    names = [m.name for m in metadata]
    radii = df[names.index("RadiusUniform")]
    angles = df[names.index("AngleUniform")]

    return radii, angles


def fitness(individual):
    """ Determine the similarity of the dataset to the unit circle. """

    radii, angles = split_individual(individual)
    return angles.var() / (radii - 1).abs().max()
```

Given that this is a somewhat more complicated task than the previous tutorial, we will employ the following measures:

- A smaller proportion of the best individuals in a population will be used to create parents
- The algorithm will be run using several seeds to explore more of the search space and to have a higher degree of confidence in the output of `edo`

[4]:
```
pop_histories, fit_histories = [], []
for seed in range(5):

    families = [edo.Family(RadiusUniform), edo.Family(AngleUniform)]
```

(continues on next page)

(continued from previous page)

```
    opt = edo.DataOptimiser(
        fitness,
        size=100,
        row_limits=[50, 100],
        col_limits=[(1, 1), (1, 1)],
        families=families,
        max_iter=30,
        best_prop=0.1,
        maximise=True,
    )

    pops, fits = opt.run(random_state=seed)

    fits["seed"] = seed
    pop_histories.append(pops)
    fit_histories.append(fits)

fit_history = pd.concat(fit_histories)
```

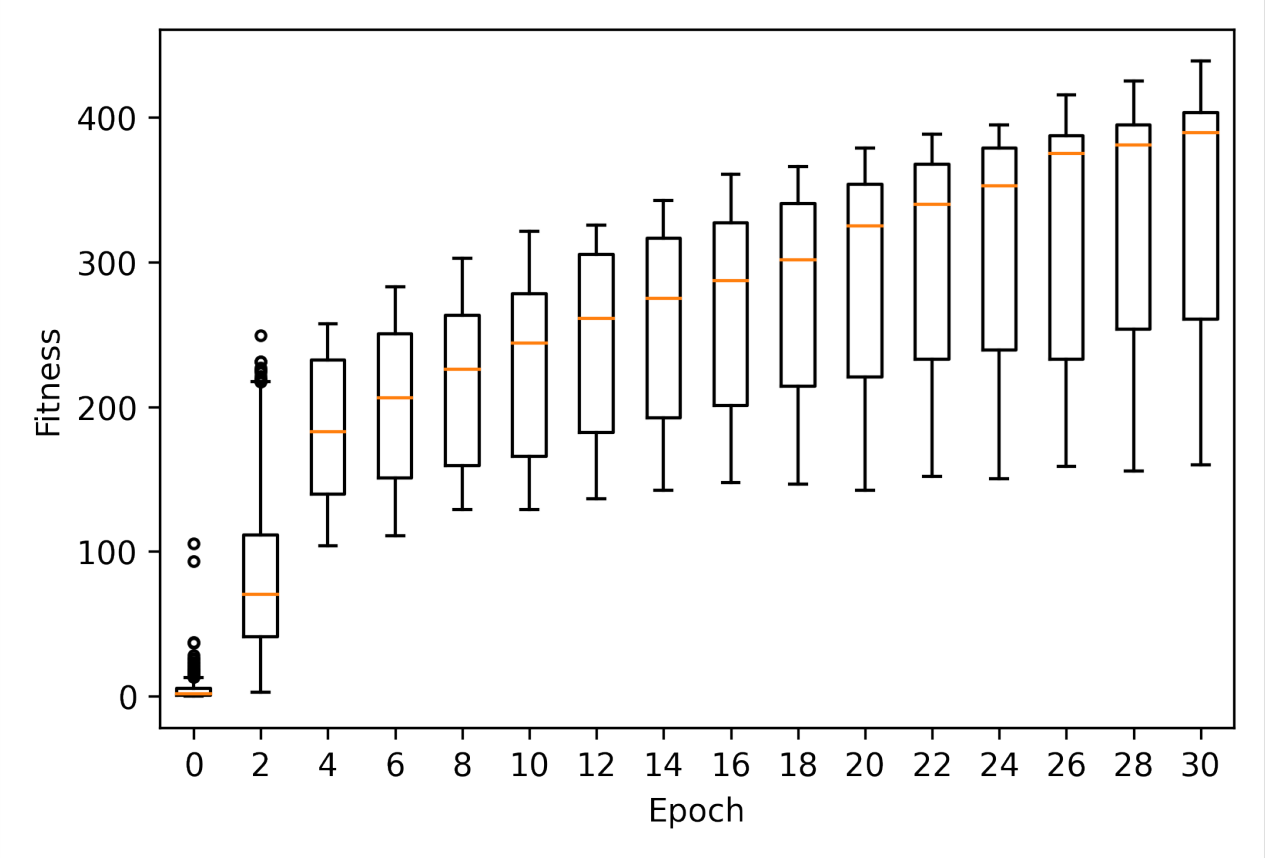
1.3.2 Visualising the results

As before, we can plot the fitness progression across these runs using `matplotlib`.

[5]:
```
import matplotlib.pyplot as plt
```

[6]:
```
_, ax = plt.subplots(dpi=300)

epochs = range(0, 31, 2)
fit = fit_history[fit_history["generation"].isin(epochs)]

xticklabels = []
for pos, (gen, data) in enumerate(fit.groupby("generation")):
    ax.boxplot(data["fitness"], positions=[pos], widths=0.5, sym=".")
    xticklabels.append(gen)

ax.set_xticklabels(xticklabels)
ax.set_xlabel("Epoch")
ax.set_ylabel("Fitness")
```

[6]:
```
Text(0, 0.5, 'Fitness')
```



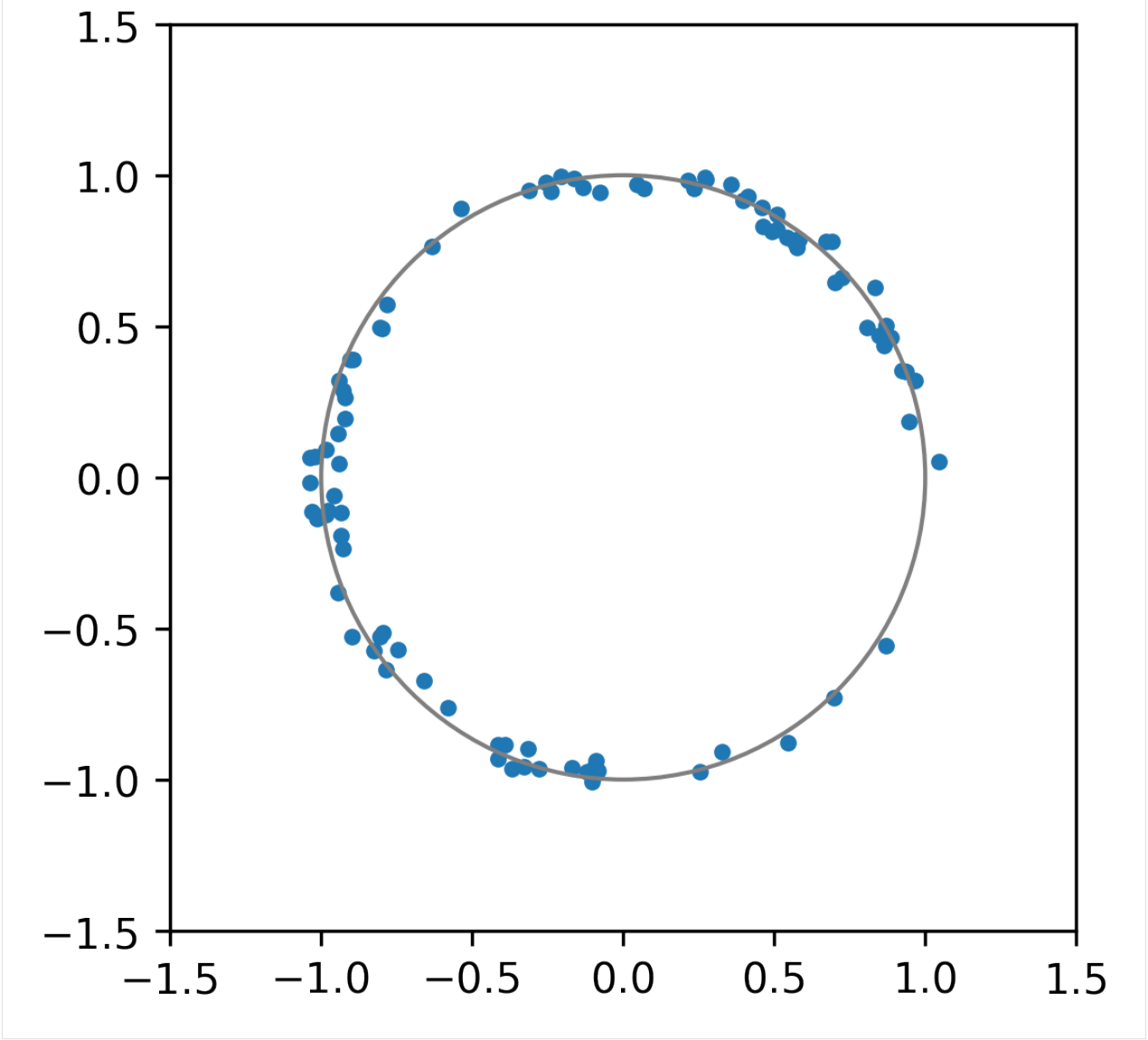
We can also take a look at the best individual across all runs.

[7]:
```
idx = fit_history["fitness"].idxmax()
seed, gen, ind = fit_history[["seed", "generation", "individual"]].iloc[idx]

best = pop_histories[seed][gen][ind]
seed, gen, ind, best
```

[7]:
```
(0,
 30,
 49,
 Individual(dataframe=           0         1
 0   1.005690 -4.928099
 1   0.971288  1.524298
 2   1.000057 -5.924903
 3   0.941059  2.931472
 4   0.984278  3.252393
 ..       ...       ...
 89  0.989021  4.615850
 90  1.022580  3.276894
 91  0.992919  3.266881
 92  1.027503 -4.978339
 93  1.004182  4.431548

 [94 rows x 2 columns], metadata=[Uniform(bounds=[0.93, 1.05]), Uniform(bounds=[-6.0,␣
↪4.74])]))
```

[8]:
```
_, ax = plt.subplots(dpi=300)

circle = plt.Circle((0, 0), 1, fill=False, linestyle="-", color="tab:gray")
ax.add_artist(circle)

radii, angles = split_individual(best)
xs, ys = radii * np.cos(angles), radii * np.sin(angles)

scatter = ax.scatter(xs, ys, marker=".")

ax.set(xlim=(-1.5, 1.5), ylim=(-1.5, 1.5), aspect="equal")
```

[8]:
```
[(-1.5, 1.5), (-1.5, 1.5), None]
```



That's a pretty good approximation to a circle. Nice.

Generated by nbsphinx from a Jupyter notebook. Formatted using Blackbook.

1.4 Breaking down k-means

In this tutorial we will be examining Lloyd's algorithm for k-means clustering. Specifically, we will be looking at datasets restricted to the plane.

This tutorial has been adapted from the case study in the publication associated with this library [WKG20].

1.4.1 Formulation

In our case, we have a two-dimensional dataset with between 20 and 60 rows taken from the unit interval that must be split into three parts, i.e. we will be clustering each dataset using k-means with $k = 3$.

The fitness of an individual will be determined by the final inertia of the clustering. The inertia is the within-cluster sum-of-squares and has an optimal value of zero.

However, given that k-means is stochastic, we will use some *smoothing* to counteract this effect and get a more reliable fitness score.

[1]:
```
import edo
import numpy as np
from edo.distributions import Uniform
from sklearn.cluster import KMeans
```

[2]:
```
def fitness(individual, num_trials):

    inertias, labels = [], []
    for seed in range(num_trials):
        km = KMeans(n_clusters=3, random_state=seed).fit(individual.dataframe)
        inertias.append(km.inertia_)
        labels.append(km.labels_)

    individual.labels = labels[np.argmin(inertias)]
    return np.min(inertias)
```

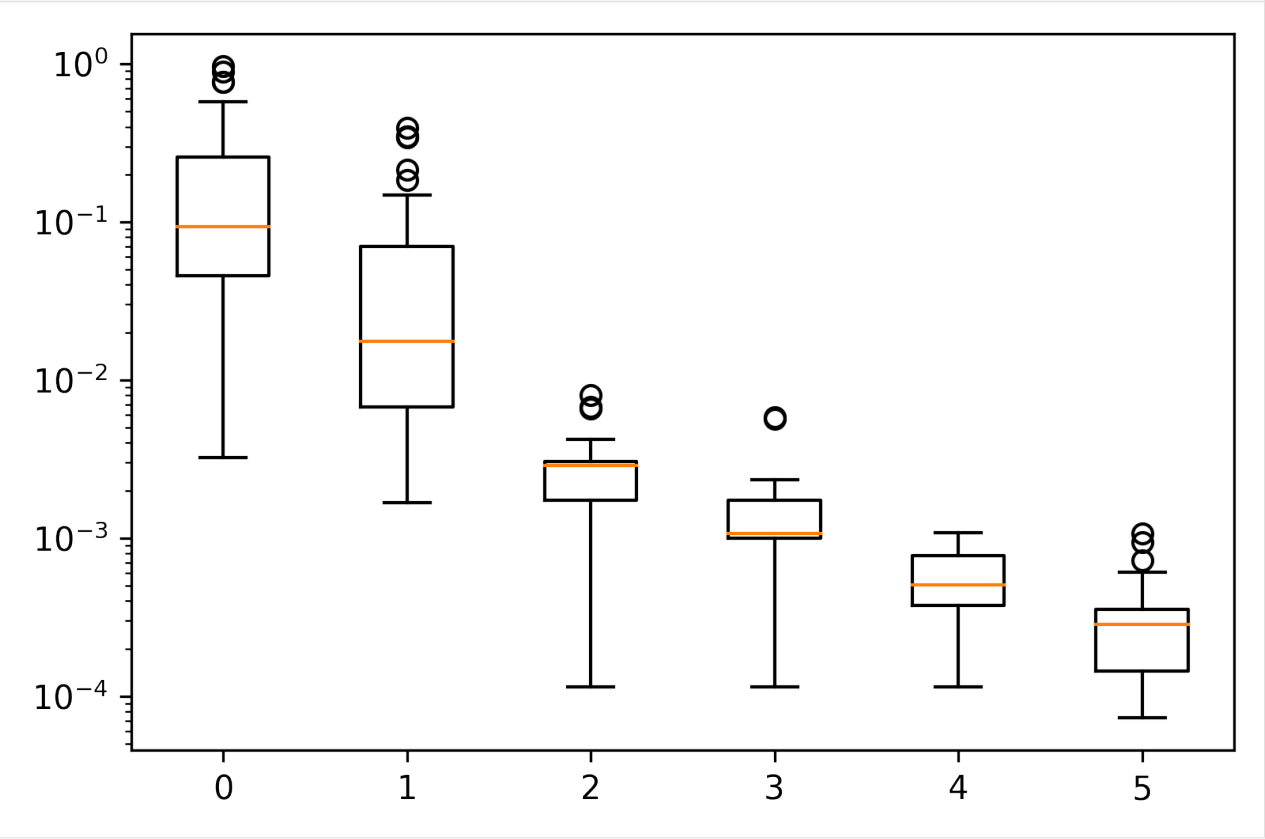
[3]:
```
Uniform.param_limits["bounds"] = [0, 1]

opt = edo.DataOptimiser(
    fitness,
    size=50,
    row_limits=[10, 50],
    col_limits=[2, 2],
    families=[edo.Family(Uniform)],
    max_iter=5,
    best_prop=0.1,
)

pop_history, fit_history = opt.run(random_state=0, fitness_kwargs={"num_trials": 5})
```

1.4.2 Visualising the results

As always, we can plot the fitness progression to get an idea of how far the EA has taken us.

[4]:
```
import matplotlib.pyplot as plt
```

[5]:
```
_, ax = plt.subplots(dpi=300)

ax.set_yscale("log")

for epoch, data in fit_history.groupby("generation"):

    ax.boxplot(data["fitness"], positions=[epoch], widths=0.5)
```



So, yes, we are moving toward that optimal value. However, an inertia of zero would be the trivial case where the three clusters are simply three unique points stacked on top of one another.

Knowing that we are in the right ballpark, it might be of use to study some individuals that were created.

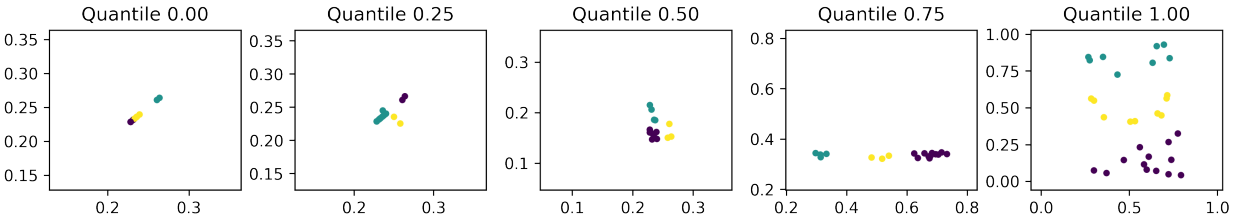
Below are the individuals representing the best to worst fitnesses at regular intervals of 25 percentiles.

[6]:
```
_, axes = plt.subplots(1, 5, figsize=(11, 2), dpi=300)

for quantile, ax in zip((0.0, 0.25, 0.5, 0.75, 1.0), axes):

    idx = (
        (fit_history["fitness"] - fit_history["fitness"].quantile(quantile))
        .abs()
        .idxmin()
```

(continues on next page)

(continued from previous page)

```
    )
    gen, ind = fit_history[["generation", "individual"]].iloc[idx]

    individual = pop_history[gen][ind]

    xs, ys = individual.dataframe[0], individual.dataframe[1]
    ax.scatter(xs, ys, c=individual.labels, s=10)

    lims = min(min(xs), min(ys)) - 0.1, max(max(xs), max(ys)) + 0.1

    ax.set(xlim=lims, ylim=lims, title=f"Quantile {quantile:.2f}")

plt.tight_layout(pad=0.5)
```



We can see here that the less-fit individuals are more dispersed and utilise more of the search space, while fitter individuals are more compact.

Low dispersion *within* clusters is to be expected since inertia measures within-cluster coherence. However, low dispersion *between* clusters is not necessarily something we want. This could be a coincidence or it could indicate that the EDO algorithm has effectively trivialised the fitness function. That is, the inertia of a dataset in a smaller domain is lower than a comparable dataset in a larger domain. This is a limitation of our fitness function that can be mitigated by changing the fitness function to, say, the silhouette coefficient, or by scaling the dataset before clustering.

Another interesting point is that the fittest individuals seem to have reduced the dimension of the search space to one dimension. Perhaps the simplest way of achieving this is to have two identical columns as has happened in the leftmost plot. By removing one dimension, the k-means algorithm is more easily able to find a centroidal Voronoi tessellation which is its aim overall. To mitigate against this, the fitness function could be adjusted to penalise datasets with high positive correlation.

These kind of adjustments and considerations are required to deeply study an algorithm or method with EDO.

Generated by nbsphinx from a Jupyter notebook. Formatted using Blackbook.

CHAPTER

TWO

HOW-TO...

2.1 Set a seed

Seeds are controlled by the `random_state` parameter in *`edo.DataOptimiser.run()`* and can be integer or an instance of `numpy.random.RandomState`.

Note: Without one, the EA here will just fall back on NumPy's innate pseudo-random number generator making any results inconsistent between runs.

Taking the example from the *first tutorial*, we can get different results by using a different seed:

```
>>> import edo
>>> from edo.distributions import Uniform
>>>
>>> Uniform.param_limits["bounds"] = [-1, 1]
>>> families = [edo.Family(Uniform)]
>>>
>>> def xsquared(ind):
...     return ind.dataframe.iloc[0, 0] ** 2
>>>
>>> opt = edo.DataOptimiser(
...     fitness=xsquared,
...     size=100,
...     row_limits=[1, 1],
...     col_limits=[1, 1],
...     families=families,
...      max_iter=5,
... )
>>> _, fit_history = opt.run(random_state=0)
>>> fit_history.head()
    fitness  generation  individual
0  0.133711           0           0
1  0.058883           0           1
2  0.682047           0           2
3  0.315748           0           3
4  0.011564           0           4
>>>
>>> opt = edo.DataOptimiser(
...     fitness=xsquared,
...     size=100,
...     row_limits=[1, 1],
...     col_limits=[1, 1],
```

(continues on next page)

(continued from previous page)

```
...     families=families,
...      max_iter=5,
... )
>>> _, fit_history = opt.run(random_state=1)
>>> fit_history.head()
    fitness  generation  individual
0  0.095955           0           0
1  0.154863           0           1
2  0.096262           0           2
3  0.081103           0           3
4  0.011293           0           4
```

2.2 Use a stopping condition

Stopping conditions allow a EA to terminate before the maximum number of iterations (generations) have been completed. Using one can save a significant amount of computational resources, and they can be based on any manner of things, such as:

- The average fitness of a generation hasn't improved in a number of generations.
- The variation between individuals in the population is reasonably low.
- When a best-case solution has been found.

We can include a stopping condition by redefining the *edo.DataOptimiser.stop()* method in a subclass to update the `converged` parameter:

```
>>> import edo
>>> import numpy as np
>>>
>>> class MyOptimiser(edo.DataOptimiser):
...     def stop(self, tolerance):
...         """
...         Stop if the population fitness variance is less than
...         ``tolerance``.
...         """
...         fitness_variance = np.var(self.pop_fitness)
...         self.converged = fitness_variance < tolerance
```

To see this in action, consider the example from the *first tutorial*:

```
>>> from edo.distributions import Uniform
>>>
>>> Uniform.param_limits["bounds"] = [-1, 1]
>>> families = [edo.Family(Uniform)]
>>>
>>> def xsquared(ind):
...     return ind.dataframe.iloc[0, 0] ** 2
>>>
>>> opt = MyOptimiser(xsquared, 100, [1, 1], [1, 1], families, max_iter=5)
```

Now we can run the algorithm as normal, and with an appropriate value of `tolerance`, it will stop before the maximum number of iterations:

```
>>> _ = opt.run(random_state=0, stop_kwargs={"tolerance": 1e-6})
>>> opt.generation
4
```

2.3 Customise the selection process

You can alter the selection discipline of the EA using two parameters in *edo.DataOptimiser*: `best_prop` and `lucky_prop`. These control how many of the best individuals and any lucky (random) individuals should be selected respectively.

For example, say we wanted to see the effect of selecting parents purely at random in each generation. Then we would set `best_prop` to be zero, and `lucky_prop` to be some value between 0 and 1:

```
>>> import edo
>>> from edo.distributions import Uniform
>>>
>>> def xsquared(ind):
...     return ind.dataframe.iloc[0, 0] ** 2
>>>
>>> opt = edo.DataOptimiser(
...     xsquared,
...     100,
...     [1, 1],
...     [1, 1],
...     [edo.Family(Uniform)],
...     best_prop=0,
...     lucky_prop=0.25,
... )
```

2.4 Customise the mutation process

The mutation process can be altered in three ways:

1. Setting the initial mutation probability
2. Adjusting (dwindling) the mutation probability over time
3. Compacting the mutation space around the best individuals

Below are some quick examples of how to do these things.

2.4.1 Setting the initial probability

This is done using the `mutation_prob` parameter in *edo.DataOptimiser*. For instance, we can remove all mutation by setting this parameter to be zero:

```
>>> import edo
>>> from edo.distributions import Uniform
>>>
>>> def xsquared(ind):
...     return ind.dataframe.iloc[0, 0] ** 2
>>>
```

(continues on next page)

(continued from previous page)

```
>>> opt = edo.DataOptimiser(
...     xsquared, 100, [1, 1], [1, 1], [edo.Family(Uniform)], mutation_prob=0
... )
```

2.4.2 Dwindling mutation probability

Sometimes an evolutionary algorithm can be thrown off once it has started converging. The purpose of the mutation process is to do this deliberately. However, as the EA progresses, mutation can make this disruption unhelpful and the population may become unpredictable or noisy.

To combat this, the *edo.DataOptimiser.dwindle()* method can be redefined in a subclass:

```
>>> class MyOptimiser(edo.DataOptimiser):
...     def dwindle(self, N=50):
...         """ Cut the mutation probability every ``N`` generations. """
...         if self.generation % N == 0:
...             self.mutation_prob /= 2
```

Any further arguments for this method should be passed in the `dwindle` parameter of *edo.DataOptimiser.run()*:

```
>>> opt = MyOptimiser(
...     xsquared,
...     100,
...     [1, 1],
...     [1, 1],
...     [edo.Family(Uniform)],
...     max_iter=1,
...     mutation_prob=1,
... )
>>>
>>> pop_history, fit_history = opt.run(dwindle_kwargs={"N": 1})
>>> opt.mutation_prob
0.5
```

2.4.3 Compacting the mutation space

The final way to alter the mutation process is to progressively reduce the mutation space via *shrinking*. This is done using the `shrinkage` parameter of *edo.DataOptimiser*:

```
>>> opt = edo.DataOptimiser(
...     xsquared, 100, [1, 1], [1, 1], [edo.Family(Uniform)], shrinkage=0.9
... )
```

2.5 Access information about an individual

Individuals are defined by three things in `edo`: a dataset, metadata about the distributions used to form the columns of that dataset, and a pseudo-random number generator for sampling from those distributions.

You can access each of these objects in the same way you would with attributes. To demonstrate, let's create an individual:

```
>>> import numpy as np
>>> from edo import Family
>>> from edo.individual import create_individual
>>> from edo.distributions import Normal, Poisson
>>>
>>> state = np.random.RandomState(0)
>>>
>>> individual = create_individual(
...     row_limits=[3, 3],
...     col_limits=[4, 4],
...     families=[Family(Normal), Family(Poisson)],
...     weights=None,
...     random_state=state,
... )
```

Then the dataframe can be accessed like this:

```
>>> individual.dataframe
           0   1  2          3
0   2.455133   8  2  13.795999
1   2.473556  13  0  -2.606494
2 -10.151318  10  2  -3.112364
```

And the metadata like this:

```
>>> individual.metadata
[Normal(mean=1.86, std=8.44), Poisson(lam=8.92), Poisson(lam=0.99), Normal(mean=-1.23,
↪ std=9.88)]
```

2.6 Customise column distributions

All distributions in `edo` have settings that can be customised. You can see all of the currently implemented distributions, and their default settings, on the *`edo.distributions`* reference page. For now, let's consider the normal distribution:

```
>>> from edo.distributions import Normal
```

The default bounds are -10 and 10 for the mean, and 0 and 10 for the standard deviation:

```
>>> Normal.param_limits
{'mean': [-10, 10], 'std': [0, 10]}
```

Changing these bounds is as simple as redefining the class attributes:

```
>>> Normal.param_limits['mean'] = [-5, 5]
>>> Normal.param_limits['std'] = [0, 1]
```

(continues on next page)

(continued from previous page)

```
>>> Normal.param_limits
{'mean': [-5, 5], 'std': [0, 1]}
```

Now all instances of normally distributed columns will have a mean between -5 and 5, and a standard deviation between 0 and 1.

In addition to this, hard bounds on the parameters can be set:

```
>>> Normal.hard_limits['mean'] = [-100, 100]
```

These hard limits are meant to stop the parameter limits from *shrinking* too far.

2.7 Implement a new column distribution

You are not limited to use only the distributions that are currently implemented in `edo`.

Say, for example, you wanted to implement a triangular distribution class. The first step would be to import the *`edo.distributions.Distribution`* base class:

```
>>> from edo.distributions import Distribution
```

Now, you define your class as normal, inheriting from the base class. The requirements on your class are as follows:

- There must be a class attribute `name` giving the name of the distribution.
- There must be a class attribute `dtype` detailing the preferred data type of the distribution.
- There must be a class attribute `hard_limits` that gives extreme limits on the parameters of the distribution.
- There must be a class attribute `param_limits` that gives the original limits on the parameters of the distribution.
- It must have a `sample` method that takes as argument: itself, an integer number of rows `nrows` and an instance of `numpy.random.RandomState`.
- The `__init__` takes only an instance of `numpy.random.RandomState`.
- The only attributes defined in the `__init__` are the parameters of that particular instance of the distribution and match the keys of `param_limits`.

So, bearing that in mind, a triangular distribution class would look something like this:

```
>>> class Triangular(Distribution):
...     """ A continuous column distribution given by the triangular
...     distribution. """
...
...     name = "Triangular"
...     dtype = float
...     hard_limits = {"bounds": [-10, 10]}
...     param_limits = {"bounds": [-10, 10]}
...
...     def __init__(self, random_state):
...
...         left, mode, right = sorted(
...             random_state.uniform(*self.param_limits["bounds"], size=3)
...         )
...         self.bounds = [left, mode, right]
```

(continues on next page)

(continued from previous page)

```
...
...     def sample(self, nrows, random_state):
...         """ Take a sample of size ``nrows`` from the triangular
...         distribution with the given bounds. """
...
...         return random_state.triangular(*self.bounds, size=nrows)
```

CHAPTER

THREE

DISCUSSION

3.1 What is an evolutionary algorithm?

Evolutionary algorithms (EAs) form a branch of population-based optimisation meta-heuristics inspired by the structures and mechanisms of evolutionary biology, and are widely recognised to be formally introduced in the 1970s by the multi-disciplined scientist John Holland in [Hol75]. The term "genetic algorithm" is typically reserved for members of a rather specific subset these search space optimisation methods that use a "chromosome" representation for the individuals in its populations.

At times, this distinction is overlooked as the defining features of this family of algorithms are their operators. More detailed summaries of them are given in this section of the documentation but briefly they are:

- *Selection*: The process by which individuals are plucked from a population to be carried forward into the next generation after being blended together to form offspring with (hopefully) favourable qualities.
- *Crossover*: The blending process. Here, pairs of individuals are combined to form one or more new individuals to be added into the next generation.
- *Mutation*: Once a new individual has been made and before they are introduced into the population, there is a chance that they can be altered.

The general structure of a basic EA is given below:

3.2 Individuals and the population

3.2.1 Representation

At the beginning of a EA, a collection of *individuals* are generated. This collection is called a *population* or *generation*. Typically, these individuals are created by randomly sampling parameters from a search space – though other methods exist. Each individual represents a solution to the problem at hand; in the case of genetic algorithms, this representation is a bit string or *chromosome* as it imitates an actual genetic chromosome.

The EDO method deals with the creation and adjustment of entire datasets. As such, there is no encoding to that aspect of an individual. In addition to this dataset, individuals are represented by a list of probability distributions. Each of these distributions acts a set of instructions on how to create, inherit from and mutate the values of the corresponding column in the dataset. These objects are stored as class attributes in the `edo.individual.Individual` class.

3.2.2 Creation

The parameter space from which individual datasets are generated is defined by the `row_limits`, `col_limits` and `families` parameters in *`edo.DataOptimiser`*. The first two describe the dimensional limits of the dataset (i.e. how tall or wide it can be) while the latter is a pool of families of probability distributions with which to fill in the dataset.

Note: You can consider a family as a sort of factory that manages and distributes independent copies of a particular probability distribution.

The step of creating an individual is to sample a number of rows and columns, creating a "skeleton" to be filled. Then, according to any family-specific limits defined in `col_limits`, each column is filled with values by:

1. Sampling a family from the pool.
2. Creating a new (or using an existing) copy of that family's distribution.
3. Creating an instance of that distribution and sampling values from it to fill the column.

A diagram depicting this process is given below:

3.3 Operators

3.3.1 Selection

The selection operator defines the process by which individuals are chosen from the current population to act as the "parents" of the next generation. Almost always, selection operators determine whether an individual should become a parent based on their fitness.

In EDO, a proportion of the best performing individuals are taken from a population into the next. You can also choose to include some randomly selected, or "lucky", individuals to be carried forward with the fittest members of the population, if there are any still available.

This selection method is a variant of the classic truncation selection method with a fixed selection proportion. When lucky individuals are included, a level of noise is introduced which can increase convergence rates [Jeb13].

Note: Taking lucky individuals should be done with caution as the associated noise can also throw the algorithm off. The use of this functionality is only encouraged for particularly complex contexts where you are unable to obtain satisfactory results otherwise.

3.3.2 Crossover

A crossover operator defines how two individuals should be combined to create a new individual (or individuals). Importantly, the crossover operator allows for the preservation of preferable characteristics found by the genetic algorithm.

In EDO, the crossover operator returns exactly one individual from a pair of parents. As is discussed **elsewhere_**, an individual is created by sampling its dimensions and then its values. Creating an offspring is done in the same way except it inherits these characteristics from its parents:

1. Inherit a number of rows and a number of columns from either parent, independently and uniformly. This is the skeleton of the dataset.
2. Pool together the columns (and respective column distributions) from both parents.
3. Sample from this pool uniformly (and without replacement) to fill in the columns of the offspring's dataset. Now the dataset has values and instructions on how to manipulate it.
4. Remove surplus rows as required, and fill in any missing values using the corresponding column information. This is now a complete individual.

Before this offspring is added to population, it must undergo *mutation*.

Example

Consider the following example where two individuals are created:

```
>>> import numpy as np
>>> from edo import Family
>>> from edo.distributions import Poisson
>>> from edo.individual import create_individual
>>> from edo.operators import crossover
>>>
>>> row_limits, col_limits = [1, 3], [2, 3]
>>> families = [Family(Poisson)]
>>> states = [np.random.RandomState(i) for i in range(2)]
>>>
>>> parents = [
...     create_individual(
...         row_limits, col_limits, families, weights=None, random_state=state
...     ) for state in states
... ]
```

These individuals' dataframes look like this:

```
>>> parents[0].dataframe
    0  1   2
0  12  0  12
>>> parents[1].dataframe
   0  1  2
0  0  5  7
1  4  4  9
```

And their metadata like this:

```
>>> parents[0].metadata
[Poisson(lam=7.15), Poisson(lam=0.87), Poisson(lam=8.33)]
>>> parents[1].metadata
[Poisson(lam=7.2), Poisson(lam=3.97), Poisson(lam=8.01)]
```

Now, we create a PRNG for the offspring and apply the crossover:

```
>>> state = np.random.RandomState(2)
>>> offspring = crossover(*parents, col_limits, families, state)
>>>
>>> offspring.dataframe
   0   1  2
0  0  12  7
```

(continues on next page)

(continued from previous page)

```
>>> offspring.metadata
[Poisson(lam=7.2), Poisson(lam=8.33), Poisson(lam=8.01)]
```

3.3.3 Mutation

To maintain a level of variety in a population and to force the evolutionary algorithm to explore more of the search space, new individuals are mutated immediately after their creation during the crossover process.

The mutation process in EDO is not quite as simple as in a traditional genetic algorithm. This is due to the representation of individuals. An individual is mutated in the following way:

1. Mutate the number of rows and columns by adding and/or removing a line from each axis with the same probability. Lines are removed at random. Rows are added by sampling a new value from each current column distribution and adding them to the bottom of the dataset. Columns are added in the same way as in the *creation process*. Note that the number of rows and columns will not mutate beyond the bounds passed in `col_limits`.
2. With the dimensions of the dataset mutated, each value in the dataset is mutated using the same mutation probability. A value is mutated by replacing it with a single value sampled from the distribution associated with its column.

Example

Consider the following mutation of an individual:

```
>>> import numpy as np
>>> from edo import Family
>>> from edo.distributions import Poisson
>>> from edo.individual import create_individual
>>> from edo.operators import mutation
>>>
>>> row_limits, col_limits = [3, 5], [2, 5]
>>> families = [Family(Poisson)]
>>> state = np.random.RandomState(0)
>>>
>>> individual = create_individual(
...     row_limits, col_limits, families, weights=None, random_state=state
... )
```

The individual looks like this:

```
>>> individual.dataframe
    0  1  2  3  4
0  12  8  4  1  7
1   6  6  5  1  5
2   8  7  7  1  3
>>> individual.metadata
[Poisson(lam=7.15), Poisson(lam=7.74), Poisson(lam=6.53), Poisson(lam=2.83),␣
↪Poisson(lam=6.92)]
```

Now we can mutate this individual after setting the mutation probability. This is deliberately large to make for a substantial mutation:

```
>>> mutation_prob = 0.7
>>> mutant = mutation(individual, mutation_prob, row_limits, col_limits, families)
```

This gives the following individual:

```
>>> mutant.dataframe
    0  1  2  3
0   8  4  1  5
1  11  3  4  5
2   9  7  3  3
>>> mutant.metadata
[Poisson(lam=7.74), Poisson(lam=6.53), Poisson(lam=2.83), Poisson(lam=6.92)]
```

3.3.4 Shrinkage

It is possible to reduce the search space of the algorithm forcibly by including some *shrinkage* or *compacting*. Under this operation, each distribution family has its parameter limits reduced by those present in the parents from a generation according to a power law presented in [AS17].

Warning: This can produce reductive results and is not recommended in normal use.

3.4 Setting seeds

Evolutionary algorithms are meta-heuristics and so are stochastic in nature. Therefore, it is a good idea to set a seed for the pseudo-random number generator when running the algorithm. By doing this, the same run can be executed again and again, and you will always obtain the same results.

A short example of how to do this is given *here*.

3.5 Smoothing

Sometimes the fitness function you wish to use will have a stochastic element. This means that when the fitness of an individual is calculated, it will not necessarily be the same on another run of the algorithm, or representative at all. However, this effect can be handled by use of a technique called smoothing. An example of this is used in the *k-means tutorial*.

There are many different ways of implementing smoothing within a fitness function; in the tutorial several repetitions are done using their own random seeds. These repetitions are then amalgamated to give a representative value for the objective function. Depending on the problem domain you are investigating, more robust or specific methods may be available to you such as recursive Bayesian estimation.

CHAPTER

FOUR

REFERENCE

4.1 edo

4.1.1 edo package

Subpackages

edo.distributions package

Submodules

edo.distributions.base module

The base class from which all distributions inherit.

class edo.distributions.base.**Distribution**
Bases: object

An abstract base class for all currently implemented distributions and those defined by users.

abstract sample(*nrows=None*, *random_state=None*)
A placeholder function for sampling from the distribution.

edo.distributions.continuous module

All currently implemented continuous distributions.

class edo.distributions.continuous.**Gamma**(*random_state*)
Bases: *edo.distributions.base.Distribution*

The gamma distribution class.

Parameters

random_state [numpy.random.RandomState] The PRNG used to sample instance parameters from `param_limits`.

Attributes

name [str] Name of the distribution, `"Gamma"`.

dtype [float] Convert a string or number to a floating point number, if possible.

param_limits [dict] A dictionary of limits on the distribution parameters. Defaults to `[0, 10]` for both `alpha` and `theta`.

alpha [float] The shape parameter sampled from `param_limits["alpha"]`. *Instance attribute.*

theta [float] The scale parameter sampled from `param_limits["theta"]`. *Instance attribute.*

dtype
alias of `float`

hard_limits = {'alpha': [0, 10], 'theta': [0, 10]}

name = 'Gamma'

param_limits = {'alpha': [0, 10], 'theta': [0, 10]}

sample(*nrows*, *random_state*)
Take a sample of size `nrows` from the gamma distribution using the provided `np.random.RandomState` instance.

class edo.distributions.continuous.**Normal**(*random_state*)
Bases: *edo.distributions.base.Distribution*

The normal distribution class.

Parameters

random_state [numpy.random.RandomState] The PRNG used to sample instance parameters from `param_limits`.

Attributes

name [str] Name of the distribution, `"Normal"`.

dtype [float] Convert a string or number to a floating point number, if possible.

param_limits [dict] A dictionary of limits on the distribution parameters. Defaults to `[-10, 10]` for `mean` and `[0, 10]` for `std`.

mean [float"] The mean, sampled from `param_limits["mean"]`. *Instance attribute.*

std [float] The standard deviation, sampled from `param_limits["std"]`. *Instance attribute.*

dtype
alias of `float`

hard_limits = {'mean': [-10, 10], 'std': [0, 10]}

name = 'Normal'

param_limits = {'mean': [-10, 10], 'std': [0, 10]}

sample(*nrows*, *random_state*)
Take a sample of size `nrows` from the normal distribution using the provided `np.random.RandomState` instance.

class edo.distributions.continuous.**Uniform**(*random_state*)
Bases: *edo.distributions.base.Distribution*

The uniform distribution class.

Parameters

random_state [numpy.random.RandomState] The PRNG used to sample instance parameters from `param_limits`.

Attributes

name [str] Name of the distribution, `Uniform`.

dtype [float] Convert a string or number to a floating point number, if possible.

param_limits [dict] A dictionary of limits on the distribution parameters. Defaults to `[-10, 10]` for `bounds`.

bounds [list of float] The lower and upper bounds of the distribution. *Instance attribute*.

dtype
alias of `float`

hard_limits = {'bounds': [-10, 10]}

name = 'Uniform'

param_limits = {'bounds': [-10, 10]}

sample(*nrows*, *random_state*)
Take a sample of size `nrows` from the uniform distribution using the provided `np.random.RandomState` instance.

edo.distributions.discrete module

All currently implemented discrete distribution classes.

class edo.distributions.discrete.**Bernoulli**(*random_state*)
Bases: *edo.distributions.base.Distribution*

The Bernoulli distribution class, i.e. a binomial distribution with exactly one trial.

Parameters

random_state [numpy.random.RandomState] The PRNG used to sample instance parameters from `param_limits`.

Attributes

name [str] Name of the distribution, `"Bernoulli"`.

dtype [int] int([x]) -> integer

param_limits [dict] A dictionary of the limits on the distribution parameter. Defaults to `[0, 1]` for `prob`.

prob [float] The success probability, sampled from `param_limits["prob"]`. *Instance attribute.*

dtype
alias of `int`

hard_limits = {'prob': [0, 1]}

name = 'Bernoulli'

param_limits = {'prob': [0, 1]}

sample(*nrows*, *random_state*)
Take a sample of size `nrows` from the Bernoulli distribution using the provided `np.random.RandomState` instance.

class edo.distributions.discrete.**Poisson**(*random_state*)
Bases: *edo.distributions.base.Distribution*

The Poisson distribution class.

Parameters

random_state [numpy.random.RandomState] The PRNG used to sample instance parameters from `param_limits`.

Attributes

name [str] Name of distribution, `"Poisson"`.

dtype [int] int([x]) -> integer

param_limits [dict] A dictionary of the limits of the distribution parameter. Defaults to `[0, 10]` for `lam`.

lam [float] The rate parameter, sampled from `param_limits["lam"]`. *Instance attribute.*

dtype
alias of `int`

hard_limits = {'lam': [0, 10]}

name = 'Poisson'

param_limits = {'lam': [0, 10]}

sample(*nrows*, *random_state*)
Take a sample of size `nrows` from the Poisson distribution using the provided `np.random.RandomState` instance.

Module contents

Top-level imports for the *edo.distributions* subpackage.

class edo.distributions.**Bernoulli**(*random_state*)
Bases: *edo.distributions.base.Distribution*

The Bernoulli distribution class, i.e. a binomial distribution with exactly one trial.

Parameters

random_state [numpy.random.RandomState] The PRNG used to sample instance parameters from `param_limits`.

Attributes

name [str] Name of the distribution, `"Bernoulli"`.

dtype [int] int([x]) -> integer

param_limits [dict] A dictionary of the limits on the distribution parameter. Defaults to `[0, 1]` for `prob`.

prob [float] The success probability, sampled from `param_limits["prob"]`. *Instance attribute.*

dtype
alias of `int`

hard_limits = {'prob': [0, 1]}

name = 'Bernoulli'

param_limits = {'prob': [0, 1]}

sample(*nrows*, *random_state*)
Take a sample of size `nrows` from the Bernoulli distribution using the provided `np.random.RandomState` instance.

class edo.distributions.**Distribution**
Bases: `object`

An abstract base class for all currently implemented distributions and those defined by users.

abstract sample(*nrows=None*, *random_state=None*)
A placeholder function for sampling from the distribution.

class edo.distributions.**Gamma**(*random_state*)
Bases: *edo.distributions.base.Distribution*

The gamma distribution class.

Parameters

random_state [numpy.random.RandomState] The PRNG used to sample instance parameters from `param_limits`.

Attributes

name [str] Name of the distribution, `"Gamma"`.

dtype [float] Convert a string or number to a floating point number, if possible.

param_limits [dict] A dictionary of limits on the distribution parameters. Defaults to `[0, 10]` for both `alpha` and `theta`.

alpha [float] The shape parameter sampled from `param_limits["alpha"]`. *Instance attribute.*

theta [float] The scale parameter sampled from `param_limits["theta"]`. *Instance attribute.*

dtype
alias of `float`

hard_limits = {'alpha': [0, 10], 'theta': [0, 10]}

name = 'Gamma'

param_limits = {'alpha': [0, 10], 'theta': [0, 10]}

sample(*nrows*, *random_state*)
Take a sample of size `nrows` from the gamma distribution using the provided `np.random.RandomState` instance.

class edo.distributions.**Normal**(*random_state*)
Bases: *edo.distributions.base.Distribution*

The normal distribution class.

Parameters

random_state [numpy.random.RandomState] The PRNG used to sample instance parameters from `param_limits`.

Attributes

name [str] Name of the distribution, `"Normal"`.

dtype [float] Convert a string or number to a floating point number, if possible.

param_limits [dict] A dictionary of limits on the distribution parameters. Defaults to `[-10, 10]` for `mean` and `[0, 10]` for `std`.

mean [float"] The mean, sampled from `param_limits["mean"]`. *Instance attribute.*

std [float] The standard deviation, sampled from `param_limits["std"]`. *Instance attribute.*

dtype
alias of `float`

hard_limits = {'mean': [-10, 10], 'std': [0, 10]}

name = 'Normal'

param_limits = {'mean': [-10, 10], 'std': [0, 10]}

sample(*nrows*, *random_state*)
Take a sample of size `nrows` from the normal distribution using the provided `np.random.RandomState` instance.

class edo.distributions.**Poisson**(*random_state*)
Bases: *edo.distributions.base.Distribution*

The Poisson distribution class.

Parameters

random_state [numpy.random.RandomState] The PRNG used to sample instance parameters from `param_limits`.

Attributes

name [str] Name of distribution, `"Poisson"`.

dtype [int] int([x]) -> integer

param_limits [dict] A dictionary of the limits of the distribution parameter. Defaults to `[0, 10]` for `lam`.

lam [float] The rate parameter, sampled from `param_limits["lam"]`. *Instance attribute.*

dtype
alias of `int`

hard_limits = {'lam': [0, 10]}

name = 'Poisson'

param_limits = {'lam': [0, 10]}

sample(*nrows*, *random_state*)
Take a sample of size `nrows` from the Poisson distribution using the provided `np.random.RandomState` instance.

class edo.distributions.**Uniform**(*random_state*)
Bases: *edo.distributions.base.Distribution*

The uniform distribution class.

Parameters

random_state [numpy.random.RandomState] The PRNG used to sample instance parameters from `param_limits`.

Attributes

name [str] Name of the distribution, `Uniform`.

dtype [float] Convert a string or number to a floating point number, if possible.

param_limits [dict] A dictionary of limits on the distribution parameters. Defaults to `[-10, 10]` for `bounds`.

bounds [list of float] The lower and upper bounds of the distribution. *Instance attribute*.

dtype
alias of `float`

hard_limits = {'bounds': [-10, 10]}

name = 'Uniform'

param_limits = {'bounds': [-10, 10]}

sample(*nrows*, *random_state*)
Take a sample of size `nrows` from the uniform distribution using the provided `np.random.RandomState` instance.

edo.operators package

Submodules

edo.operators.crossover module

Functions for the crossover process.

`edo.operators.crossover.`**crossover**(*parent1*, *parent2*, *col_limits*, *families*, *random_state*, *prob=0.5*)
Blend the information from two parents to create a new `Individual`. Dimensions are inherited first, forming a "skeleton" that is filled with column-metadata pairs. These pairs are selected from either parent uniformly. Missing values are filled in as necessary.

Parameters

parent1 [Individual] The first individual to be blended.

parent2 [Individual] The second individual to be blended.

col_limits [list] Lower and upper bounds on the number of columns `offspring` can have. Used in case of tuple limits.

families [list] Families of distributions with which to create new columns. Used in case of tuple column limits.

random_state [numpy.random.RandomState] The PRNG associated with the offspring.

prob [float, optional] The cut-off probability with which to inherit dimensions from `parent1` over `parent2`.

Returns

——-

offspring [Individual] A new individual formed from the dimensions and columns of its parents.

edo.operators.mutation module

Functions related to the mutation operator.

edo.operators.mutation.**mutate_ncols**(*dataframe*, *metadata*, *col_limits*, *families*, *weights*, *random_state*, *prob*)
Mutate the number of columns an individual has by adding a new column and/or dropping a column at random. In either case, the bounds defined in `col_limits` cannot be exceeded.

edo.operators.mutation.**mutate_nrows**(*dataframe*, *metadata*, *row_limits*, *random_state*, *prob*)
Mutate the number of rows an individual has by adding a new row and/or dropping a row at random so as not to exceed the bounds of `row_limits`.

edo.operators.mutation.**mutate_values**(*dataframe*, *metadata*, *random_state*, *prob*)
Iterate over the values of `dataframe` and mutate them each with probability `prob`. Mutating a value is done by resampling from the associated column distribution in `metadata`.

edo.operators.mutation.**mutation**(*individual*, *prob*, *row_limits*, *col_limits*, *families*, *weights=None*)
Mutate an individual. Here, the characteristics of an individual can be split into two parts: their dimensions, and their values. Each of these parts is mutated in a different way using the same probability, `prob`.

Parameters

individual [Individual] The individual to be mutated.

prob [float] The probability with which any characteristic of `individual` should be mutated.

row_limits [list] Lower and upper limits on the number of rows an individual can have.

col_limits [list] Lower and upper limits on the number of columns an individual can have.

families: list Families of distributions with which to create new columns.

weights [list, optional] Probabilities with which to sample a distribution `families`. If `None`, sample uniformly.

Returns

mutant [Individual] A (potentially) mutated individual.

edo.operators.selection module

The selection operator.

edo.operators.selection.**selection**(*population*, *pop_fitness*, *best_prop*, *lucky_prop*, *random_state*, *maximise=False*)
Given a population, select a proportion of the "best" individuals and another of the "lucky" individuals (if they are available) to form a set of potential parents.

Parameters

population [list] All current individuals.

pop_fitness [list] The fitness of each individual in `population`.

best_prop [float] The proportion of the fittest individuals in `population` to be selected.

lucky_prop [float] The proportion of lucky individuals left in `population` to be selected after the "best" have been selected.

maximise [bool, optional] Determines whether an individual's fitness should be maximal or not. Defaults to `False`.

Returns

parents [dict] The individuals chosen to potentially become parents and their index in the current population.

edo.operators.shrink module

Functions for shrinking the search space.

edo.operators.shrink.**shrink**(*parents*, *families*, *itr*, *shrinkage*)

Given the current progress of the evolutionary algorithm, shrink its search space, i.e. the parameter spaces for each of the distribution classes in `families`.

Parameters

parents [list of *Individual* instances] The parent individuals for this iteration.

families [list of *Distribution* instances] The families of distributions to be shrunk.

itr [int] The current iteration.

shrinkage [float] The shrinkage factor between 0 and 1.

Returns

families [list of *Distribution* instances] The altered families.

edo.operators.util module

A collection of functions for use across several operators.

edo.operators.util.**get_family_counts**(*metadata*, *families*)

Get the number of instances in *metadata* that belong to each family in *families*.

Module contents

edo.operators.**crossover**(*parent1*, *parent2*, *col_limits*, *families*, *random_state*, *prob=0.5*)

Blend the information from two parents to create a new `Individual`. Dimensions are inherited first, forming a "skeleton" that is filled with column-metadata pairs. These pairs are selected from either parent uniformly. Missing values are filled in as necessary.

Parameters

parent1 [Individual] The first individual to be blended.

parent2 [Individual] The second individual to be blended.

col_limits [list] Lower and upper bounds on the number of columns `offspring` can have. Used in case of tuple limits.

families [list] Families of distributions with which to create new columns. Used in case of tuple column limits.

random_state [numpy.random.RandomState] The PRNG associated with the offspring.

prob [float, optional] The cut-off probability with which to inherit dimensions from `parent1` over `parent2`.

Returns

———-

offspring [Individual] A new individual formed from the dimensions and columns of its parents.

edo.operators.**mutation**(*individual*, *prob*, *row_limits*, *col_limits*, *families*, *weights=None*)

Mutate an individual. Here, the characteristics of an individual can be split into two parts: their dimensions, and their values. Each of these parts is mutated in a different way using the same probability, `prob`.

Parameters

individual [Individual] The individual to be mutated.

prob [float] The probability with which any characteristic of `individual` should be mutated.

row_limits [list] Lower and upper limits on the number of rows an individual can have.

col_limits [list] Lower and upper limits on the number of columns an individual can have.

families: list Families of distributions with which to create new columns.

weights [list, optional] Probabilities with which to sample a distribution `families`. If `None`, sample uniformly.

Returns

mutant [Individual] A (potentially) mutated individual.

edo.operators.**selection**(*population*, *pop_fitness*, *best_prop*, *lucky_prop*, *random_state*, *maximise=False*)

Given a population, select a proportion of the "best" individuals and another of the "lucky" individuals (if they are available) to form a set of potential parents.

Parameters

population [list] All current individuals.

pop_fitness [list] The fitness of each individual in `population`.

best_prop [float] The proportion of the fittest individuals in `population` to be selected.

lucky_prop [float] The proportion of lucky individuals left in `population` to be selected after the "best" have been selected.

maximise [bool, optional] Determines whether an individual's fitness should be maximal or not. Defaults to `False`.

Returns

parents [dict] The individuals chosen to potentially become parents and their index in the current population.

edo.operators.**shrink**(*parents*, *families*, *itr*, *shrinkage*)

Given the current progress of the evolutionary algorithm, shrink its search space, i.e. the parameter spaces for each of the distribution classes in `families`.

Parameters

parents [list of *Individual* instances] The parent individuals for this iteration.

families [list of *Distribution* instances] The families of distributions to be shrunk.

itr [int] The current iteration.

shrinkage [float] The shrinkage factor between 0 and 1.

Returns

families [list of *Distribution* instances] The altered families.

Submodules

edo.family module

The distribution subtype handler.

class edo.family.**Family**(*distribution*, *max_subtypes=None*)
Bases: object

A class for handling all concurrent subtypes of a distribution class. A subtype is an independent copy of the distribution class allowing more of the search space to be explored.

Parameters

distribution [edo.distributions.Distribution] The distribution class to keep track of. Must be of the same form as those in edo.distributions.

max_subtypes [int] The maximum number of subtypes in the family that are currently being used in a run of the EA. There is no limit by default.

Attributes

name [str] The name of the family's distribution followed by Family.

subtype_id [int] A counter that increments when new subtypes are created. Used as an identifier for a given subtype.

subtypes [dict] A dictionary that maps subtype identifiers to their corresponding subtype. This gets updated during a run to those that are currently being used in the population.

all_subtypes [dict] A dictionary of all subtypes that have been created in the family.

random_state [np.random.RandomState] The PRNG associated with this family to be used for the sampling and creation of subtypes.

add_subtype(*subtype_name=None*, *attributes=None*)
Create a copy of the distribution class that is identical and independent of the original.

classmethod load(*distribution*, *root='.edocache'*)
Load in any existing cached subtype dictionaries for distribution and restore the subtype along with the family's random state.

make_instance(*random_state*)
Select an existing subtype at random – or create a new one if there is space available – and return an instance of that subtype.

reset(*root=None*)
Reset the family to have no subtypes and the default numpy PRNG. If root is passed then any cached information about the family is deleted.

save(*root='.edocache'*)
Save the current subtypes in the family and the family's random state in the root directory.

edo.fitness module

Functions for calculating individual and population fitness.

edo.fitness.**get_population_fitness**(*population*, *fitness*, *processes=None*, ***kwargs*)
Return the fitness of each individual in the population. This can be done in parallel by specifying a number of cores to use for independent processes.

edo.fitness.**write_fitness**(*fitness*, *generation*, *root*)
Write the generation fitness to file in the `root` directory.

edo.individual module

A collection of objects to facilitate an individual representation.

class edo.individual.**Individual**(*dataframe*, *metadata*, *random_state=None*)
Bases: `object`

A class to represent an individual in the EA.

Parameters

dataframe [pd.DataFrame or dd.DataFrame] The dataframe of the individual.

metadata [list] A list of distributions that are associated with the respective column of `dataframe`.

random_state [np.random.RandomState, optional] The PRNG for the individual. If not provided, the default PRNG is used.

Attributes

fitness [float] The fitness of the individual. Initialises as `None`.

classmethod from_file(*path*, *distributions*, *family_root='.edocache'*, *method='pandas'*)
Create an instance of `Individual` from the files at `path` and `family_root` using either `pandas` or `dask` to read in individuals. Always fall back on `pandas`.

to_file(*path*, *family_root='.edocache'*)
Write self to file.

edo.individual.**create_individual**(*row_limits*, *col_limits*, *families*, *weights*, *random_state*)
Create an individual within the limits provided.

Parameters

row_limits [list] Lower and upper bounds on the number of rows a dataset can have.

col_limits [list] Lower and upper bounds on the number of columns a dataset can have. Tuples can be used to indicate limits on the number of columns needed from each family in `families`.

families [list] A list of `edo.Family` instances handling the column distributions that can be selected from.

weights [list] A sequence of relative weights with which to sample from `families`. If `None`, then sampling is uniform.

random_state [numpy.random.RandomState] The PRNG associated with the individual to use for its random sampling.

edo.optimiser module

The evolutionary dataset optimisation algorithm class.

class edo.optimiser.**DataOptimiser**(*fitness*, *size*, *row_limits*, *col_limits*, *families*, *weights=None*, *max_iter=100*, *best_prop=0.25*, *lucky_prop=0*, *crossover_prob=0.5*, *mutation_prob=0.01*, *shrinkage=None*, *maximise=False*)

Bases: `object`

The (evolutionary) dataset optimiser. A class that generates data for a given fitness function and evolutionary parameters.

Parameters

fitness [func] Any real-valued function that at least takes an instance of `Individual` as argument. Any further arguments should be passed in the `kwargs` parameter of the `run` method.

size [int] The size of the population to create.

row_limits [list] Lower and upper bounds on the number of rows a dataset can have.

col_limits [list] Lower and upper bounds on the number of columns a dataset can have.

Tuples can also be used to specify the min/maximum number of columns there can be of each element in `families`.

families [list] A list of `edo.Family` instances that handle the distribution classes used to populate the individuals in the EA.

weights [list] A set of relative weights on how to select elements from `families`. If `None`, they will be chosen uniformly.

max_iter [int] The maximum number of iterations to be carried out before terminating.

best_prop [float] The proportion of a population from which to select the "best" individuals to be parents.

lucky_prop [float] The proportion of a population from which to sample some "lucky" individuals to be parents. Defaults to `0`.

crossover_prob [float] The probability with which to sample dimensions from the first parent over the second in a crossover operation. Defaults to `0.5`.

mutation_prob [float] The probability of a particular characteristic of an individual being mutated. If using a `dwindle` method, this is an initial probability.

shrinkage [float] The relative size to shrink each parameter's limits by for each distribution in `families`. Defaults to `None` but must be between 0 and 1 (exclusive).

maximise [bool] Determines whether `fitness` is a function to be maximised or not. Fitness scores are minimised by default.

dwindle(***kwargs*)
A placeholder for a function which can adjust (typically, reduce) the mutation probability over the run of the EA.

run(*root=None*, *random_state=None*, *processes=None*, *fitness_kwargs=None*, *stop_kwargs=None*, *dwindle_kwargs=None*)
Run the evolutionary algorithm under the given constraints.

Parameters

root [str, optional] The directory in which to write all generations to file. If `None`, nothing is written to file. Instead, every generation is kept in memory and is returned at the end. If writing to file, one generation is held in memory at a time and everything is returned upon termination as a tuple containing `dask` objects.

random_state [int or np.ran.RandomState, optional] The random seed or state for a particular run of the algorithm. If `None`, the default PRNG is used.

processes [int, optional] The number of parallel processes to use when calculating the population fitness. If `None` then a single-thread scheduler is used.

fitness_kwargs [dict, optional] Any additional parameters for the fitness function should be placed here.

stop_kwargs [dict, optional] Any additional parameters for the `stop` method should be placed here.

dwindle_kwargs [dict, optional] Any additional parameters for the `dwindle` method should be placed here.

Returns

pop_history [list] Every individual in each generation as a nested list of `Individual` instances.

fit_history [`pd.DataFrame` or `dask.dataframe.DataFrame`] Every individual's fitness in each generation.

stop(***kwargs*)

A placeholder for a function which acts as a stopping condition on the EA.

edo.population module

Functions for the creation and updating of a population.

edo.population.**create_initial_population**(*row_limits*, *col_limits*, *families*, *weights*, *random_states*)

Create an initial population for the genetic algorithm based on the given parameters.

Parameters

size [int] The number of individuals in the population.

row_limits [list] Limits on the number of rows a dataset can have.

col_limits [list] Limits on the number of columns a dataset can have.

families [list] A list of `edo.Family` instances that handle the column distribution classes.

weights [list] Relative weights with which to sample from `families`. If `None`, sampling is done uniformly.

random_states [dict] A mapping of the index of the population to a `numpy.random.RandomState` instance that is to be assigned to the individual at that index in the population.

Returns

population [list] A population of newly created individuals.

edo.population.**create_new_population**(*parents*, *population*, *crossover_prob*, *mutation_prob*, *row_limits*, *col_limits*, *families*, *weights*, *random_states*)

Given a set of potential parents to be carried into the next generation, create offspring from pairs within that set until there are enough individuals.

Parameters

parents [list] A list of *edo.individual.Individual* instances used to create new offspring.

population [list] The current population.

crossover_prob [float] The probability with which to sample dimensions from the first parent over the second during crossover.

mutation_prob [float] The probability with which to mutate a component of a newly created individual.

row_limits [list] Limits on the number of rows a dataset can have.

col_limits [list] Limits on the number of columns a dataset can have.

families [list] The `edo.Family` instances from which to draw distribution instances.

weights [list] Weights used to sample elements from `families`.

random_states [dict] The PRNGs assigned to each individual in the population.

edo.version module

The current version of the library.

Module contents

Top-level imports for the library.

class edo.**DataOptimiser**(*fitness*, *size*, *row_limits*, *col_limits*, *families*, *weights=None*, *max_iter=100*, *best_prop=0.25*, *lucky_prop=0*, *crossover_prob=0.5*, *mutation_prob=0.01*, *shrinkage=None*, *maximise=False*)

Bases: `object`

The (evolutionary) dataset optimiser. A class that generates data for a given fitness function and evolutionary parameters.

Parameters

fitness [func] Any real-valued function that at least takes an instance of `Individual` as argument. Any further arguments should be passed in the `kwargs` parameter of the `run` method.

size [int] The size of the population to create.

row_limits [list] Lower and upper bounds on the number of rows a dataset can have.

col_limits [list] Lower and upper bounds on the number of columns a dataset can have.

Tuples can also be used to specify the min/maximum number of columns there can be of each element in `families`.

families [list] A list of `edo.Family` instances that handle the distribution classes used to populate the individuals in the EA.

weights [list] A set of relative weights on how to select elements from `families`. If `None`, they will be chosen uniformly.

max_iter [int] The maximum number of iterations to be carried out before terminating.

best_prop [float] The proportion of a population from which to select the "best" individuals to be parents.

lucky_prop [float] The proportion of a population from which to sample some "lucky" individuals to be parents. Defaults to `0`.

crossover_prob [float] The probability with which to sample dimensions from the first parent over the second in a crossover operation. Defaults to `0.5`.

mutation_prob [float] The probability of a particular characteristic of an individual being mutated. If using a `dwindle` method, this is an initial probability.

shrinkage [float] The relative size to shrink each parameter's limits by for each distribution in `families`. Defaults to `None` but must be between 0 and 1 (exclusive).

maximise [bool] Determines whether `fitness` is a function to be maximised or not. Fitness scores are minimised by default.

dwindle(***kwargs*)
A placeholder for a function which can adjust (typically, reduce) the mutation probability over the run of the EA.

run(*root=None*, *random_state=None*, *processes=None*, *fitness_kwargs=None*, *stop_kwargs=None*, *dwindle_kwargs=None*)
Run the evolutionary algorithm under the given constraints.

Parameters

root [str, optional] The directory in which to write all generations to file. If `None`, nothing is written to file. Instead, every generation is kept in memory and is returned at the end. If writing to file, one generation is held in memory at a time and everything is returned upon termination as a tuple containing `dask` objects.

random_state [int or np.ran.RandomState, optional] The random seed or state for a particular run of the algorithm. If `None`, the default PRNG is used.

processes [int, optional] The number of parallel processes to use when calculating the population fitness. If `None` then a single-thread scheduler is used.

fitness_kwargs [dict, optional] Any additional parameters for the fitness function should be placed here.

stop_kwargs [dict, optional] Any additional parameters for the `stop` method should be placed here.

dwindle_kwargs [dict, optional] Any additional parameters for the `dwindle` method should be placed here.

Returns

pop_history [list] Every individual in each generation as a nested list of `Individual` instances.

fit_history [`pd.DataFrame` or `dask.dataframe.DataFrame`] Every individual's fitness in each generation.

stop(***kwargs*)
A placeholder for a function which acts as a stopping condition on the EA.

class edo.**Family**(*distribution*, *max_subtypes=None*)

Bases: object

A class for handling all concurrent subtypes of a distribution class. A subtype is an independent copy of the distribution class allowing more of the search space to be explored.

Parameters

distribution [edo.distributions.Distribution] The distribution class to keep track of. Must be of the same form as those in edo.distributions.

max_subtypes [int] The maximum number of subtypes in the family that are currently being used in a run of the EA. There is no limit by default.

Attributes

name [str] The name of the family's distribution followed by Family.

subtype_id [int] A counter that increments when new subtypes are created. Used as an identifier for a given subtype.

subtypes [dict] A dictionary that maps subtype identifiers to their corresponding subtype. This gets updated during a run to those that are currently being used in the population.

all_subtypes [dict] A dictionary of all subtypes that have been created in the family.

random_state [np.random.RandomState] The PRNG associated with this family to be used for the sampling and creation of subtypes.

add_subtype(*subtype_name=None*, *attributes=None*)

Create a copy of the distribution class that is identical and independent of the original.

classmethod load(*distribution*, *root='.edocache'*)

Load in any existing cached subtype dictionaries for distribution and restore the subtype along with the family's random state.

make_instance(*random_state*)

Select an existing subtype at random – or create a new one if there is space available – and return an instance of that subtype.

reset(*root=None*)

Reset the family to have no subtypes and the default numpy PRNG. If root is passed then any cached information about the family is deleted.

save(*root='.edocache'*)

Save the current subtypes in the family and the family's random state in the root directory.

4.2 Bibliography

This is a collection of various bibliographic items referenced in the documentation.

4.3 Citation instructions

4.3.1 Citing the library

Please use the following to cite the library:

```
@misc{edo-library,
    author = {{The EDO library developers}},
    title = {edo: <RELEASE TITLE>},
    year = <RELEASE YEAR>,
    doi = {<DOI INFORMATION>},
    url = {http://doi.org/<DOI INFORMATION>}
}
```

To check the relevant details (i.e. `RELEASE TITLE`, `RELEASE YEAR` and `DOI NUMBER`) head to the library's Zenodo page:

4.3.2 Citing the paper

If you wish to cite the paper, then use the following:

```
@article{edo-paper,
    title = {Evolutionary dataset optimisation: learning algorithm quality
             through evolution},
    author = {Wilde, Henry and Knight, Vincent and Gillard, Jonathan},
    journal = {Applied Intelligence},
    year = 2020,
    volume = 50,
    pages = {1172--1191},
    doi = {10.1007/s10489-019-01592-4},
}
```

4.4 Contributing to the library

Contributions are always welcome whether they come in the form of providing a fix for a current issue, reporting a bug or implementing an enhancement to the library code itself. Pull requests (PRs) will be reviewed and collaboration is encouraged.

To make a contribution via a PR, follow these steps:

1. Make a fork of the GitHub repo and clone your fork locally:

   ```
   $ git clone https://github.com/<your-username>/edo.git
   ```

2. Install the library in development mode. If you use Anaconda, there is a `conda` environment file (`environment.yml`) with all of the development dependencies:

   ```
   $ cd edo
   $ conda env create -f environment.yml
   $ conda activate edo-dev
   $ python setup.py develop
   ```

3. Make your changes and write tests to go with them. Ensure that they pass and you have 100% coverage:

```
$ python -m pytest --cov=edo --cov-fail-under=100 tests
```

4. Push to your fork and open a pull request.

BIBLIOGRAPHY

[AS17] Adil Amirjanov and Fahreddin Sadikoglu. Linear adjustment of a search space in genetic algorithm. *Procedia Computer Science*, 120:953–960, 2017.

[Hol75] John H. Holland. *Adaption in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[Jeb13] Khalid Jebari. Selection methods for genetic algorithms. *International Journal of Emerging Sciences*, 3:333–344, 2013.

[WKG20] Henry Wilde, Vincent Knight, and Jonathan Gillard. Evolutionary dataset optimisation: learning algorithm quality through evolution. *Applied Intelligence*, 50:1172–1191, 2020. doi:10.1007/s10489-019-01592-4.

PYTHON MODULE INDEX

e

INDEX

H

I

L

M

N

P

R

S

T

U

W