
EDAspy

Release 1.0.0

Vicente P. Soloviev

Sep 14, 2022

CONTENTS:

| | | |
|----------|--|-----------|
| 1 | EDAspy | 1 |
| 1.1 | Introduction | 1 |
| 1.2 | Examples | 2 |
| 1.3 | Getting started | 2 |
| 1.4 | Build from Source | 2 |
| 1.4.1 | Prerequisites | 2 |
| 1.4.2 | Building | 2 |
| 1.5 | Testing | 2 |
| 2 | Examples | 3 |
| 2.1 | Using UMDAc for continuous optimization | 3 |
| 2.2 | Using UMDAd for feature selection in a toy example | 4 |
| 2.3 | Building my own EDA implementation | 6 |
| 2.4 | Using EGNA for continuous optimization | 7 |
| 2.5 | Using EMNA for continuous optimization | 7 |
| 2.6 | Using EDAs for time series and times series transformation selection | 8 |
| 3 | Changelog | 11 |
| 3.1 | v1.0.0 | 11 |
| 3.2 | v0.2.0 | 11 |
| 3.3 | v0.1.2 | 11 |
| 3.4 | v0.1.1 | 12 |
| 3.5 | v0.1.0 | 12 |
| 4 | Getting started | 13 |
| 4.1 | Build from Source | 13 |
| 4.1.1 | Prerequisites | 13 |
| 4.1.2 | Building | 13 |
| 4.2 | Testing | 13 |
| 5 | Formal documentation | 15 |
| 5.1 | EDAspy package | 15 |
| 5.1.1 | Subpackages | 15 |
| 5.1.1.1 | EDAspy.benchmarks package | 15 |
| 5.1.1.2 | EDAspy.optimization package | 21 |
| 5.1.1.3 | EDAspy.timeseries package | 32 |
| 5.1.2 | Module contents | 32 |
| 6 | Indices and tables | 33 |
| | Python Module Index | 35 |

1.1 Introduction

EDAspy presents some implementations of the Estimation of Distribution Algorithms (EDAs). EDAs are a type of evolutionary algorithms. Depending on the type of the probabilistic model embedded in the EDA, and the type of variables considered, we will use a different EDA implementation.

The pseudocode of EDAs is the following:

1. Random initialization of the population.
2. Evaluate each individual of the population.
3. Select the top best individuals according to cost function evaluation.
4. Learn a probabilistic model from the best individuals selected.
5. Sampled another population.
6. If stopping criteria is met, finish; else, go to 2.

EDAspy allows to create a custom version of the EDA. Using the modular probabilistic models and the initializers, this can be embedded into the EDA baseline and used for different purposes. If this fits you, take a look on the examples section to the EDACustom example.

EDAspy also incorporates a set of benchmarks in order to compare the algorithms trying to minimize these cost functions.

The following implementations are available in EDAspy:

- **UMDA**: Univariate Marginal Distribution Algorithm binary. It can be used as a simple example of EDA where the variables are binary and there are not dependencies between variables. Some usages include feature selection, for example.
- **UMDAc**: Univariate Marginal Distribution Algorithm continuous. In this EDA all the variables assume a Gaussian distribution and there are not dependencies considered between the variables. Some usages include hyperparameter optimization, for example.
- **EGNA**: Estimation of Gaussian Distribution Algorithm. This is a complex implementation in which dependencies between the variables are considered during the optimization. In each iteration, a Gaussian Bayesian network is learned and sampled. The variables in the model are assumed to be Gaussian and also dependencies between them. This implementation is focused in continuous optimization.
- **EMNA**: Estimation of Multivariate Normal Algorithm. This is a similar implementation to EGNA, in which instead of using a Gaussian Bayesian network, a multivariate Gaussian distribution is iteratively learned and sampled. As in EGNA, the dependencies between variables are considered and assumed to be linear Gaussian. This implementation is focused in continuous optimization.

- Categorical EDA. In this implementation we consider some independent categorical variables. Some usages include portfolio optimization, for example.

1.2 Examples

Some examples are available in <https://github.com/VicentePerezSoloviev/EDAspy/tree/master/notebooks>

1.3 Getting started

For installing EDAspy from Pypi execute the following command using pip:

```
pip install EDAspy
```

1.4 Build from Source

1.4.1 Prerequisites

- Python 3.6, 3.7, 3.8 or 3.9.
- Pybnesian, numpy, pandas.

1.4.2 Building

Clone the repository:

```
git clone https://github.com/VicentePerezSoloviev/EDAspy.git
cd EDAspy
git checkout v1.0.0 # You can checkout a specific version if you want
python setup.py install
```

1.5 Testing

The library contains tests that can be executed using `pytest`. Install it using pip:

```
pip install pytest
```

Run the tests with:

```
pytest
```

EXAMPLES

Some easy examples are shown in this section. To see the following code explained and executed in Jupyter Notebooks visit the GitHub repository where all notebooks are available or access through the following links.

2.1 Using UMDAc for continuous optimization

In this notebook we use the UMDAc implementation for the optimization of a cost function. This cost function that we are using in this notebook is a wellknown benchmark and is available in EDAspy.

```
from EDAspy.optimization.univariate import UMDAc
from EDAspy.benchmarks import ContinuousBenchmarkingCEC14
import matplotlib.pyplot as plt
```

We will be using 10 variables for the optimization.

```
n_vars = 10
benchmarking = ContinuousBenchmarkingCEC14(n_vars)
```

We initialize the EDA with the following parameters:

```
umda = UMDAc(size_gen=100, max_iter=100, dead_iter=10, n_variables=10, alpha=0.5)
# We leave bound by default
eda_result = umda.minimize(cost_function=benchmarking.cec14_4, output_runtime=True)
```

We use the `eda_result` object to extract all the desired information from the execution.

```
print('Best cost found:', eda_result.best_cost)
print('Best solution:\n', eda_result.best_ind)
```

We plot the best cost in each iteration to show how the MAE of the feature selection is reduced compared to using all the variables.

```
plt.figure(figsize = (14,6))

plt.title('Best cost found in each iteration of EDA')
plt.plot(list(range(len(eda_result.history))), eda_result.history, color='b')
plt.xlabel('iteration')
plt.ylabel('MAE')
plt.show()
```

2.2 Using UMDAd for feature selection in a toy example

In this notebooks we show a toy example for feature selection using the binary implementation of EDA in EDAspy. For this, we try to select the optimal subset of variables for a forecasting model. The metric that we use for evaluation is the Mean Absolute Error (MAE) of the subset in the forecasting model.

```
# loading essential libraries first
import statsmodels.api as sm
from statsmodels.tsa.api import VAR
import matplotlib.pyplot as plt
from sklearn.metrics import mean_absolute_error

# EDAspy libraries
from EDAspy.optimization import UMDAd
```

We will use a small dataset to show an example of usage. We usually use a Feature Subset selector when a great amount of variables is available to use.

```
# import some data
mdata = sm.datasets.macrodatab.load_pandas().data
df = mdata.iloc[:, 2:]
df.head()
```

```
variables = list(df.columns)
variable_y = 'pop' # pop is the variable we want to forecast
variables = list(set(variables) - {variable_y}) # array of variables to select among
↳ transformations
variables
```

We define a cost function which receives a dictionary with variables names as keys of the dictionary and values 1/0 if they are used or not respectively.

The functions returns the Mean Absolute Error found with the combination of variables selected.

```
def cost_function(variables_list, nob=20, maxlags=10, forecastings=10):
    """
    variables_list: array of size the number of variables, where a 1 is to choose the
    ↳ variable, and 0 to
    reject it.
    nob: how many observations for validation
    maxlags: previous lags used to predict
    forecasting: number of observations to predict

    return: MAE of the prediction with the real validation data
    """

    variables_chosen = []
    for i, j in zip(variables, variables_list):
        if j == 1:
            variables_chosen.append(i)

    data = df[variables_chosen + [variable_y]]
```

(continues on next page)

(continued from previous page)

```

df_train, df_test = data[0:-nobs], data[-nobs:]

model = VAR(df_train)
results = model.fit(maxlags=maxlags, ic='aic')

lag_order = results.k_ar
array = results.forecast(df_train.values[-lag_order:], forecastings)

variables_ = list(data.columns)
position = variables_.index(variable_y)

validation = [array[i][position] for i in range(len(array))]
mae = mean_absolute_error(validation, df_test['pop'][-forecastings:])

return mae

```

We calculate the MAE found using all the variables. This is an easy example so the difference between the MAE found using all the variables and the MAE found after optimizing the model, will be very small. But this is appreciated with more difference when large datasets are used.

```

# build the dictionary with all 1s
selection = [1]*len(variables)

mae_pre_eda = cost_function(selection)
print('MAE without using EDA:', mae_pre_eda)

```

We initialize the EDA with the following parameters, and run the optimizer over the cost function defined above. The vector of statistics is initialized to None so the EDA implementation will initialize it. If you desire to initialize it in a way to favour some of the variables you can create a numpy array with all the variables the same probability to be chosen or not (0.5), and the one you want to favour to nearly 1. This will make the EDA to choose the variable nearly always.

```

eda = UMDAd(size_gen=30, max_iter=100, dead_iter=10, n_variables=len(variables), alpha=0.
↪5, vector=None,
           lower_bound=0.2, upper_bound=0.9, elite_factor=0.2, disp=True)

eda_result = eda.minimize(cost_function=cost_function, output_runtime=True)

```

Note that the algorithm is minimizing correctly, but due to the fact that it is a toy example, there is not a high variance from the beginning to the end.

```

print('Best cost found:', eda_result.best_cost)
print('Variables chosen')
variables_chosen = []
for i, j in zip(variables, eda_result.best_ind):
    if j == 1:
        variables_chosen.append(i)
print(variables_chosen)

```

We plot the best cost in each iteration to show how the MAE of the feature selection is reduced compared to using all the variables.

```
plt.figure(figsize = (14,6))

plt.title('Best cost found in each iteration of EDA')
plt.plot(list(range(len(eda_result.history))), eda_result.history, color='b')
plt.xlabel('iteration')
plt.ylabel('MAE')
plt.show()
```

2.3 Building my own EDA implementation

In this notebook we show how the EDA can be implemented in a modular way using the components available in EDAspy. This way, the user is able to build implementations that may not be considered in the state-of-the-art. EDAspy also has the implementations of typical EDA implementations used in the state-of-the-art.

We first import from EDAspy all the needed functions and classes. To build our own EDA we use a modular class that extends the abstract class of EDA used as a baseline of all the EDA implementations in EDAspy.

```
from EDAspy.optimization.custom import EDACustom, GBN, UniformGenInit
from EDAspy.benchmarks import ContinuousBenchmarkingCEC14
```

We initialize an object with the EDACustom object. Note that, independently of the pm and init parameters, we are going to overwrite these with our own objects. If not, we have to choose which is the ID of the pm and init we want.

```
n_variables = 10
my_eda = EDACustom(size_gen=100, max_iter=100, dead_iter=n_variables, n_variables=n_
↳ variables, alpha=0.5,
                    elite_factor=0.2, disp=True, pm=4, init=4, bounds=(-50, 50))

benchmarking = ContinuousBenchmarkingCEC14(n_variables)
```

We now implement our initializer and probabilistic model and add these to our EDA.

```
my_gbn = GBN([str(i) for i in range(n_variables)])
my_init = UniformGenInit(n_variables)

my_eda.pm = my_gbn
my_eda.init = my_init
```

We run our EDA in one of the benchmarks that is implemented in EDAspy.

```
eda_result = my_eda.minimize(cost_function=benchmarking.cec14_4)
```

We can access the results in the result object:

```
print(eda_result)
```

2.4 Using EGNA for continuous optimization

In this notebook we use the EGNA approach to optimize a wellknown benchmark. Note that EGNA learns and sampled a GBN in each iteration. Import the algorithm and the benchmarks from EDAspy

```
from EDAspy.optimization import EGNA
from EDAspy.benchmarks import ContinuousBenchmarkingCEC14
```

We will be using a benchmark with 10 variables.

```
n_vars = 10
benchmarking = ContinuousBenchmarkingCEC14(n_vars)
```

We initialize the EDA with the following parameters:

```
egna = EGNA(size_gen=300, max_iter=100, dead_iter=20, n_variables=10,
            landscape_bounds=(-60, 60))

eda_result = enga.minimize(benchmarking.cec14_4, True)
```

We plot the best cost found in each iteration of the algorithm.

```
plt.figure(figsize = (14,6))

plt.title('Best cost found in each iteration of EDA')
plt.plot(list(range(len(eda_result.history))), eda_result.history, color='b')
plt.xlabel('iteration')
plt.ylabel('MAE')
plt.show()
```

2.5 Using EMNA for continuous optimization

In this notebook we use the EMNA approach to optimize a wellknown benchmark. Note that EMNA learns and sampled a multivariate Gaussian in each iteration. Import the algorithm and the benchmarks from EDAspy.

```
from EDAspy.optimization import EMNA
from EDAspy.benchmarks import ContinuousBenchmarkingCEC14
```

We will be using a benchmark with 10 variables.

```
n_vars = 10
benchmarking = ContinuousBenchmarkingCEC14(n_vars)
```

We initialize the EDA with the following parameters:

```
emna = EMNA(size_gen=300, max_iter=100, dead_iter=20, n_variables=10,
            landscape_bounds=(-60, 60))

eda_result = emna.minimize(benchmarking.cec14_4, True)
```

We plot the best cost found in each iteration of the algorithm.

```
plt.figure(figsize = (14,6))

plt.title('Best cost found in each iteration of EDA')
plt.plot(list(range(len(eda_result.history))), eda_result.history, color='b')
plt.xlabel('iteration')
plt.ylabel('MAE')
plt.show()
```

2.6 Using EDAs for time series and times series transformation selection

When working with Time series in a Machine Learning project it is very common to try different combinations of the time series in order to perform better the forecasting model. In this section we will apply an EDA to select the optimal subset of variables and time series transformations to improve the model.

```
# loading essential libraries first
import pandas as pd
import statsmodels.api as sm
from statsmodels.tsa.api import VAR
import matplotlib.pyplot as plt
from sklearn.metrics import mean_absolute_error

# EDAspy libraries
from EDAspy.timeseries import EDA_ts_fts as EDA
from EDAspy.timeseries import TSTransformations
```

```
# import some data
mdata = sm.datasets.macrodta.load_pandas().data
df = mdata.iloc[:, 2:12]
df.head()
```

```
variables = list(df.columns)
variable_y = 'pop' # pop is the variable we want to forecast
variables = list(set(variables) - {variable_y}) # array of variables to select among
↳ transformations
variables
```

We define a cost function which receives a dictionary **with** variables names **as** keys of
 ↳ the dictionary **and**
 values **1/0** **if** they are used **or not** respectively.

```
TSTransf = TSTransformations(df)
transformations = ['detrend', 'smooth', 'log'] # postfix to variables, to denote the
↳ transformation

# build the transformations
for var in variables:
    transformation = TSTransf.de_trending(var)
    df[var + 'detrend'] = transformation
```

(continues on next page)

(continued from previous page)

```

for var in variables:
    transformation = TSTransf.smoothing(var, window=10)
    df[var + 'smooth'] = transformation

for var in variables:
    transformation = TSTransf.log(var)
    df[var + 'log'] = transformation

```

Define the cost function to calculate the Mean Absolute Error

```

def cost_function(variables_list, nobs=20, maxlags=15, forecastings=10):
    """
    variables_list: list of variables without the variable_y
    nobs: how many observations for validation
    maxlags: previous lags used to predict
    forecasting: number of observations to predict

    return: MAE of the prediction with the real validation data
    """

    data = df[variables_list + [variable_y]]

    df_train, df_test = data[0:-nobs], data[-nobs:]

    model = VAR(df_train)
    results = model.fit(maxlags=maxlags, ic='aic')

    lag_order = results.k_ar
    array = results.forecast(df_train.values[-lag_order:], forecastings)

    variables_ = list(data.columns)
    position = variables_.index(variable_y)

    validation = [array[i][position] for i in range(len(array))]
    mae = mean_absolute_error(validation, df_test['pop'][-forecastings:])

    return mae

```

We take the normal variables without any time series transformation and try to forecast the y variable using the same cost function defined. This value is stored to be compared with the optimum solution found

```

eda = UMDAd(size_gen=30, max_iter=100, dead_iter=10, n_variables=len(variables), alpha=0.
↪ 5, vector=None,
    lower_bound=0.2, upper_bound=0.9, elite_factor=0.2, disp=True)

eda_result = eda.minimize(cost_function=cost_function, output_runtime=True)

```

Note that the algorithm is minimizing correctly, but due to the fact that it is a toy example, there is not a high variance from the beginning to the end.

```

mae_pre_eda = cost_function(variables)
print('MAE without using EDA:', mae_pre_eda)

```

Initialization of the initial vector of statistics. Each variable has a 50% probability to be or not chosen

```
vector = pd.DataFrame(columns=list(variables))
vector.loc[0] = 0.5
```

Run the algorithm. The code will print some further information during execution

```
eda = EDA(max_it=50, dead_it=5, size_gen=15, alpha=0.7, vector=vector,
          array_transformations=transformations, cost_function=cost_function)
best_ind, best_MAE = eda.run(output=True)
```

We show some plots of the best solutions found during the execution in each iteration of the algorithm.

```
# some plots
hist = eda.historic_best

relative_plot = []
mx = 999999999
for i in range(len(hist)):
    if hist[i] < mx:
        mx = hist[i]
        relative_plot.append(mx)
    else:
        relative_plot.append(mx)

print('Solution:', best_ind, '\nMAE post EDA: %.2f' % best_MAE, '\nMAE pre EDA: %.2f' %
      ↪mae_pre_eda)

plt.figure(figsize = (14,6))

ax = plt.subplot(121)
ax.plot(list(range(len(hist))), hist)
ax.title.set_text('Local cost found')
ax.set_xlabel('iteration')
ax.set_ylabel('MAE')

ax = plt.subplot(122)
ax.plot(list(range(len(relative_plot))), relative_plot)
ax.title.set_text('Best global cost found')
ax.set_xlabel('iteration')
ax.set_ylabel('MAE')

plt.show()
```

CHANGELOG

3.1 v1.0.0

- This version implies a change in the way of using the EDAs.
- All EDAs extend an abstract class so, all EDAs have the same outline and the same minimize function.
- The cost function is now used only for the minimize function, so it is easier to be used.
- The probabilistic models and initialization models are treated separately from the EDA implementations so the user is able to decide whether to use a probabilistic model or other in the EDAs custom implementation.
- The user is able to export and read the configuration of an EDA in order to re-use the same implementation in the future.
- All the EDA implementations have their own name according to the state-of-the-art of EDAs.
- More tests have been added.
- Documentation has been redone.
- Deprecation warning to TimeSeries selector. This class will be formatted in following versions.
- The structure in the package has been removed and also the names.
- The implementation of EGAN with evidences has been removed to avoid having rpy2 as a dependency.

3.2 v0.2.0

- Time series transformations selection was added as a new functionality of the package.
- Added a notebooks section to show some real use cases of EDAspy. (3 implementations)

3.3 v0.1.2

- Added tests

3.4 v0.1.1

- Fixed bugs.
- Added documentation to readdocs.

3.5 v0.1.0

- First operative version 4 EDAs implemented.
- univariate EDA discrete.
- Univariate EDA continuous.
- Multivariate continuous EDA with evidences
- Multivariate continuous EDA with no evidences gaussian distribution.

GETTING STARTED

For installing EDAspy from Pypi execute the following command using pip:

```
pip install EDAspy
```

4.1 Build from Source

4.1.1 Prerequisites

- Python 3.6, 3.7, 3.8 or 3.9.
- Pybnesian, numpy, pandas.

4.1.2 Building

Clone the repository:

```
git clone https://github.com/VicentePerezSoloviev/EDAspy.git
cd EDAspy
git checkout v1.0.0 # You can checkout a specific version if you want
python setup.py install
```

4.2 Testing

The library contains tests that can be executed using `pytest`. Install it using pip:

```
pip install pytest
```

Run the tests with:

```
pytest
```


FORMAL DOCUMENTATION

5.1 EDAspy package

5.1.1 Subpackages

5.1.1.1 EDAspy.benchmarks package

Submodules

EDAspy.benchmarks.binary module

EDAspy.benchmarks.binary.**one_max**(array: Union[list, array]) → Union[float, int]

One max benchmark. :param array: solution to be evaluated in the cost function :return: evaluation of the solution

EDAspy.benchmarks.continuous module

class EDAspy.benchmarks.continuous.**ContinuousBenchmarkingCEC14**(dim: int)

Bases: object

ackley_function(x: Union[array, list]) → float

Ackley's Function

Parameters

x – solution to be evaluated

Returns

solution evaluation

Return type

float

bent_cigar_function(x: Union[array, list]) → float

Bent Cigar function

Parameters

x – solution to be evaluated

Returns

solution evaluation

Return type

float

cec14_1(x : *Union[array, list]*) \rightarrow float

Rotated High Conditioned Elliptic Function

Parameters \mathbf{x} – solution to be evaluated**Returns**

solution evaluation

Return type

float

cec14_10(x : *Union[array, list]*) \rightarrow float

Shifted Schwefel's Function

Parameters \mathbf{x} – solution to be evaluated**Returns**

solution evaluation

Return type

float

cec14_11(x : *Union[array, list]*) \rightarrow float

Shifted and Rotated Schwefel's Function

Parameters \mathbf{x} – solution to be evaluated**Returns**

solution evaluation

Return type

float

cec14_12(x : *Union[array, list]*) \rightarrow float

Shifted and Rotated Katsuura Function

Parameters \mathbf{x} – solution to be evaluated**Returns**

solution evaluation

Return type

float

cec14_13(x : *Union[array, list]*) \rightarrow float

Shifted and Rotated HappyCat Function

Parameters \mathbf{x} – solution to be evaluated**Returns**

solution evaluation

Return type

float

cec14_14(x : *Union[array, list]*) \rightarrow float

Shifted and Rotated HGBat Function

Parameters

\mathbf{x} – solution to be evaluated

Returns

solution evaluation

Return type

float

cec14_16(x : *Union[array, list]*) \rightarrow float

Shifted and Rotated Expanded Scaffer's F6 Function

Parameters

\mathbf{x} – solution to be evaluated

Returns

solution evaluation

Return type

float

cec14_2(x : *Union[array, list]*) \rightarrow float

Rotated Bent Cigar Function

Parameters

\mathbf{x} – solution to be evaluated

Returns

solution evaluation

Return type

float

cec14_3(x : *Union[array, list]*) \rightarrow float

Rotated Discus Function

Parameters

\mathbf{x} – solution to be evaluated

Returns

solution evaluation

Return type

float

cec14_4(x : *Union[array, list]*) \rightarrow float

Shifted and Rotated Rosenbrock's Function

Parameters

\mathbf{x} – solution to be evaluated

Returns

solution evaluation

Return type

float

cec14_5(x : *Union[array, list]*) \rightarrow float

Shifted and Rotated Rosenbrock's Function

Parameters

\mathbf{x} – solution to be evaluated

Returns

solution evaluation

Return type

float

cec14_6(x : *Union[array, list]*) \rightarrow float

Shifted and Rotated Weierstrass Function

Parameters

\mathbf{x} – solution to be evaluated

Returns

solution evaluation

Return type

float

cec14_7(x : *Union[array, list]*) \rightarrow float

Shifted and Rotated Griewank's Function

Parameters

\mathbf{x} – solution to be evaluated

Returns

solution evaluation

Return type

float

cec14_8(x : *Union[array, list]*) \rightarrow float

Shifted Rastrigin's Function

Parameters

\mathbf{x} – solution to be evaluated

Returns

solution evaluation

Return type

float

cec14_9(x : *Union[array, list]*) \rightarrow float

Shifted and Rotated Rastrigin's Function

Parameters

\mathbf{x} – solution to be evaluated

Returns

solution evaluation

Return type

float

discuss_function(x : *Union[array, list]*) \rightarrow float

Discuss function

Parameters

\mathbf{x} – solution to be evaluated

Returns

solution evaluation

Return type

float

expanded_scaffer_f6_function(*x: Union[array, list]*)

Expanded Scaffer's F6 Function

Parameters**x** – solution to be evaluated**Returns**

solution evaluation

Return type

float

griewank_function(*x: Union[array, list]*) → float

Griewank's Function

Parameters**x** – solution to be evaluated**Returns**

solution evaluation

Return type

float

happycat_function(*x: Union[array, list]*) → float

HappyCat Function

Parameters**x** – solution to be evaluated**Returns**

solution evaluation

Return type

float

hgbat_function(*x: Union[array, list]*) → float

HGBat Function

Parameters**x** – solution to be evaluated**Returns**

solution evaluation

Return type

float

high_conditioned_elliptic_function(*x: Union[array, list]*) → float**katsuura_function**(*x: Union[array, list]*) → float

Katsuura Function

Parameters**x** – solution to be evaluated

Returns

solution evaluation

Return type

float

mod_schwefels_function(*x*: *Union[array, list]*) → float

Modified Schwefel's Function

Parameters**x** – solution to be evaluated**Returns**

solution evaluation

Return type

float

rastrigins_function(*x*: *Union[array, list]*) → float

Rastrigin's Function

Parameters**x** – solution to be evaluated**Returns**

solution evaluation

Return type

float

rosenbrock_function(*x*: *Union[array, list]*) → float

Rosenbrock's Function

Parameters**x** – solution to be evaluated**Returns**

solution evaluation

Return type

float

weierstrass_function(*x*: *Union[array, list]*) → float

Weierstrass Function

Parameters**x** – solution to be evaluated**Returns**

solution evaluation

Return type

float

Module contents

5.1.1.2 EDAspy.optimization package

Subpackages

EDAspy.optimization.custom package

Subpackages

EDAspy.optimization.custom.initialization_models package

Submodules

EDAspy.optimization.custom.initialization_models.multi_gauss_gininit module

```

class EDAspy.optimization.custom.initialization_models.multi_gauss_gininit.MultiGaussGenInit(n_variables:
    int,
    means_vector:
    array
    =
    array([],
    dtype=float64,
    cov_matrix:
    array
    =
    array([],
    dtype=float64,
    lower_bound:
    float
    =
    -
    100,
    upper_bound:
    float
    =
    100)

```

Bases: GenInit

Initial generation simulator based on the probabilistic model of multivariate Gaussian distribution.

sample(size: int) → array

Sample several times the initializer.

Parameters

size – number of samplings.

Returns

array with the dataset sampled.

Return type

np.array

EDAspy.optimization.custom.initialization_models.uni_bin_geninit module

```
class EDAspy.optimization.custom.initialization_models.uni_bin_geninit.UniBinGenInit(n_variables:  
                                          int,  
                                          means_vector:  
                                          array  
                                          = ar-  
                                          ray([],  
                                          dtype=float64))
```

Bases: GenInit

Initial generation simulator based on the probabilistic model of univariate binary probabilities.

sample(*size: int*) → array

Sample several times the initializer.

Parameters

size – number of samplings.

Returns

array with the dataset sampled

Return type

np.array

EDAspy.optimization.custom.initialization_models.uni_gauss_geninit module

```

class EDAspy.optimization.custom.initialization_models.uni_gauss_geninit.UniGaussGenInit(n_variables:
                                                    int,
                                                    means_vector:
                                                    ar-
                                                    ray
                                                    =
                                                    ar-
                                                    ray([],
                                                    dtype=float64),
                                                    stds_vector:
                                                    ar-
                                                    ray
                                                    =
                                                    ar-
                                                    ray([],
                                                    dtype=float64),
                                                    lower_bound:
                                                    int
                                                    =
                                                    -
                                                    100,
                                                    higher_bound:
                                                    int
                                                    =
                                                    100)

```

Bases: GenInit

Initial generation simulator based on the probabilistic model of univariate binary probabilities.

sample(size) → array

Sample several times the initializer.

Parameters

size – number of samplings.

Returns

array with the dataset sampled.

Return type

np.array.

EDAspy.optimization.custom.initialization_models.uniform_geninit module

```

class EDAspy.optimization.custom.initialization_models.uniform_geninit.UniformGenInit(n_variables:
                                                    int,
                                                    lower_bound:
                                                    float
                                                    = -
                                                    100,
                                                    up-
                                                    per_bound:
                                                    float
                                                    =
                                                    100)

```

Bases: GenInit

Initial generation simulator based on independent uniform distributions.

sample(*size: int*) → array

Sample several times the initializer.

Parameters

size – number of samplings.

Returns

array with the dataset sampled.

Return type

np.array.

Module contents

EDAspy.optimization.custom.probabilistic_models package

Submodules

EDAspy.optimization.custom.probabilistic_models.gaussian_bayesian_network module

class EDAspy.optimization.custom.probabilistic_models.gaussian_bayesian_network.**GBN**(*variables: list*)

Bases: ProbabilisticModel

This probabilistic model is Gaussian Bayesian Network. All the relationships between the variables in the model are defined to be linearly Gaussian, and the variables distributions are assumed to be Gaussian. This is a very common approach when facing to continuous data as it is relatively easy and fast to learn a Gaussian distributions between variables. This implementation uses Pybnesian library [1].

References

[1]: Atienza, D., Bielza, C., & Larrañaga, P. (2022). PyBNesian: an extensible Python package for Bayesian networks. *Neurocomputing*, 504, 204-209.

learn(*dataset: array*)

Learn a Gaussian Bayesian network from the dataset passed as argument.

Parameters

dataset – dataset from which learn the GBN.

print_structure() → list

Prints the arcs between the nodes that represent the variables in the dataset. This function must be used after the learning process.

Returns

list of arcs between variables

Return type

list

sample(*size: int*) → array

Samples the Gaussian Bayesian network several times defined by the user. The dataset is returned as a numpy matrix. The sampling process is implemented using probabilistic logic sampling.

Parameters

size – number of samplings of the Gaussian Bayesian network.

Returns

array with the dataset sampled.

Return type

np.array

EDAspy.optimization.custom.probabilistic_models.multivariate_gaussian module

```
class EDAspy.optimization.custom.probabilistic_models.multivariate_gaussian.MultiGauss(variables:
                                                                                          list,
                                                                                          lower_bound:
                                                                                          float,
                                                                                          up-
                                                                                          per_bound:
                                                                                          float)
```

Bases: ProbabilisticModel

This class implements all the code needed to learn and sample multivariate Gaussian distributions defined by a vector of means and a covariance matrix among the variables. This is a simpler approach compared to Gaussian Bayesian networks, as multivariate Gaussian distributions do not identify conditional dependences between the variables.

learn(dataset: array)

Estimates a multivariate Gaussian distribution from the dataset.

Parameters

dataset – dataset from which learn the multivariate Gaussian distribution.

sample(size: int) → array

Samples the multivariate Gaussian distribution several times defined by the user. The dataset is returned as a numpy matrix.

Parameters

size – number of samplings of the Gaussian Bayesian network.

Returns

array with the dataset sampled.

Return type

np.array

EDAspy.optimization.custom.probabilistic_models.univariate_binary module

```
class EDAspy.optimization.custom.probabilistic_models.univariate_binary.UniBin(variables:
                                                                                   list, up-
                                                                                   per_bound:
                                                                                   float,
                                                                                   lower_bound:
                                                                                   float)
```

Bases: ProbabilisticModel

This is the simplest probabilistic model implemented in this package. This is used for binary EDAs where all the solutions are binary. The implementation involves a vector of independent probabilities [0, 1]. When sampling, a

random float is sampled [0, 1]. If the float is below the probability, then the sampling is a 1. Thus, the probabilities show probabilities of a sampling being 1.

learn(*dataset: array*)

Estimates the independent probability of each variable of being 1.

Parameters

dataset – dataset from which learn the probabilistic model.

sample(*size: int*) → array

Samples new solutions from the probabilistic model. In each solution, each variable is sampled from its respective binary probability.

Parameters

size – number of samplings of the probabilistic model.

Returns

array with the dataset sampled.

Return type

np.array

EDAspy.optimization.custom.probabilistic_models.univariate_gaussian module

```
class EDAspy.optimization.custom.probabilistic_models.univariate_gaussian.UniGauss(variables:  
                                          list,  
                                          lower_bound:  
                                          float)
```

Bases: ProbabilisticModel

This class implements the univariate Gaussians. With this implementation we are updating N univariate Gaussians in each iteration. When a dataset is given, each column is updated independently. The implementation involves a matrix with two rows, in which the first row are the means and the second one, are the standard deviations.

learn(*dataset: array*)

Estimates the independent Gaussian for each variable.

Parameters

dataset – dataset from which learn the probabilistic model.

sample(*size: int*) → array

Samples new solutions from the probabilistic model. In each solution, each variable is sampled from its respective normal distribution.

Parameters

size – number of samplings of the probabilistic model.

Returns

array with the dataset sampled

Return type

np.array

Module contents

Submodules

EDAspy.optimization.custom.eda_custom module

class EDAspy.optimization.custom.eda_custom.EDACustom(*size_gen: int, max_iter: int, dead_iter: int, n_variables: int, alpha: float, elite_factor: float, disp: bool, pm: int, init: int, bounds: tuple*)

Bases: [EDA](#)

This class allows the user to define an EDA by custom. This implementation is thought to be extended and extend the methods to allow different implementations. Moreover, the probabilistic models and initializations can be combined to invent or design a custom EDA.

The class allows the user to export and load the settings of previous EDA configurations, so this favours the implementation of auto-tuning approaches, for example.

Example

This example uses some very well-known benchmarks from CEC14 conference to be solved using a custom implementation of EDAs.

```
from EDAspy.optimization.custom import EDACustom, GBN, UniformGenInit
from EDAspy.benchmarks import ContinuousBenchmarkingCEC14

n_variables = 10
my_eda = EDACustom(size_gen=100, max_iter=100, dead_iter=n_variables, n_variables=n_variables,
    alpha=0.5, elite_factor=0.2, disp=True, pm=4, init=4, bounds=(-50, 50))

benchmarking = ContinuousBenchmarkingCEC14(n_variables)

my_gbn = GBN([str(i) for i in range(n_variables)])
my_init = UniformGenInit(n_variables)

my_eda.pm = my_gbn
my_eda.init = my_init

eda_result = my_eda.minimize(cost_function=benchmarking.cec14_4)
```

export_settings() → dict

Export the settings of the EDA. :return: dictionary with the configuration. :rtype dict

EDAspy.optimization.custom.eda_custom.**read_settings**(*settings: dict*) → [EDACustom](#)

This function is implemented to automatic implement the EDA custom by importing the configuration of a previous implementation. The function accepts the configuration exported from a previous EDA.

Parameters

settings (*dict*) – dictionary with the previous configuration.

Returns

EDA custom automatic built.

Return type*EDACustom***Module contents****EDAspy.optimization.multivariate package****Submodules****EDAspy.optimization.multivariate.egna module**

```
class EDAspy.optimization.multivariate.egna.EGNA(size_gen: int, max_iter: int, dead_iter: int,  
                                                  n_variables: int, landscape_bounds: tuple, alpha:  
                                                  float = 0.5, elite_factor: float = 0.4, disp: bool =  
                                                  True)
```

Bases: *EDA*

Estimation of Gaussian Networks Algorithm. This type of Estimation-of-Distribution Algorithm uses a Gaussian Bayesian Network from where new solutions are sampled. This multivariate probabilistic model is updated in each iteration with the best individuals of the previous generation.

EGNA [1] has shown to improve the results for more complex optimization problem compared to the univariate EDAs that can be found implemented in this package. Different modifications have been done into this algorithm such as in [2] where some evidences are input to the Gaussian Bayesian Network in order to restrict the search space in the landscape.

Example

This example uses some very well-known benchmarks from CEC14 conference to be solved using an Estimation of Gaussian Networks Algorithm (EGNA).

```
from EDAspy.optimization import EGNA  
from EDAspy.benchmarks import ContinuousBenchmarkingCEC14  
  
benchmarking = ContinuousBenchmarkingCEC14(10)  
  
egna = EGNA(size_gen=300, max_iter=100, dead_iter=20, n_variables=10,  
            landscape_bounds=(-60, 60))  
  
eda_result = egna.minimize(benchmarking.cec14_4, True)
```

References

[1]: Larrañaga, P., & Lozano, J. A. (Eds.). (2001). Estimation of distribution algorithms: A new tool for evolutionary computation (Vol. 2). Springer Science & Business Media.

[2]: Vicente P. Soloviev, Pedro Larrañaga and Concha Bielza (2022). Estimation of distribution algorithms using Gaussian Bayesian networks to solve industrial optimization problems constrained by environment variables. Journal of Combinatorial Optimization.

EDAspy.optimization.multivariate.emna module

```
class EDAspy.optimization.multivariate.emna.EMNA(size_gen: int, max_iter: int, dead_iter: int,
          n_variables: int, landscape_bounds: tuple, alpha:
          float = 0.5, elite_factor: float = 0.4, disp: bool =
          True, lower_bound: float = 0.5, upper_bound: float
          = 100)
```

Bases: [EDA](#)

Estimation of Multivariate Normal Algorithm (EMNA) [1] is a multivariate continuous EDA in which no probabilistic graphical models are used during runtime. In each iteration the new solutions are sampled from a multivariate normal distribution built from the elite selection of the previous generation.

In this implementation, as in EGNA, the algorithm is initialized from a uniform sampling in the landscape bound you input in the constructor of the algorithm. If a different initialization_models is desired, then you can override the class and this specific method.

This algorithm is widely used in the literature and compared for different optimization tasks with its competitors in the EDAs multivariate continuous research topic.

Example

This example uses some very well-known benchmarks from CEC14 conference to be solved using an Estimation of Multivariate Normal Algorithm (EMNA).

```
from EDAspy.optimization import EMNA
from EDAspy.benchmarks import ContinuousBenchmarkingCEC14

benchmarking = ContinuousBenchmarkingCEC14(10)

emna = EMNA(size_gen=300, max_iter=100, dead_iter=20, n_variables=10,
            landscape_bounds=(-60, 60), std_bound=5)

eda_result = emna.minimize(cost_function=benchmarking.cec14_4)
```

References

[1]: Larrañaga, P., & Lozano, J. A. (Eds.). (2001). Estimation of distribution algorithms: A new tool for evolutionary computation (Vol. 2). Springer Science & Business Media.

Module contents

EDAspy.optimization.univariate package

Submodules

EDAspy.optimization.univariate.umd_binary module

```
class EDAspy.optimization.univariate.umdabinary.UMDAd(size_gen: int, max_iter: int, dead_iter: int,
                                                       n_variables: int, alpha: float = 0.5, vector:
                                                       Optional[array] = None, lower_bound: float
                                                       = 0.2, upper_bound: float = 0.8,
                                                       elite_factor: float = 0.4, disp: bool = True)
```

Bases: [EDA](#)

Univariate marginal Estimation of Distribution algorithm binary. New individuals are sampled from a univariate binary probabilistic model. It can be used for hyper-parameter optimization or to optimize a function.

UMDA [1] is a specific type of Estimation of Distribution Algorithm (EDA) where new individuals are sampled from univariate binary distributions and are updated in each iteration of the algorithm by the best individuals found in the previous iteration. In this implementation each individual is an array of 0s and 1s so new individuals are sampled from a univariate probabilistic model updated in each iteration. Optionally it is possible to set lower and upper bound to the probabilities to avoid premature convergence.

This approach has been widely used and shown to achieve very good results in a wide range of problems such as Feature Subset Selection or Portfolio Optimization.

Example

This short example runs UMDAd for a toy example of the One-Max problem.

```
from EDAspy.benchmarks import one_max
from EDAspy.optimization import UMDAc, UMDAd

def one_max_min(array):
    return -one_max(array)

umda = UMDAd(size_gen=100, max_iter=100, dead_iter=10, n_variables=10)
# We leave bound by default
eda_result = umda.minimize(one_max_min, True)
```

References

[1]: Mühlenbein, H., & Paass, G. (1996, September). From recombination of genes to the estimation of distributions I. Binary parameters. In International conference on parallel problem solving from nature (pp. 178-187). Springer, Berlin, Heidelberg.

EDAspy.optimization.univariate.umdac_continuous module

```
class EDAspy.optimization.univariate.umdac_continuous.UMDAd(size_gen: int, max_iter: int, dead_iter:
                                                             int, n_variables: int, alpha: float = 0.5,
                                                             vector: Optional[array] = None,
                                                             lower_bound: float = 0.5, elite_factor:
                                                             float = 0.4, disp: bool = True)
```

Bases: [EDA](#)

Univariate marginal Estimation of Distribution algorithm continuous. New individuals are sampled from a univariate normal probabilistic model. It can be used for hyper-parameter optimization or to optimize a function.

UMDA [1] is a specific type of Estimation of Distribution Algorithm (EDA) where new individuals are sampled from univariate normal distributions and are updated in each iteration of the algorithm by the best individuals

found in the previous iteration. In this implementation each individual is an array of 0s and 1s so new individuals are sampled from a univariate probabilistic model updated in each iteration. Optionally it is possible to set lower bound to the standard deviation of the normal distribution for the variables to avoid premature convergence.

This algorithm has been widely used for different applications such as in [2] where it is applied to optimize the parameters of a quantum parametric circuit and is shown how it outperforms other approaches in specific situations.

Example

This short example runs UMDAc for a toy example of the One-Max problem in the continuous space.

```
from EDAspy.benchmarks import ContinuousBenchmarkingCEC14
from EDAspy.optimization import UMDAc

n_vars = 10
benchmarking = ContinuousBenchmarkingCEC14(n_vars)

umda = UMDAc(size_gen=100, max_iter=100, dead_iter=10, n_variables=10, alpha=0.5)
# We leave bound by default
eda_result = umda.minimize(benchmarking.cec4, True)
```

References

- [1]: Larrañaga, P., & Lozano, J. A. (Eds.). (2001). Estimation of distribution algorithms: A new tool for evolutionary computation (Vol. 2). Springer Science & Business Media.
- [2]: Vicente P. Soloviev, Pedro Larrañaga and Concha Bielza (2022, July). Quantum Parametric Circuit Optimization with Estimation of Distribution Algorithms. In 2022 The Genetic and Evolutionary Computation Conference (GECCO). DOI: <https://doi.org/10.1145/3520304.3533963>

Module contents

Submodules

EDAspy.optimization.eda module

```
class EDAspy.optimization.eda.EDA(size_gen: int, max_iter: int, dead_iter: int, n_variables: int, alpha: float
    = 0.5, elite_factor: float = 0.4, disp: bool = True)
```

Bases: ABC

Abstract class which defines the general performance of the algorithms. The baseline of the EDA approach is defined in this object. The specific configurations is defined in the class of each specific algorithm.

export_settings() → dict

Export the configuration of the algorithm to an object to be loaded in other execution. :return: dict

property init

minimize(cost_function: callable, output_runtime: bool = True)

Parameters

- **cost_function** – cost function to be optimized and accepts an array as argument.

- **output_runtime** – true if information during runtime is desired.

Returns

EdaResult object

Return type

EdaResult

property **pm**

EDAspy.optimization.eda_result module

class EDAspy.optimization.eda_result.**EdaResult**(*best_ind: array, best_cost: float, n_fev: int, history: list, settings: dict*)

Bases: object

Object used to encapsulate the result and information of the EDA during the execution

Module contents

5.1.1.3 EDAspy.timeseries package

Submodules

EDAspy.timeseries.TS_transformations module

EDAspy.timeseries.TransformationsFeatureSelection module

Module contents

5.1.2 Module contents

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

e

- EDAspy, 32
- EDAspy.benchmarks, 21
- EDAspy.benchmarks.binary, 15
- EDAspy.benchmarks.continuous, 15
- EDAspy.optimization, 32
- EDAspy.optimization.custom, 28
- EDAspy.optimization.custom.eda_custom, 27
- EDAspy.optimization.custom.initialization_models,
24
- EDAspy.optimization.custom.initialization_models.multi_gauss_geninit,
21
- EDAspy.optimization.custom.initialization_models.uni_bin_geninit,
22
- EDAspy.optimization.custom.initialization_models.uni_gauss_geninit,
22
- EDAspy.optimization.custom.initialization_models.uniform_geninit,
23
- EDAspy.optimization.custom.probabilistic_models,
27
- EDAspy.optimization.custom.probabilistic_models.gaussian_bayesian_network,
24
- EDAspy.optimization.custom.probabilistic_models.multivariate_gaussian,
25
- EDAspy.optimization.custom.probabilistic_models.univariate_binary,
25
- EDAspy.optimization.custom.probabilistic_models.univariate_gaussian,
26
- EDAspy.optimization.eda, 31
- EDAspy.optimization.eda_result, 32
- EDAspy.optimization.multivariate, 29
- EDAspy.optimization.multivariate.egna, 28
- EDAspy.optimization.multivariate.emna, 29
- EDAspy.optimization.univariate, 31
- EDAspy.optimization.univariate.umda_binary,
29
- EDAspy.optimization.univariate.umda_continuous,
30

INDEX

A

`ackley_function()` (*EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14* method), 15

B

`bent_cigar_function()` (*EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14* method), 15

C

`cec14_1()` (*EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14* method), 16

`cec14_10()` (*EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14* method), 16

`cec14_11()` (*EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14* method), 16

`cec14_12()` (*EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14* method), 16

`cec14_13()` (*EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14* method), 16

`cec14_14()` (*EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14* method), 16

`cec14_16()` (*EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14* method), 17

`cec14_2()` (*EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14* method), 17

`cec14_3()` (*EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14* method), 17

`cec14_4()` (*EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14* method), 17

`cec14_5()` (*EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14* method), 17

`cec14_6()` (*EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14* method), 18

`cec14_7()` (*EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14* method), 18

`cec14_8()` (*EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14* method), 18

`cec14_9()` (*EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14* method), 18

`ContinuousBenchmarkingCEC14` (class in *EDAspy.benchmarks.continuous*), 15

D

`discuss_benchmarking()` (*EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14* method), 18

E

`EDA` (class in *EDAspy.optimization.eda*), 31

`EDA_custom()` (class in *EDAspy.optimization.custom.eda_custom*), 27

`EdaResult` (class in *EDAspy.optimization.eda_result*), 32

`EDAspy`

- `EDAspy.benchmarks` module, 21
- `EDAspy.benchmarks.binary` module, 15
- `EDAspy.benchmarks.continuous` module, 15
- `EDAspy.optimization` module, 32
- `EDAspy.optimization.custom` module, 28
- `EDAspy.optimization.custom.eda_custom` module, 27
- `EDAspy.optimization.custom.initialization_models` module, 24
- `EDAspy.optimization.custom.initialization_models.multi_gauss` module, 21
- `EDAspy.optimization.custom.initialization_models.uni_bin_g` module, 22
- `EDAspy.optimization.custom.initialization_models.uni_gauss` module, 22
- `EDAspy.optimization.custom.initialization_models.uniform_g` module, 23
- `EDAspy.optimization.custom.probablistic_models` module, 27
- `EDAspy.optimization.custom.probablistic_models.gaussian_b` module, 24
- `EDAspy.optimization.custom.probablistic_models.multivaria` module, 25
- `EDAspy.optimization.custom.probablistic_models.univariate` module, 25

EDAspy.optimization.custom.probabilistic_models.univariate_gaussian
 module, 26
 EDAspy.optimization.eda
 module, 31
 EDAspy.optimization.eda_result
 module, 32
 EDAspy.optimization.multivariate
 module, 29
 EDAspy.optimization.multivariate.egna
 module, 28
 EDAspy.optimization.multivariate.emna
 module, 29
 EDAspy.optimization.univariate
 module, 31
 EDAspy.optimization.univariate.umda_binary
 module, 29
 EDAspy.optimization.univariate.umda_continuous
 module, 30
 EGNA (class in EDAspy.optimization.multivariate.egna),
 28
 EMNA (class in EDAspy.optimization.multivariate.emna),
 29
 expanded_scaffer_f6_function()
 (EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14
 method), 19
 export_settings() (EDAspy.optimization.custom.eda_custom.EDACustom
 method), 27
 export_settings() (EDAspy.optimization.eda.EDA
 method), 31
G
 GBN (class in EDAspy.optimization.custom.probabilistic_models.gaussian_bayesian_network),
 24
 griewank_function()
 (EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14
 method), 19
H
 happycat_function()
 (EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14
 method), 19
 hgbat_function() (EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14
 method), 19
 high_conditioned_elliptic_function()
 (EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14
 method), 19
I
 init (EDAspy.optimization.eda.EDA property), 31
K
 katsuura_function()
 (EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14
 method), 19

learn() (EDAspy.optimization.custom.probabilistic_models.gaussian_bayesian_network.
 method), 24
 learn() (EDAspy.optimization.custom.probabilistic_models.multivariate_gaussian.
 method), 25
 learn() (EDAspy.optimization.custom.probabilistic_models.univariate_bivariate.
 method), 26
 learn() (EDAspy.optimization.custom.probabilistic_models.univariate_gaussian.
 method), 26

M

minimize() (EDAspy.optimization.eda.EDA method), 31
 mod_schwefels_function()
 (EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14
 method), 20

module

EDAspy, 32
 EDAspy.benchmarks, 21
 EDAspy.benchmarks.binary, 15
 EDAspy.benchmarks.continuous, 15
 EDAspy.optimization, 32
 EDAspy.optimization.custom, 28
 EDAspy.optimization.custom.eda_custom, 27
 EDAspy.optimization.custom.initialization_models,
 24
 EDAspy.optimization.custom.initialization_models.multivariate,
 21
 EDAspy.optimization.custom.initialization_models.univariate,
 22
 EDAspy.optimization.custom.initialization_models.univariate_gaussian,
 22
 EDAspy.optimization.custom.initialization_models.univariate_multivariate,
 23
 EDAspy.optimization.custom.probabilistic_models,
 27
 EDAspy.optimization.custom.probabilistic_models.gaussian_bayesian_network,
 24
 EDAspy.optimization.custom.probabilistic_models.multivariate_gaussian,
 25
 EDAspy.optimization.custom.probabilistic_models.univariate_bivariate,
 25
 EDAspy.optimization.custom.probabilistic_models.univariate_gaussian,
 26
 EDAspy.optimization.eda, 31
 EDAspy.optimization.eda_result, 32
 EDAspy.optimization.multivariate, 29
 EDAspy.optimization.multivariate.egna, 28
 EDAspy.optimization.multivariate.emna, 29
 EDAspy.optimization.univariate, 31
 EDAspy.optimization.univariate.umda_binary,
 29
 EDAspy.optimization.univariate.umda_continuous,

MultiGauss (class in *EDAspy.optimization.custom.probabilitic_models.multivariate_gaussian*), 25
MultiGaussGenInit (class in *EDAspy.optimization.custom.initialization_models.uniform_geninit*), 23
MultiGauss (class in *EDAspy.optimization.custom.probabilitic_models.univariate_gaussian*), 26
UniGaussGenInit (class in *EDAspy.optimization.custom.initialization_models.uni_gauss_geninit*), 22
one_max() (in module *EDAspy.benchmarks.binary*), 15
P
pm (*EDAspy.optimization.eda.EDA* property), 32
print_structure() (*EDAspy.optimization.custom.probabilitic_models.gaussian_bayesian_network.GBN* method), 24
R
rastrigins_function() (*EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14* method), 20
read_settings() (in module *EDAspy.optimization.custom.eda_custom*), 27
rosenbrock_function() (*EDAspy.benchmarks.continuous.ContinuousBenchmarkingCEC14* method), 20
S
sample() (*EDAspy.optimization.custom.initialization_models.multi_gauss_geninit.MultiGaussGenInit* method), 21
sample() (*EDAspy.optimization.custom.initialization_models.uni_bin_geninit.UniBinGenInit* method), 22
sample() (*EDAspy.optimization.custom.initialization_models.uni_gauss_geninit.UniGaussGenInit* method), 23
sample() (*EDAspy.optimization.custom.initialization_models.uniform_geninit.UniformGenInit* method), 24
sample() (*EDAspy.optimization.custom.probabilitic_models.gaussian_bayesian_network.GBN* method), 24
sample() (*EDAspy.optimization.custom.probabilitic_models.multivariate_gaussian.MultiGauss* method), 25
sample() (*EDAspy.optimization.custom.probabilitic_models.univariate_binary.UniBin* method), 26
sample() (*EDAspy.optimization.custom.probabilitic_models.univariate_gaussian.UniGauss* method), 26
U
UMDAc (class in *EDAspy.optimization.univariate.umda_continuous*), 30
UMDAc (class in *EDAspy.optimization.univariate.umda_binary*), 29
UniBin (class in *EDAspy.optimization.custom.probabilitic_models.univariate_binary*), 25
UniBinGenInit (class in *EDAspy.optimization.custom.initialization_models.uni_bin_geninit*), 22