EDAspy

Release 1.1.0

Vicente P. Soloviev

Jun 28, 2023

CONTENTS:

CHAPTER

ONE

EDASPY

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 initializators, 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:

- UMDAd: 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 de 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 exampled.

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
```

CHAPTER

TWO

EXAMPLES

Some toy 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.macrodata.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, nobs=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.
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
"""

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 weith 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 minimzing correctly, but doe 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 parameteres, we are goind 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 initializator 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 SPEDA for continuous optimization

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

```
from EDAspy.optimization import SPEDA
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:

```
speda = SPEDA(size_gen=300, max_iter=100, dead_iter=20, n_variables=10,
              landscape_bounds=(-60, 60), l=10)

eda_result = speda.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()
```

Let's visualize the BN structure learnt in the last iteration of the algorithm.

```
from EDAspy.optimization import plot_bn

plot_bn(speda.pm.print_structure(), n_variables=n_vars)
```

2.5 Using SPEDA for continuous optimization

In this notebook we use the MultivariateKEDA approach to optimize a wellknown benchmark. Note that KEDA learns and samples a KDE estimated Bayesian network in each iteration. Import the algorithm and the benchmarks from EDAspy.

```
from EDAspy.optimization import MultivariateKEDA
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:

```
keeda = MultivariateKEDA(size_gen=300, max_iter=100, dead_iter=20, n_variables=10,
                        landscape_bounds=(-60, 60), l=10)

eda_result = keda.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('function cost')
plt.show()
```

Let's visualize the BN structure learnt in the last iteration of the algorithm.

```
from EDAspy.optimization import plot_bn

plot_bn(keda.pm.print_structure(), n_variables=n_vars)
```

2.6 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 = egna.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()
```

Let's visualize the BN structure learnt in the last iteration of the algorithm.

```
from EDAspy.optimization import plot_bn

plot_bn(egna.pm.print_structure(), n_variables=n_variables)
```

2.7 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()
```

2.8 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
```

(continues on next page)

(continued from previous page)

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

```
# import some data
mdata = sm.datasets.macrodata.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

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:]
```

(continues on next page)

(continued from previous page)

```
    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 minimzing correctly, but doe 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 statitstics. 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)
```

(continues on next page)

(continued from previous page)

```
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()
```

CHAPTER

THREE

CHANGELOG

3.1 v1.1.1

- This version implements the SPEDA algorithm to allow dependencies between variables that fit Gaussian distributions and KDE nodes.
- This version implements the multivariate version of KEDA, which shares all the characteristics with the SPEDA approach, with the exception that all the nodes have to be estimated with KDE. Gaussian nodes are forbidden.
- This version implements a function to plot the BN structure learnt in the EDA implementations.
- This version enforces the tests to avoid bugs in the algorithms.
- This version implements the possibility of settings white and black boxes to set the mandatory or forbidden arcs in the BN structure learnt in each iteration.
- This version solves several bugs present in v1.0.2.
- This version implements the parallelization for all the EDAs.
- This version allows initialize the algorithm from a custom set of samples.
- This version implements the multivariate and univariate KEDA algorithms, where variables are estimated using KDE.

3.2 v1.0.2

- This version solves a bug in the EGNA optimizer related to the Gaussian Bayesian network structure learning.

3.3 v1.0.1

- This version solves a bug in the UMDAd optimizer related to the limits of the std in each variable.

3.4 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 some 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.
- Th user is able to export and read the configuration of and 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.

3.5 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)

3.6 v0.1.2

- Added tests

3.7 v0.1.1

- Fixed bugs.
- Added documentation to readdocs.

3.8 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.

CHAPTER

FOUR

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
```

CHAPTER

FIVE

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`

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

Bent Cigar function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

Discuss function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

Rosenbrock's Function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

Ackley's Function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

Weierstrass Function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

Griewank's Function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

Rastrigin's Function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

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

Modified Schwefel's Function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

Katsuura Function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

HappyCat Function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

HGBat Function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

Expanded Scaffer's F6 Function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

Rotated High Conditioned Elliptic Function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

Rotated Bent Cigar Function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

Rotated Discus Function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

Shifted and Rotated Rosenbrock's Function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

Shifted and Rotated Rosenbrock's Function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

Shifted and Rotated Weierstrass Function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

Shifted and Rotated Griewank's Function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

Shifted Rastrigin's Function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

Shifted and Rotated Rastrigin's Function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

Shifted Schwefel's Function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

Shifted and Rotated Schwefel's Function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

Shifted and Rotated Katsuura Function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

Shifted and Rotated HappyCat Function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

Shifted and Rotated HGBat Function :param x: solution to be evaluated :return: solution evaluation :rtype: float

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

Shifted and Rotated Expanded Scaffer's F6 Function :param x: solution to be evaluated :return: solution evaluation :rtype: 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_geninit module

class `EDAspy.optimization.custom.initialization_models.multi_gauss_geninit.`**`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 initializator.

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 = array([], dtype=float64)*)

Bases: `GenInit`

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

sample(*size: int*) → array

Sample several times the initializator.

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: array = array([], dtype=float64)*, *stds_vector: array = array([], 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 initializator.

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*, *upper_bound: float = 100*)

Bases: `GenInit`

Initial generation simulator based on independent uniform distributions.

sample(*size: int*) → array

Sample several times the initializator.

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.semiparametric_bayesian_network module

class EDAspy.optimization.custom.probabilistic_models.semiparametric_bayesian_network.**SPBN**(*variables: list*, *white_list: Optional[list] = None*, *black_list: Optional[list] = None*)

Bases: `ProbabilisticModel`

This probabilistic model is a Semiparametric Bayesian network [1]. It allows dependencies between variables which have been estimated using KDE with variables which fit a Gaussian distribution.

References

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

`learn`(*dataset: array*, *num_folds: int = 10*, **args*, ***kwargs*)

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

Parameters

- `dataset` – dataset from which learn the SPBN.
- `num_folds` – Number of folds used for the SPBN learning. The higher, the more accurate, but also higher CPU demand. By default, it is set to 10.
- `max_iters` – number maximum of iterations for the learning process.

`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 Semiparametric 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 Semiparametric Bayesian network.

Returns
array with the dataset sampled.

Return type
np.array

`logl`(*data: DataFrame*)

Returns de log-likelihood of some data in the model.

Parameters
`data` – dataset to evaluate its likelihood in the model.

Returns
log-likelihood of the instances in the model.

Return type
np.array

EDAspy.optimization.custom.probabilistic_models.gaussian_bayesian_network module

class EDAspy.optimization.custom.probabilistic_models.gaussian_bayesian_network.**GBN**(*variables: list*, *white_list: Optional[list] = None*, *black_list: Optional[list] = None*, *evidences: Optional[dict] = None*)

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*, **args*, ***kwargs*)

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

logl(*data: DataFrame*)

Returns de log-likelihood of some data in the model.

Parameters
`data` – dataset to evaluate its likelihood in the model.

Returns
log-likelihood of the instances in the model.

Return type
np.array

`get_mu`(*var_mus=None*) → array

Computes the conditional mean of the Gaussians of each node in the GBN.

Parameters
var_mus (*list*) – Variables to compute its Gaussian mean. If None, then all the variables are computed.

Returns
Array with the conditional Gaussian means.

Return type
np.array

`get_sigma`(*var_sigma=None*) → array

Computes the conditional covariance matrix of the model for the variables in the GBN.

Parameters
var_sigma (*list*) – Variables to compute its Gaussian mean. If None, then all the variables are computed.

Returns
Matrix with the conditional covariance matrix.

Return type
np.array

`inference`(*evidence*, *var_names) -> (<built-in function array>*, *<built-in function array>*)

Compute the posterior conditional probability distribution conditioned to some given evidences. :param evidence: list of values fixed as evidences in the model. :type evidence: list :param var_names: list of variables measured in the model. :type var_names: list :return: (posterior mean, posterior covariance matrix) :rtype: (np.array, np.array)

EDAspy.optimization.custom.probabilistic_models.multivariate_gaussian module

class `EDAspy.optimization.custom.probabilistic_models.multivariate_gaussian.MultiGauss`(*variables: list*, *lower_bound: float*, *upper_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 dependeces between the variables.

`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

learn(*dataset: array*, **args*, ***kwargs*)

Estimates a multivariate Gaussian distribution from the dataset.

Parameters
`dataset` – dataset from which learn the multivariate Gaussian distribution.

EDAspy.optimization.custom.probabilistic_models.univariate_binary module

class `EDAspy.optimization.custom.probabilistic_models.univariate_binary.`**UniBin**(*variables: list*, *upper_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.

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

learn(*dataset: array*, **args*, ***kwargs*)

Estimates the independent probability of each variable of being 1.

Parameters
`dataset` – dataset from which learn the probabilistic model.

print_structure() → list

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.

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

learn(*dataset: array*, **args*, ***kwargs*)

Estimates the independent Gaussian for each variable.

Parameters
dataset – dataset from which learn the probabilistic model.

print_structure() → list

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.speda module

class EDAspy.optimization.multivariate.speda.**SPEDA**(*size_gen: int*, *max_iter: int*, *dead_iter: int*, *n_variables: int*, *landscape_bounds: tuple*, *l: float*, *alpha: float = 0.5*, *disp: bool = True*, *black_list: Optional[list] = None*, *white_list: Optional[list] = None*, *parallelize: bool = False*, *init_data: Optional[array] = None*)

Bases: *EDA*

Semiparametric Estimation of Distribution Algorithm [1]. This type of Estimation-of-Distribution Algorithm uses a semiparametric Bayesian network [2] which allows dependencies between variables which have been estimated using KDE with variables which fits a Gaussian distribution. By this way, it avoid the assumption of Gaussianity in the variables of the optimization problem. This multivariate probabilistic model is updated in each iteration with the best individuals of the previous generations.

SPEDA has shown to improve the results for more complex optimization problem compared to the univariate EDAs that can be found implemented in this package, multivariate EDAs such as EGNA, or EMNA, and other population-based algorithms. See [1] for numerical results.

Example

This example uses some very well-known benchmarks from CEC14 conference to be solved using a Semiparametric Estimation of Distribution Algorithm (SPEDA).

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

benchmarking = ContinuousBenchmarkingCEC14(10)

speda = SPEDA(size_gen=300, max_iter=100, dead_iter=20, n_variables=10,
              landscape_bounds=(-60, 60), l=10)

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

References

[1]: Vicente P. Soloviev, Concha Bielza and Pedro Larrañaga. Semiparametric Estimation of Distribution Algorithm for continuous optimization. 2022

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

property pm: ProbabilisticModel

Returns the probabilistic model used in the EDA implementation.

Returns
probabilistic model.

Return type
ProbabilisticModel

property init: GenInit

Returns the initializer used in the EDA implementation.

Returns
initializer.

Return type
GenInit

export_settings() → dict

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

Returns
configuration dictionary.

Return type
dict

minimize(*cost_function: callable*, *output_runtime: bool = True*, **args*, ***kwargs*) → *EdaResult*

Minimize function to execute the EDA optimization. By default, the optimizer is designed to minimize a cost function; if maximization is desired, just add a minus sign to your cost function.

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 with results and information.

Return type
EdaResult

EDAspy.optimization.multivariate.keda module

class EDAspy.optimization.multivariate.keda.**MultivariateKEDA**(*size_gen: int*, *max_iter: int*, *dead_iter: int*, *n_variables: int*, *landscape_bounds: tuple*, *l: float*, *alpha: float = 0.5*, *disp: bool = True*, *black_list: Optional[list] = None*, *white_list: Optional[list] = None*, *parallelize: bool = False*, *init_data: Optional[array] = None*)

Bases: *EDA*

Kernel Estimation of Distribution Algorithm [1]. This type of Estimation-of-Distribution Algorithm uses a KDE Bayesian network [2] which allows dependencies between variables which have been estimated using KDE. This multivariate probabilistic model is updated in each iteration with the best individuals of the previous generations.

Example

This example uses some very well-known benchmarks from CEC14 conference to be solved using a Kernel Estimation of Distribution Algorithm (KEDA).

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

benchmarking = ContinuousBenchmarkingCEC14(10)

keda = MultivariateKEDA(size_gen=300, max_iter=100, dead_iter=20, n_variables=10,
                        landscape_bounds=(-60, 60), l=10)

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

References

[1]: Vicente P. Soloviev, Concha Bielza and Pedro Larrañaga. Semiparametric Estimation of Distribution Algorithm for continuous optimization. 2022

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

property pm: ProbabilisticModel

Returns the probabilistic model used in the EDA implementation.

Returns
probabilistic model.

Return type
ProbabilisticModel

property init: GenInit

Returns the initializer used in the EDA implementation.

Returns
initializer.

Return type
GenInit

export_settings() → dict

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

Returns
configuration dictionary.

Return type
dict

minimize(*cost_function: callable*, *output_runtime: bool = True*, **args*, ***kwargs*) → *EdaResult*

Minimize function to execute the EDA optimization. By default, the optimizer is designed to minimize a cost function; if maximization is desired, just add a minus sign to your cost function.

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 with results and information.

Return type
EdaResult

EDAspy.optimization.multivariate.keda module

class EDAspy.optimization.multivariate.keda.**MultivariateKEDA**(*size_gen: int*, *max_iter: int*, *dead_iter: int*, *n_variables: int*, *landscape_bounds: tuple*, *l: float*, *alpha: float = 0.5*, *disp: bool = True*, *black_list: Optional[list] = None*, *white_list: Optional[list] = None*, *parallelize: bool = False*, *init_data: Optional[array] = None*)

Bases: *EDA*

Kernel Estimation of Distribution Algorithm [1]. This type of Estimation-of-Distribution Algorithm uses a KDE Bayesian network [2] which allows dependencies between variables which have been estimated using KDE. This multivariate probabilistic model is updated in each iteration with the best individuals of the previous generations.

Example

This example uses some very well-known benchmarks from CEC14 conference to be solved using a Kernel Estimation of Distribution Algorithm (KEDA).

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

benchmarking = ContinuousBenchmarkingCEC14(10)

keda = MultivariateKEDA(size_gen=300, max_iter=100, dead_iter=20, n_variables=10,
                        landscape_bounds=(-60, 60), l=10)

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

References

[1]: Vicente P. Soloviev, Concha Bielza and Pedro Larrañaga. Semiparametric Estimation of Distribution Algorithm for continuous optimization. 2022

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

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*, *black_list: Optional[list] = None*, *white_list: Optional[list] = None*, *parallelize: bool = False*, *init_data: Optional[array] = None*)

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.

property pm: ProbabilisticModel

Returns the probabilistic model used in the EDA implementation.

Returns
probabilistic model.

Return type
ProbabilisticModel

property init: GenInit

Returns the initializer used in the EDA implementation.

Returns
initializer.

Return type
GenInit

export_settings() → dict

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

Returns
configuration dictionary.

Return type
dict

minimize(*cost_function: callable*, *output_runtime: bool = True*, **args*, ***kwargs*) → *EdaResult*

Minimize function to execute the EDA optimization. By default, the optimizer is designed to minimize a cost function; if maximization is desired, just add a minus sign to your cost function.

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 with results and information.

Return type

EdaResult

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*, *parallelize: bool = False*, *init_data: Optional[array] = None*)

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.

property pm: ProbabilisticModel

Returns the probabilistic model used in the EDA implementation.

Returns
probabilistic model.

Return type
ProbabilisticModel

property init: GenInit

Returns the initializer used in the EDA implementation.

Returns
initializer.

Return type
GenInit

export_settings() → dict

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

Returns
configuration dictionary.

Return type
dict

minimize(*cost_function: callable*, *output_runtime: bool = True*, **args*, ***kwargs*) → *EdaResult*

Minimize function to execute the EDA optimization. By default, the optimizer is designed to minimize a cost function; if maximization is desired, just add a minus sign to your cost function.

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 with results and information.

Return type
EdaResult

Module contents

EDAspy.optimization.univariate package

Submodules

EDAspy.optimization.univariate.umda_binary module

class EDAspy.optimization.univariate.umda_binary.**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*, *parallelize: bool = False*, *init_data: Optional[array] = None*)

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.

property pm: ProbabilisticModel

Returns the probabilistic model used in the EDA implementation.

Returns
probabilistic model.

Return type
ProbabilisticModel

export_settings() → dict

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

Returns
configuration dictionary.

Return type
dict

property init: GenInit

Returns the initializer used in the EDA implementation.

Returns
initializer.

Return type
GenInit

minimize(*cost_function: callable*, *output_runtime: bool = True*, **args*, ***kwargs*) → *EdaResult*

Minimize function to execute the EDA optimization. By default, the optimizer is designed to minimize a cost function; if maximization is desired, just add a minus sign to your cost function.

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 with results and information.

Return type
EdaResult

EDAspy.optimization.univariate.umda_continuous module

class EDAspy.optimization.univariate.umda_continuous.**UMDAc**(*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*, *parallelize: bool = False*, *init_data: Optional[array] = None*)

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

Example

This short example runs UMDAc for a benchmark function optimization 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

property init: GenInit

Returns the initializer used in the EDA implementation.

Returns
initializer.

Return type
GenInit

export_settings() → dict

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

Returns
configuration dictionary.

Return type
dict

minimize(*cost_function: callable*, *output_runtime: bool = True*, **args*, ***kwargs*) → *EdaResult*

Minimize function to execute the EDA optimization. By default, the optimizer is designed to minimize a cost function; if maximization is desired, just add a minus sign to your cost function.

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 with results and information.

Return type
EdaResult

property pm: ProbabilisticModel

Returns the probabilistic model used in the EDA implementation.

Returns
probabilistic model.

Return type
ProbabilisticModel

EDAspy.optimization.univariate.keda

class EDAspy.optimization.univariate.keda.**UnivariateKEDA**(*size_gen: int, max_iter: int, dead_iter: int, n_variables: int, alpha: float = 0.5, landscape_bounds: tuple = (-100, 100), elite_factor: float = 0.4, disp: bool = True, parallelize: bool = False, init_data: Optional[array] = None*)

Bases: *EDA*

Univariate Kernel Density Estimation Algorithm (u_KEDA). New individuals are sampled from a KDE model. It can be used for hyper-parameter optimization or to optimize a function.

u_KEDA [1] is a specific type of Estimation of Distribution Algorithm (EDA) where new individuals are sampled from univariate KDEs 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 real data so new individuals are sampled from a univariate probabilistic model updated in each iteration.

Example

This short example runs UMDAc for a benchmark function optimization problem in the continuous space.

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

n_vars = 10
benchmarking = ContinuousBenchmarkingCEC14(n_vars)

keda = UnivariateKEDA(size_gen=100, max_iter=100, dead_iter=10, n_variables=10,␣
↪alpha=0.5)
# We leave bound by default
eda_result = keda.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.

property init: GenInit

Returns the initializer used in the EDA implementation.

Returns
initializer.

Return type
GenInit

property pm: ProbabilisticModel

Returns the probabilistic model used in the EDA implementation.

Returns
probabilistic model.

Return type
ProbabilisticModel

export_settings() → dict

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

Returns
configuration dictionary.

Return type
dict

minimize(*cost_function: callable*, *output_runtime: bool = True*, **args*, ***kwargs*) → *EdaResult*

Minimize function to execute the EDA optimization. By default, the optimizer is designed to minimize a cost function; if maximization is desired, just add a minus sign to your cost function.

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 with results and information.

Return type
EdaResult

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*, *parallelize: bool = False*, *init_data: Optional[array] = None*, **args*, ***kwargs*)

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.

Returns
configuration dictionary.

Return type
dict

minimize(*cost_function: callable*, *output_runtime: bool = True*, **args*, ***kwargs*) → *EdaResult*

Minimize function to execute the EDA optimization. By default, the optimizer is designed to minimize a cost function; if maximization is desired, just add a minus sign to your cost function.

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 with results and information.

Return type

EdaResult

property pm: ProbabilisticModel

Returns the probabilistic model used in the EDA implementation.

Returns

probabilistic model.

Return type

ProbabilisticModel

property init: GenInit

Returns the initializer used in the EDA implementation.

Returns

initializer.

Return type

GenInit

EDAspy.optimization.eda_result module

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

Bases: `object`

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

EDAspy.optimization.tools module

EDAspy.optimization.tools.**arcs2adj_mat**(*arcs: list*, *n_variables: int*) → array

This function transforms the list of arcs in the BN structure to an adjacency matrix.

Parameters

- **arcs** (*list*) – list of arcs in the BN structure.
- **n_variables** (*int*) – number of variables.

Returns

adjacency matrix

Return type

np.array

`EDAspy.optimization.tools.plot_bn(arcs: list, var_names: list, pos: Optional[dict] = None, curved_arcs: bool = True, curvature: float = -0.3, node_size: int = 500, node_color: str = 'red', edge_color: str = 'black', arrow_size: int = 15, node_transparency: float = 0.9, edge_transparency: float = 0.9, node_line_widths: float = 2, title: Optional[str] = None, output_file: Optional[str] = None)`

This function Plots a BN structure as a directed acyclic graph.

Parameters

- **arcs** (*list(tuple)*) – Arcs in the BN structure.
- **var_names** (*list*) – List of variables.
- **pos** (*dict {name of variables: tuples with coordinates}*) – Positions in the plot for each node.
- **curved_arcs** (*bool*) – True if curved arcs are desired.
- **curvature** (*float*) – Radians of curvature for edges. By default, -0.3.
- **node_size** (*int*) – Size of the nodes in the graph. By default, 500.
- **node_color** (*str*) – Color set to nodes. By default, 'red'.
- **edge_color** (*str*) – Color set to edges. By default, 'black'.
- **arrow_size** (*int*) – Size of arrows in edges. By default, 15.
- **node_transparency** (*float*) – Alpha value [0, 1] that defines the transparency of the node. By default, 0.9.
- **edge_transparency** (*float*) – Alpha value [0, 1] that defines the transparency of the edge. By default, 0.9.
- **node_line_widths** (*float*) – Width of the nodes contour lines. By default, 2.0.
- **title** (*str*) – Title for Figure. By default, None.
- **output_file** (*str*) – Path to save the figure locally.

Returns

Figure.

Module contents

5.1.1.3 EDAspy.timeseries package

Submodules

EDAspy.timeseries.TS_transformations module

`class EDAspy.timeseries.TS_transformations.TSTransformations(data)`

Bases: `object`

Tool to calculate time series transformations. Some time series transformations are given. This is just a very simple tool. It is not mandatory to use this tool to use the time series transformations selector. It is only disposed to be handy.

`data = -1`

`de_trending`(*variable*, *plot=False*)

Removes the trend of the time series.

Parameters

- **variable** (*string*) – string available in data DataFrame
- **plot** (*bool*) – if True plot is give, if False, not

Returns
time series detrended

Return type
list

`log`(*variable*, *plot=False*)

Calculate the logarithm of the time series.

Parameters

- **variable** (*string*) – name of variables
- **plot** (*bool*) – if True a plot is given.

Returns
time series transformation

Return type
list

`box_cox`(*variable*, *lmbda*, *plot=False*)

Calculate Box Cox time series transformation.

Parameters

- **variable** (*string*) – name of variable
- **lmbda** (*float*) – lambda parameter of Box Cox transformation
- **plot** (*bool*) – if True, plot is given.ç

Returns
time series transformation

Return type
list

`smoothing`(*variable*, *window*, *plot=False*)

Calculate time series smoothing transformation.

Parameters

- **variable** (*string*) – name of variable
- **window** (*int*) – number of previous instances taken to smooth.
- **plot** (*bool*) – if True, plot is given

Returns
time series transformation

Return type
list

`power`(*variable*, *power*, *plot=False*)

Calculate power time series transformation.

Parameters

- **variable** (*string*) – name of variable
- **power** (*int*) – exponent to calculate
- **plot** (*bool*) – if True, plot is given

Returns
time series transformation

Return type
list

`exponential`(*variable*, *numerator*, *plot=False*)

Calculate exponential time series transformation.

Parameters

- **variable** (*string*) – name of variable
- **numerator** (*int*) – numerator of the transformation
- **plot** (*bool*) – if True, plot is given

Returns
time series transformation

Return type
list

EDAspy.timeseries.TransformationsFeatureSelection module

class `EDAspy.timeseries.TransformationsFeatureSelection.TransformationsFSEDA`(*max_it*, *dead_it*, *size_gen*, *alpha*, *vector*, *array_transformations*, *cost_function*)

Bases: `object`

Estimation of Distribution Algorithm that uses a Dirichlet distribution to select among the different time series transformations that best improve the cost function to optimize.

…

Attributes:

generation: pandas DataFrame
Last generation of the algorithm.

best_MAE: float
Best cost found.

best_ind: pandas DataFrame
First row of the pandas DataFrame. Can be casted to dictionary.

history_best: list
List of the costs found during runtime.

size_gen: int
Parameter set by user. Number of the individuals in each generation.

max_it: int
Parameter set by user. Maximum number of iterations of the algorithm.

dead_it: int
Parameter set by user. Number of iterations after which, if no improvement reached, the algorithm finishes.

vector: pandas DataFrame
When initialized, parameters set by the user. When finished, statistics learned by the user.

cost_function:
Set by user. Cost function set to optimize.

`generation = Empty DataFrame Columns: [] Index: []`

`output_plot = ''`

`historic_best = []`

`best_MAE = 99999999999`

`best_ind = ''`

`new_generation()`

Creates a new generation of individuals. Updates the generation DataFrame

`check_generation()`

Check the cost of each individual of the generation in the cost function

`individuals_selection()`

Selection of the best individuals to mutate the next generation

`update_vector_probabilities()`

Re-build the vector of statistics based on the selection of the best individuals of the generation.

`run`(*output=True*)

Algorithm run execution

Parameters
`output` (*bool*) – If True then an output is printed in each iteration. Otherwise, not

Returns
best_individual, best MAE found

Return type
list, float

Module contents

5.1.2 Module contents

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

e

INDEX

A

B

C

D

E

G

H

I

K

L

M

W