
desdeo_problem

Release 1.0

Multiobjective Optimization Group

Jun 23, 2022

CONTENTS

1	Requirements	3
2	Installation	5
2.1	For users	5
2.2	For developers	5
3	Indices and tables	59
	Python Module Index	61
	Index	63

Contains tools to model and define multiobjective optimization problems to be used in the DESDEO framework.

REQUIREMENTS

- Python 3.7 or newer.
- [Poetry dependency manager](#) : Only for developers.

See *pyproject.toml* for Python package requirements.

INSTALLATION

To install and use this package on a *nix-based system, follow one of the following procedures.

2.1 For users

First, create a new virtual environment for the project. Then install the package using the following command:

```
$ pip install desdeo_problem
```

2.2 For developers

Download the code or clone it with the following command:

```
$ git clone https://github.com/industrial-optimization-group/desdeo-problem
```

Then, create a new virtual environment for the project and install the package in it:

```
$ cd desdeo-problem  
$ poetry init  
$ poetry install
```

2.2.1 API Documentation

desdeo_problem.problem Package

Desdeo-problem package

This package is for creating a problem for desdeo to solve. It includes modules for Variables, Objectives, Constraints, and actual Problem.

Functions

<code>constraint_function_factory</code> (lhs, rhs, operator)	A function that creates an evaluator.
<code>variable_builder</code> (names, initial_values[, ...])	Automatically build all variable objects.

`constraint_function_factory`

`desdeo_problem.problem.constraint_function_factory`(lhs, rhs, operator)

A function that creates an evaluator.

This function creates an evaluator to be used with the `ScalarConstraint` class. Constraints should be formulated in a way where all the mathematical expression are on the left hand side, and the constants on the right hand side.

Parameters

- **lhs** (*Callable*) – The left hand side of the constraint. Should be a callable function representing a mathematical expression.
- **rhs** (*float*) – The right hand side of a constraint. Represents the right hand side of the constraint.
- **operator** (*str*) – The kind of constraint. Can be ‘==’, ‘<’, ‘>’.

Returns A function that can be called to evaluate the rhs and which returns representing how the constraint is obeyed. A negative value represent a violation of the constraint and a positive value an agreement with the constraint. The absolute value of the float is a direct indicator how the constraint is violated/agreed with.

Return type `Callable`

Raises `ValueError` – The supplied operator is not supported.

`variable_builder`

`desdeo_problem.problem.variable_builder`(names, initial_values, lower_bounds=None, upper_bounds=None)

Automatically build all variable objects.

Parameters

- **names** (*List[str]*) – Names of the variables
- **initial_values** (*np.ndarray*) – Initial values taken by the variables.
- **lower_bounds** (*Union[List[float], np.ndarray, optional]*) – Lower bounds of the variables. If None, it defaults to negative infinity. Defaults to None.
- **upper_bounds** (*Union[List[float], np.ndarray, optional]*) – Upper bounds of the variables. If None, it defaults to positive infinity. Defaults to None.

Raises `VariableError` – Lengths of the input arrays are different.

Returns List of variable objects

Return type `List[Variable]`

Classes

<i>ScalarConstraint</i> (name, n_decision_vars, ...)	A simple scalar constraint that evaluates to a single scalar.
<i>ConstraintError</i>	Raised when an error related to the Constraint class is encountered.
<i>ConstraintBase</i> ()	Base class for constraints.
<i>ObjectiveBase</i> ()	The abstract base class for objectives.
<i>ObjectiveError</i>	Raised when an error related to the Objective class is encountered.
<i>ObjectiveEvaluationResults</i> (objectives[, ...])	The return object of <problem>.evaluate methods.
<i>VectorObjectiveBase</i> ()	The abstract base class for multiple objectives which are calculated at once.
<i>ScalarObjective</i> (name, evaluator[, ...])	A simple objective function that returns a scalar.
<i>_ScalarObjective</i> (name, evaluator[, ...])	
<i>VectorObjective</i> (name, evaluator[, ...])	An objective object that calculated one or more objective functions.
<i>ScalarDataObjective</i> (name, data[, evaluator, ...])	A simple Objective class for single valued objectives.
<i>_ScalarDataObjective</i> (name, data[, ...])	
<i>VectorDataObjective</i> (name, data[, evaluator, ...])	A Objective class for multi/valued objectives.
<i>ProblemError</i>	Raised when an error related to the Problem class is encountered.
<i>ProblemBase</i> ()	The base class for the problems.
<i>EvaluationResults</i> (objectives, fitness[, ...])	The return object of <problem>.evaluate methods.
<i>ScalarMOPProblem</i> (objectives, variables, ...)	A multiobjective optimization problem.
<i>ScalarDataProblem</i> (decision_vectors, ...)	A problem class for case where the data is pre-computed.
<i>MOPProblem</i> (objectives, variables[, ...])	An user defined multiobjective optimization problem.
<i>DataProblem</i> (data, variable_names, ...[, ...])	A class for a data based problem.
<i>ExperimentalProblem</i> (variable_names, ...[, ...])	A problem class for data-based problem.
<i>VariableError</i>	Raised when an error is encountered during the handling of the Variable objects.
<i>VariableBuilderError</i>	Raised when an error is encountered during the handling of the Variable objects.
<i>Variable</i> (name, initial_value[, lower_bound, ...])	Simple variable with a name, initial value and bounds.
<i>DiscreteDataProblem</i> (data, variable_names, ...)	A problem class for data-based problems with discrete values.

ScalarConstraint

class desdeo_problem.problem.**ScalarConstraint**(name, n_decision_vars, n_objective_funs, evaluator)

Bases: *desdeo_problem.problem.Constraint.ConstraintBase*

A simple scalar constraint that evaluates to a single scalar.

Parameters

- **name** (*str*) – Name of the constraint.
- **n_decision_vars** (*int*) – Number of decision variables present in the constraint.
- **n_objective_funs** (*int*) – Number of objective functions present in the constraint.

- **evaluator** (*Callable*) – A callable to evaluate the constraint.

__name

Name of the constraint.

Type str

__n_decision_vars

Number of decision variables present in the constraint.

Type int

__n_objective_funs

Number of objective functions present in the constraint.

Type int

__evaluator

A callable to evaluate the constraint.

Type Callable

Attributes Summary

<i>evaluator</i>	constraint evaluator callable
<i>n_decision_vars</i>	number of decision variables
<i>n_objective_funs</i>	number of objective functions
<i>name</i>	name

Methods Summary

<i>evaluate</i> (decision_vector, objective_vector)	Evaluate the constraint.
---	--------------------------

Attributes Documentation

evaluator

constraint evaluator callable

Returns A callable to evaluate the constraint.

Return type Callable

Type Property

n_decision_vars

number of decision variables

Returns Number of decision variables

Return type int

Type Property

n_objective_funs

number of objective functions

Returns Number of objective functions

Return type int

Type Property

name

name

Returns Name of the constraint

Return type str

Type Property

Methods Documentation**evaluate**(*decision_vector*, *objective_vector*)

Evaluate the constraint.

This evaluates the constraint and return a float indicating how and if the constraint was violated. A negative value indicates a violation and a positive value indicates a non-violation.

Parameters

- **decision_vector** (*np.ndarray*) – A *decision_vector* containing the values of the decision variables.
- **objective_vector** (*np.ndarray*) – A *decision_vector* containing the values of the objective functions.

Returns A float indicating how the constraint holds.

Return type float

Raises *ConstraintError* – When something goes wrong evaluating the constraint or the objectives and decision vectors are of wrong shape.

ConstraintError**exception** `desdeo_problem.problem.ConstraintError`

Raised when an error related to the Constraint class in encountered.

ConstraintBase**class** `desdeo_problem.problem.ConstraintBase`

Bases: `abc.ABC`

Base class for constraints.

Methods Summary

<code>evaluate</code> (<code>decision_vector</code> , <code>objective_vector</code>)	Evaluate the constraint functions.
--	------------------------------------

Methods Documentation

abstract evaluate(*decision_vector*, *objective_vector*)

Evaluate the constraint functions.

This function will evaluate constraints and return a float indicating how severely the constraint has been broken.

Parameters

- **decision_vector** (*np.ndarray*) – A `decision_vector` containing the decision variable values.
- **objective_vector** (*np.ndarray*) – A `decision_vector` containing the objective function values.

Returns A float representing how and if the constraint has been violated. A positive value represents no violation and a negative value represents a violation. The absolute value of the returned float functions as an indicator of the severity of the violation (or how well the constraint holds, if the returned value of positive).

Return type float

ObjectiveBase

class `desdeo_problem.problem.ObjectiveBase`

Bases: `abc.ABC`

The abstract base class for objectives.

Methods Summary

<code>evaluate</code> (<code>decision_vector</code> [, <code>use_surrogate</code>])	Evaluates the objective according to a decision variable vector.
---	--

Methods Documentation

evaluate(*decision_vector*, *use_surrogate=False*)

Evaluates the objective according to a decision variable vector.

Uses surrogate model if `use_surrogates` is true. If `use_surrogates` is False, uses `func_evaluate` which evaluates using the true objective function.

Parameters

- **decision_vector** (*np.ndarray*) – A vector of Variables to be used in the evaluation of the objective.
- **use_surrogate** (*bool*) – A boolean which determines whether to use surrogates or true function evaluator. False by default.

Return type *ObjectiveEvaluationResults*

ObjectiveError

exception `desdeo_problem.problem.ObjectiveError`

Raised when an error related to the Objective class is encountered.

ObjectiveEvaluationResults

class `desdeo_problem.problem.ObjectiveEvaluationResults`(*objectives: Union[float, numpy.ndarray], uncertainty: Union[None, float, numpy.ndarray] = None*)

Bases: tuple

The return object of `<problem>.evaluate` methods.

objectives

The objective function value/s for the input vector.

Type Union[float, np.ndarray]

uncertainty

The uncertainty in the objective value/s.

Type Union[None, float, np.ndarray]

VectorObjectiveBase

class `desdeo_problem.problem.VectorObjectiveBase`

Bases: `abc.ABC`

The abstract base class for multiple objectives which are calculated at once.

Methods Summary

<code>evaluate</code> (<i>decision_vector</i> [, <i>use_surrogate</i>])	Evaluates the objective according to a decision variable vector.
---	--

Methods Documentation

evaluate(*decision_vector*, *use_surrogate=False*)

Evaluates the objective according to a decision variable vector.

Uses surrogate model if *use_surrogates* is true. If *use_surrogates* is False, uses *func_evaluate* which evaluates using the true objective function.

Parameters

- **decision_vector** (*np.ndarray*) – A vector of Variables to be used in the evaluation of the objective.

- **use_surrogate** (*bool*) – A boolean which determines whether to use surrogates or true function evaluator. False by default.

Return type *ObjectiveEvaluationResults*

ScalarObjective

class `desdeo_problem.problem.ScalarObjective`(*name, evaluator, lower_bound=- inf, upper_bound=inf, maximize=None*)

Bases: *desdeo_problem.problem.Objective.ObjectiveBase*

A simple objective function that returns a scalar.

To be depreciated

Parameters

- **name** (*str*) – Name of the objective.
- **evaluator** (*Callable*) – The function to evaluate the objective's value.
- **lower_bound** (*float*) – The lower bound of the objective.
- **upper_bound** (*float*) – The upper bound of the objective.
- **maximize** (*bool*) – Boolean to determine whether the objective is to be maximized.

__name

Name of the objective.

Type *str*

__value

The current value of the objective function.

Type *float*

__evaluator

The function to evaluate the objective's value.

Type *Callable*

__lower_bound

The lower bound of the objective.

Type *float*

__upper_bound

The upper bound of the objective.

Type *float*

maximize

List of boolean to determine whether the objectives are to be maximized. All false by default

Type *List[bool]*

Raises *ObjectiveError* – When ill formed bounds are given.

Attributes Summary

<i>evaluator</i>	evaluator for the objective
<i>lower_bound</i>	lower bound of the objective.
<i>name</i>	name
<i>upper_bound</i>	upper bound of the objective.
<i>value</i>	value

Attributes Documentation

evaluator

evaluator for the objective

Returns evaluator

Return type callable

Type Property

lower_bound

lower bound of the objective.

Returns lower bound of the objective

Return type float

Type Property

name

name

Returns name

Return type str

Type Property

upper_bound

upper bound of the objective.

Returns upper bound of the objective

Return type float

Type Property

value

value

Returns value

Return type float

Type Property

`_ScalarObjective`

```
class desdeo_problem.problem._ScalarObjective(name, evaluator, lower_bound=- inf, upper_bound=inf,
                                             maximize=None)
```

Bases: `desdeo_problem.problem.Objective.ScalarObjective`

`VectorObjective`

```
class desdeo_problem.problem.VectorObjective(name, evaluator, lower_bounds=None,
                                             upper_bounds=None, maximize=None)
```

Bases: `desdeo_problem.problem.Objective.VectorObjectiveBase`

An objective object that calculated one or more objective functions.

To be renamed to `Objective`

Attributes: `__name` (`List[str]`): Names of the various objectives in a list `__evaluator` (`Callable`): The function that evaluates the objective values `__lower_bounds` (`Union[List[float], np.ndarray]`, optional): Lower bounds

of the objective values. Defaults to `None`.

`__upper_bounds` (`Union[List[float], np.ndarray]`, optional): **Upper bounds** of the objective values. Defaults to `None`.

`__maximize` (`List[bool]`): **List of boolean to determine whether the** objectives are to be maximized. All false by default

`__n_of_objects` (`int`): The number of objectives

Parameters

- **name** (`List[str]`) – Names of the various objectives in a list
- **evaluator** (`Callable`) – The function that evaluates the objective values
- **lower_bounds** (`Union[List[float], np.ndarray]`, optional) – Lower bounds of the objective values. Defaults to `None`.
- **upper_bounds** (`Union[List[float], np.ndarray]`, optional) – Upper bounds of the objective values. Defaults to `None`.
- **maximize** (`Optional[List[bool]]`) – *List of boolean to determine whether the objectives are to be maximized. All false by default*

Attributes Summary

<code>evaluator</code>	evaluator
<code>lower_bounds</code>	lower bounds
<code>n_of_objectives</code>	number of objectives
<code>name</code>	name
<code>upper_bounds</code>	upper bounds
<code>values</code>	values

Attributes Documentation

evaluator

evaluator

Returns Evaluator of the objective

Return type Callable

Type Property

lower_bounds

lower bounds

Returns lower bounds for vector valued objective.

Return type np.ndarray

Type Property

n_of_objectives

number of objectives

Returns the number of objectives

Return type int

Type Property

name

name

Returns name of the objective

Return type str

Type Property

upper_bounds

upper bounds

Returns upper bounds for vector valued objective.

Return type np.ndarray

Type Property

values

values

Returns Evaluated value and uncertainty of evaluation

Return type Tuple[float]

Type Property

ScalarDataObjective

class `desdeo_problem.problem.ScalarDataObjective`(*name, data, evaluator=None, lower_bound=- inf, upper_bound=inf, maximize=None*)

Bases: `desdeo_problem.problem.Objective.ScalarObjective`

A simple Objective class for single valued objectives.

To be deprecated.

Use when the an evaluator/simulator returns a single objective value or when there is no evaluator/simulator

X

Dataframe with corresponds the points where the objective value is known.

Type `pd.DataFrame`

y

The objective values corresponding the points.

Type `pd.Series`

variable_names

The names of the variables in X

Type `pd.Index`

_model

Model of the data

Type `BaseRegressor`

Parameters

- **name** (`List[str]`) – The name of the objective. Should be the same as a column name in the data.
- **data** (`pd.DataFrame`) – The data in a pandas dataframe. The columns should be named after variables/objective.
- **evaluator** (`Union[None, Callable]`, *optional*) – A python function that contains the analytical function or calls the simulator to get the true objective value. By default None, as this is not required.
- **lower_bound** (`float`, *optional*) – Lower bound of the objective, by default `-np.inf`
- **upper_bound** (`float`, *optional*) – Upper bound of the objective, by default `np.inf`
- **maximize** (`List[bool]`, *optional*) – Boolean describing whether the objective is to be maximized or not, by default None, which defaults to `[False]`, hence minimizes.

Raises `ObjectiveError` – When the name provided during initialization does not match any name in the columns of the data provided during initialization.

Methods Summary

<code>train(model[, model_parameters, index, data])</code>	Train surrogate model for the objective.
--	--

Methods Documentation

train(*model*, *model_parameters=None*, *index=None*, *data=None*)

Train surrogate model for the objective.

Parameters

- **model** (*BaseRegressor*) – A regressor. The regressor, when initialized, should have a fit method and a predict method. The predict method should return the predicted objective value, as well as the uncertainty value, in a tuple. If the regressor does not support calculating uncertainty, return a tuple of objective value and None.
- **model_parameters** (*Dict*) – ****model_parameters** is passed to the model when initialized.
- **index** (*List[int]*, *optional*) – Indices of the samples (in self.X and self.y), to be used to train the surrogate model. By default None, which trains the model on the entire dataset. This behaviour may be changed in the future to support test-train split or cross validation.
- **data** (*pd.DataFrame*, *optional*) – Extra data to be used for training only. This data is not saved. By default None, which then uses self.X and self.y for training.

Raises *ObjectiveError* – For unexpected errors

ScalarDataObjective

class `desdeo_problem.problem._ScalarDataObjective`(*name*, *data*, *evaluator=None*, *lower_bound=-inf*, *upper_bound=inf*, *maximize=None*)

Bases: `desdeo_problem.problem.Objective.ScalarDataObjective`

VectorDataObjective

class `desdeo_problem.problem.VectorDataObjective`(*name*, *data*, *evaluator=None*, *lower_bounds=None*, *upper_bounds=None*, *maximize=None*)

Bases: `desdeo_problem.problem.Objective.VectorObjective`

A Objective class for multi/valued objectives.

Use when the an evaluator/simulator returns a multiple objective values or when there is no evaluator/simulator.

x

Dataframe with corresponds the points where the objective value is known.

Type `pd.DataFrame`

y

The objective values corresponding the points.

Type `pd.Series`

variable_names

The names of the variables in X

Type pd.Index

_model

BaseRegressor (or None if not trained) models for each objective, keys are the names of objectives.

Type Dict

_model_trained

boolean if model is trained for each objective, keys are the names of objectives. Default false.

Type Dict

Parameters

- **name** (*List[str]*) – The name of the objective. Should be the same as a column name in the data.
- **data** (*pd.DataFrame*) – The data in a pandas dataframe. The columns should be named after variables/objective.
- **evaluator** (*Union[None, Callable]*, *optional*) – A python function that contains the analytical function or calls the simulator to get the true objective value. By default None, as this is not required.
- **lower_bound** (*float*, *optional*) – Lower bound of the objective, by default -np.inf
- **upper_bound** (*float*, *optional*) – Upper bound of the objective, by default np.inf
- **maximize** (*List[bool]*, *optional*) – Boolean describing whether the objective is to be maximized or not, by default None, which defaults to [False], hence minimizes.

Raises *ObjectiveError* – When the name provided during initialization does not match any name in the columns of the data provided during initialization.

Methods Summary

<code>train(models[, model_parameters, index, data])</code>	Train surrogate models for the objective.
---	---

Methods Documentation

train(*models, model_parameters=None, index=None, data=None*)

Train surrogate models for the objective.

Parameters

- **model** (*BaseRegressor* or *List[BaseRegressors]*) – A regressor or a list of regressors. The regressor/s, when initialized, should have a fit method and a predict method. The predict method should return the predicted objective value, as well as the uncertainty value, in a tuple. If the regressor does not support calculating uncertainty, return a tuple of objective value and None. If a single regressor is provided, that regressor is used for all the objectives. If a list of regressors is provided, and if the list contains one regressor for each objective, then those individual regressors are used to model the objectives. If the number of regressors is not equal to the number of objectives, an error is raised.

- **model_parameters** (*Dict or List[Dict]*) – The parameters for the regressors. Should be a dict if a single regressor is provided. If a list of regressors is provided, the parameters should be in a list of dicts, same length as the list of regressors(= number of objs).
- **index** (*List[int], optional*) – Indices of the samples (in self.X and self.y), to be used to train the surrogate model. By default None, which trains the model on the entire dataset. This behaviour may be changed in the future to support test-train split or cross validation.
- **data** (*pd.DataFrame, optional*) – Extra data to be used for training only. This data is not saved. By default None, which then uses self.X and self.y for training.

Raises

- **ObjectiveError** – If the formats of the model and model parameters do not match
- **ObjectiveError** – If the lengths of list of models and/or model parameter dictionaries are not equal to the number of objectives.

ProblemError

exception desdeo_problem.problem.**ProblemError**

Raised when an error related to the Problem class is encountered.

ProblemBase

class desdeo_problem.problem.**ProblemBase**

Bases: abc.ABC

The base class for the problems.

All other problem classes should be derived from this.

nadir

Nadir values for the problem, initiated = None

Type np.ndarray

ideal

Ideal values for the problem, initiated = None

Type np.ndarray

nadir_fitness

Fitness values for nadir, initiated = None

Type np.ndarray

ideal_fitness

Fitness values for ideal, initiated = None

Type np.ndarray

__n_of_objectives

Number of objectives, initiated = 0

Type int

__n_of_variables

Number of variables, initiated = 0

Type int

__decision_vectors

Array of decision variable vectors, initiated = None

Type np.ndarray

__objective_vectors

Array of objective variable vectors, initiated = None

Type np.ndarray

Attributes Summary

<i>decision_vectors</i>	Property, returns the decision variable vectors array.
<i>n_of_objectives</i>	Property, returns the number of objectives.
<i>n_of_variables</i>	Property, returns the number of variables.

Methods Summary

<i>evaluate</i> (decision_vectors[, use_surrogate])	Abstract method to evaluate problem.
<i>evaluate_constraint_values</i> ()	Abstract method to evaluate constraint values.
<i>get_variable_bounds</i> ()	Abstract method to get variable bounds

Attributes Documentation**decision_vectors**

Property, returns the decision variable vectors array.

Returns decision vector array

Return type np.ndarray

n_of_objectives

Property, returns the number of objectives.

Returns The number of objectives

Return type int

n_of_variables

Property, returns the number of variables.

Returns The number of variables

Return type int

Methods Documentation

abstract evaluate(*decision_vectors*, *use_surrogate=False*)

Abstract method to evaluate problem.

Evaluates the problem using an ensemble of input vectors. Uses surrogate models if available. Otherwise, it uses the true evaluator.

Parameters

- **decision_vectors** (*np.ndarray*) – An array of decision variable
- **vectors.** (*input*) –
- **use_surrogate** (*bool*) – A bool to control whether to use the true, potentially
- **objectives.** (*expensive function or a surrogate model to evaluate the*) –

Returns

Dict with the following keys:

- **'objectives'** (*np.ndarray*): The objective function values for each input vector.
- **'constraints'** (*Union[np.ndarray, None]*): The constraint values of the problem corresponding each input vector.
- **'fitness'** (*np.ndarray*): Equal to objective values if objective is to be minimized. Multiplied by (-1) if objective to be maximized.
- **'uncertainty'** (*Union[np.ndarray, None]*): The uncertainty in the objective values.

Return type (Dict)

abstract evaluate_constraint_values()

Abstract method to evaluate constraint values.

Evaluate just the constraint function values using the attributes *decision_vectors* and *objective_vectors*

Note: Currently not supported by *ScalarMOPProblem*

Return type *Optional[ndarray]*

abstract get_variable_bounds()

Abstract method to get variable bounds

Return type *Optional[ndarray]*

EvaluationResults

class *desdeo_problem.problem.EvaluationResults*(*objectives: numpy.ndarray, fitness: numpy.ndarray, constraints: Union[None, numpy.ndarray] = None, uncertainty: Union[None, numpy.ndarray] = None*)

Bases: *tuple*

The return object of *<problem>.evaluate* methods.

objectives

The objective function values for each input vector.

Type np.ndarray

fitness

Equal to objective values if objective is to be minimized. Multiplied by (-1) if objective to be maximized.

Type np.ndarray

constraints

The constraint values of the problem corresponding each input vector.

Type Union[None, np.ndarray]

uncertainty

The uncertainty in the objective values.

Type Union[None, np.ndarray]

ScalarMOPProblem

class desdeo_problem.problem.ScalarMOPProblem(*objectives, variables, constraints, nadir=None, ideal=None*)

Bases: *desdeo_problem.problem.Problem.ProblemBase*

A multiobjective optimization problem.

To be deprecated.

A multiobjective optimization problem with user defined objective functions, constraints and variables. The objectives each return a single scalar.

Parameters

- **objectives** (*List[ScalarObjective]*) – A list containing the objectives of the problem.
- **variables** (*List[Variable]*) – A list containing the variables of the problem.
- **constraints** (*List[ScalarConstraint]*) – A list containing the constraints of the problem. If no constraints exist, None may be supplied as the value.
- **nadir** (*Optional[np.ndarray]*) – The nadir point of the problem.
- **ideal** (*Optional[np.ndarray]*) – The ideal point of the problem.

__n_of_objectives

The number of objectives in the problem.

Type int

__n_of_variables

The number of variables in the problem.

Type int

__n_of_constraints

The number of constraints in the problem.

Type int

__nadir

The nadir point of the problem.

Type np.ndarray

__ideal

The ideal point of the problem.

Type np.ndarray

__objectives

A list containing the objectives of the problem.

Type List[*ScalarObjective*]

__constraints

A list containing the constraints of the problem.

Type List[*ScalarConstraint*]

Raises *ProblemError* – Ill formed nadir and/or ideal vectors are supplied.

Attributes Summary

<i>constraints</i>	the list of constraints.
<i>ideal</i>	the ideal point of the problem.
<i>n_of_constraints</i>	the number of constraints.
<i>n_of_objectives</i>	the number of objectives.
<i>n_of_variables</i>	the number of variables.
<i>nadir</i>	the nadir point of the problem.
<i>objectives</i>	the list of objectives.
<i>variables</i>	the list of problem variables.

Methods Summary

<i>evaluate</i> (decision_vectors[, use_surrogate])	Evaluates the problem using an ensemble of input vectors.
<i>evaluate_constraint_values</i> ()	Evaluate constraint values.
<i>get_objective_names</i> ()	Get objective names.
<i>get_uncertainty_names</i> ()	Return the names of the objectives present in the problem in the order they were added.
<i>get_variable_bounds</i> ()	Get the variable bounds.
<i>get_variable_lower_bounds</i> ()	Get variable lower bounds.
<i>get_variable_names</i> ()	Get variable names.
<i>get_variable_upper_bounds</i> ()	Get variable upper bounds.

Attributes Documentation

constraints

the list of constraints.

Returns the list of constraints

Return type List[_*ScalarObjective*]

Type Property

ideal: numpy.ndarray

the ideal point of the problem.

Returns the ideal point of the problem.

Return type np.ndarray

Type Property

n_of_constraints

the number of constraints.

Returns the number of constraints.

Return type int

Type Property

n_of_objectives

the number of objectives.

Returns the number of objectives.

Return type int

Type Property

n_of_variables

the number of variables.

Returns the number of variables.

Return type int

Type Property

nadir: numpy.ndarray

the nadir point of the problem.

Returns the nadir point of the problem.

Return type np.ndarray

Type Property

objectives

the list of objectives.

Returns the list of objectives

Return type List[_*ScalarObjective*]

Type Property

variables

the list of problem variables.

Returns the list of problem variables

Return type List[_ScalarObjective]

Type Property

Methods Documentation**evaluate**(*decision_vectors*, *use_surrogate=False*)

Evaluates the problem using an ensemble of input vectors.

Parameters **decision_vectors** (*np.ndarray*) – An 2D array of decision variable input vectors. Each column represent the values of each decision variable.

Returns

If constraint are defined, returns the objective vector values and corresponding constraint values. Or, if no constraints are defined, returns just the objective vector values with None as the constraint values.

Return type Tuple[np.ndarray, Union[None, np.ndarray]]

Raises *ProblemError* – The decision_vectors have wrong dimensions.

evaluate_constraint_values()

Evaluate constraint values.

Evaluate just the constraint function values using the attributes decision_vectors and objective_vectors

Raises *NotImplementedError* –

Note: Currently not supported by ScalarMOPProblem

Return type Optional[ndarray]

get_objective_names()

Get objective names.

Return the names of the objectives present in the problem in the order they were added.

Returns Names of the objectives in the order they were added.

Return type List[str]

get_uncertainty_names()

Return the names of the objectives present in the problem in the order they were added.

Returns Names of the objectives in the order they were added.

Return type List[str]

get_variable_bounds()

Get the variable bounds.

Return the upper and lower bounds of each decision variable present in the problem as a 2D numpy array. The first column corresponds to the lower bounds of each variable, and the second column to the upper bound.

Returns Lower and upper bounds of each variable as a 2D numpy array. If undefined variables, return None instead.

Return type np.ndarray

get_variable_lower_bounds()

Get variable lower bounds.

Return the lower bounds of each variable as a list. The order of the bounds follows the order the variables were added to the problem.

Returns An array with the lower bounds of the variables.

Return type np.ndarray

get_variable_names()

Get variable names.

Return the variable names of the variables present in the problem in the order they were added.

Returns Names of the variables in the order they were added.

Return type List[str]

get_variable_upper_bounds()

Get variable upper bounds.

Return the upper bounds of each variable as a list. The order of the bounds follows the order the variables were added to the problem. :returns: An array with the upper bounds of the variables. :rtype: np.ndarray

ScalarDataProblem

class desdeo_problem.problem.ScalarDataProblem(*decision_vectors, objective_vectors*)

Bases: *desdeo_problem.problem.Problem.ProblemBase*

A problem class for case where the data is pre-computed.

To be depreciated

Defines a problem with pre-computed data representing a multiobjective optimization problem with scalar valued objective functions.

Parameters

- **decision_vectors** (*np.ndarray*) – A 2D vector of decision_vectors. Each row represents a solution with the value for each decision_vectors defined on the columns.
- **objective_vectors** (*np.ndarray*) – A 2D vector of objective function values. Each row represents one objective vector with the values for the invidual objective functions defined on the columns.

decision_vectors

See args

Type np.ndarray

objective_vectors

See args

Type np.ndarray

__epsilon

A small floating point number to shift the bounds of the variables. See, `get_variable_bounds`, default value 1e-6

Type float

__constraints

A list of defined constraints.

Type List[*ScalarConstraint*]

nadir

The nadir point of the problem.

Type np.ndarray

ideal

The ideal point of the problem.

Type np.ndarray

__model_exists

is there a model for this problem

Type bool

Note: It is assumed that the `decision_vectors` and objectives follow a direct one-to-one mapping, i.e., the objective values on the *i*th row in ‘objectives’ should represent the solution of the multiobjective problem when evaluated with the `decision_vectors` on the *i*th row in ‘`decision_vectors`’.

Attributes Summary

<i>constraints</i>	Constraints.
<i>epsilon</i>	epsilon.

Methods Summary

<i>evaluate</i> (<code>decision_vectors</code>)	Evaluate the values of the objectives at the given decision.
<i>evaluate_constraint_values</i> ()	Evaluate the constraint values.
<i>get_variable_bounds</i> ()	Get the variable bounds.

Attributes Documentation**constraints**

Constraints.

Returns list of the defined constraints

Return type List[*ScalarConstraint*]

Type Property

epsilon

epsilon.

Returns epsilon value (for shifting the bounds of variables)

Return type float

Type Property

Methods Documentation

evaluate(*decision_vectors*)

Evaluate the values of the objectives at the given decision.

Evaluate the values of the objectives corresponding to the decision *decision_vectors*.

Parameters **decision_vectors** (*np.ndarray*) – A 2D array with the decision *decision_vectors* to be evaluated on each row.

Returns

A 2D array with the objective values corresponding to each decision vectors on the rows.

Return type *nd.ndarray*

Note: At the moment, this function just maps the given decision *decision_vectors* to the closest decision variable present (using an L2 distance) in the problem and returns the corresponding objective vector.

evaluate_constraint_values()

Evaluate the constraint values.

Evaluate the constraint values for each defined constraint. A positive value indicates that a constraint is adhered to, a negative value indicates a violated constraint.

Returns

A 2D array with each row representing the constraint values for different objective vectors. One column for each constraint. If no constraint function are defined, returns None.

Return type Optional[*np.ndarray*]

get_variable_bounds()

Get the variable bounds.

Returns

The variable bounds in a stack. The epsilon value will be added to the upper bounds and subtracted from the lower bounds to return closed bounds.

Return type *np.array[float]*

Note: If *self.epsilon* is zero, the bounds will represent an open range.

MOPProblem

class `desdeo_problem.problem.MOPProblem`(*objectives*, *variables*, *constraints=None*, *nadir=None*, *ideal=None*)

Bases: `desdeo_problem.problem.Problem.ProblemBase`

An user defined multiobjective optimization problem.

A multiobjective optimization problem with user defined objective functions, constraints, and variables.

Parameters

- **objectives** (`List[Union[ScalarObjective, VectorObjective]]`) – A list containing the objectives of the problem.
- **variables** (`List[Variable]`) – A list containing the variables of the problem.
- **constraints** (`List[ScalarConstraint]`) – A list of the constraints of the problem.
- **nadir** (`Optional[np.ndarray]`, *optional*) – Nadir point of the problem. Defaults to None.
- **ideal** (`Optional[np.ndarray]`, *optional*) – Ideal point of the problem. Defaults to None.

`__objectives`

A list containing the objectives of the problem.

Type `List[Union[ScalarObjective, VectorObjective]]`

`__variables`

A list containing the variables of the problem.

Type `List[Variable]`

`__constraints`

A list of the constraints of the problem.

Type `List[ScalarConstraint]`

`__nadir`

Nadir point of the problem. Defaults to None.

Type `Optional[np.ndarray]`, *optional*

`__ideal`

Ideal point of the problem. Defaults to None.

Type `Optional[np.ndarray]`, *optional*

`__n_of_variables`

The number of variables

Type `int`

`__n_of_objectives`

The number of objectives

Type `int`

Raises `ProblemError` – If ideal or nadir vectors are not the same size as number of objectives.

Attributes Summary

<i>constraints</i>	list of constraints.
<i>n_of_constraints</i>	number of constraints.
<i>n_of_fitnesses</i>	number of dimensions of the fitness matrix.
<i>n_of_objectives</i>	number of objectives.
<i>n_of_variables</i>	number of variables.
<i>objectives</i>	list of objectives.
<i>variables</i>	List of variables

Methods Summary

<i>evaluate</i> (decision_vectors[, use_surrogate])	Evaluates the problem using an ensemble of input vectors.
<i>evaluate_constraint_values</i> (decision_vectors, ...)	Evaluate constraint values
<i>evaluate_fitness</i> (objective_vectors)	Evaluate fitness of the objectives.
<i>evaluate_objectives</i> (decision_vectors[, ...])	Evaluate objective values of the problem
<i>get_objective_names</i> ()	Get objective names.
<i>get_variable_bounds</i> ()	Get variable bounds.
<i>get_variable_lower_bounds</i> ()	Get variable lower bounds.
<i>get_variable_names</i> ()	Get variable names.
<i>get_variable_upper_bounds</i> ()	Get variable upper bounds.
<i>number_of_objectives</i> (obj_instance)	Return the number of objectives in the given obj_instance.
<i>update_ideal</i> (objective_vectors, fitness)	Update the ideal vector

Attributes Documentation

constraints

list of constraints.

Returns list of constraints

Return type List[*ScalarConstraint*]

Type Property

n_of_constraints

number of constraints.

Returns Number of constraints

Return type int

Type Property

n_of_fitnesses

number of dimensions of the fitness matrix. May be different than the number of objectives in inherited classes.

Returns number of fitness dimensions.

Return type int

Type Property

n_of_objectives

number of objectives.

Returns number of objectives

Return type int

Type Property

n_of_variables

number of variables.

Returns Number of variables.

Return type int

Type Property

objectives

list of objectives.

Returns list of objectives

Return type List[*ScalarObjective*]

Type Property

variables

List of variables

Returns list of variables

Return type List[*Variable*]

Type Property

Methods Documentation

evaluate(*decision_vectors*, *use_surrogate=False*)

Evaluates the problem using an ensemble of input vectors.

Parameters

- **decision_vectors** (*np.ndarray*) – An 2D array of decision variable input vectors. Each column represent the values of each decision variable.
- **use_surrogate** (*bool*) – A bool to control whether to use the true, potentially expensive function or a surrogate model to evaluate the objectives.

Returns If constraint are defined, returns the objective vector values and corresponding constraint values. Or, if no constraints are defined, returns just the objective vector values with None as the constraint values.

Return type Tuple[*np.ndarray*, Union[None, *np.ndarray*]]

Raises

- **ProblemError** – The *decision_vectors* have wrong dimensions.
- **ValueError** – If *decision_vectors* violate the lower or upper bounds.

evaluate_constraint_values(*decision_vectors*, *objective_vectors*)

Evaluate constraint values

Evaluate just the constraint function values using the attributes *decision_vectors* and *objective_vectors*

Parameters

- **decision_vectors** (*np.ndarray*) – An 2D array of decision variable input vectors. Each column represent the values of each decision variable.
- **use_surrogate** (*bool*) – A bool to control whether to use the true, potentially expensive function or a surrogate model to evaluate the objectives.

Returns

if there are constraints, then this returns *np.ndarray* of constraint values, else returns *None*

Return type *Optional[np.ndarray]*

Raises **NotImplementedError** –

Note: Currently not supported by *ScalarMOPProblem*

evaluate_fitness(*objective_vectors*)

Evaluate fitness of the objectives.

Parameters **objective_vectors** (*np.ndarray*) – objective vector array

Returns fitness of each objective vector

Return type *np.ndarray*

evaluate_objectives(*decision_vectors*, *use_surrogate=False*)

Evaluate objective values of the problem

Parameters

- **decision_vectors** (*np.ndarray*) – An 2D array of decision variable input vectors. Each column represent the values of each decision variable.
- **use_surrogate** (*bool*) – A bool to control whether to use the true, potentially expensive function or a surrogate model to evaluate the objectives.

Returns Objective vector values with their uncertainty.

Return type *Tuple[np.ndarray]*

get_objective_names()

Get objective names. Return the names of the objectives present in the problem in the order they were added.

Returns Names of the objectives in the order they were added.

Return type *List[str]*

get_variable_bounds()

Get variable bounds. Return the upper and lower bounds of each decision variable present in the problem as a 2D numpy array. The first column corresponds to the lower bounds of each variable, and the second column to the upper bound.

Returns Lower and upper bounds of each variable as a 2D numpy array. If undefined variables, return *None* instead.

Return type np.ndarray

get_variable_lower_bounds()

Get variable lower bounds.

Return the lower bounds of each variable as a list. The order of the bounds follows the order the variables were added to the problem.

Returns An array with the lower bounds of the variables.

Return type np.ndarray

get_variable_names()

Get variable names. Return the variable names of the variables present in the problem in the order they were added.

Returns Names of the variables in the order they were added.

Return type List[str]

get_variable_upper_bounds()

Get variable upper bounds.

Return the upper bounds of each variable as a list. The order of the bounds follows the order the variables were added to the problem.

Returns An array with the upper bounds of the variables.

Return type np.ndarray

static number_of_objectives(obj_instance)

Return the number of objectives in the given obj_instance.

Parameters **obj_instance** (*Union[ScalarObjective, VectorObjective]*) – An instance of one of the objective classes

Raises *ProblemError* – Raised when obj_instance is not an instance of the supported classes

Returns Number of objectives in obj_instance

Return type int

update_ideal(objective_vectors, fitness)

Update the ideal vector

Parameters

- **objective_vectors** (*np.ndarray*) – Objective vectors
- **fitness** (*np.ndarray*) – fitness of objective vectors.

DataProblem

```
class desdeo_problem.problem.DataProblem(data, variable_names, objective_names, bounds=None,
                                         maximize=None, objectives=None, variables=None,
                                         constraints=None, nadir=None, ideal=None)
```

Bases: *desdeo_problem.problem.Problem.MOProblem*

A class for a data based problem.

A problem class for data-based problem. This supports surrogate modelling. Data should be given in the form of a pandas dataframe.

Parameters

- **data** (*pd.DataFrame*) – The input data. This will be used for training the model.
- **variable_names** (*List[str]*) – Names of the variables in the dataframe provided.
- **objective_names** (*List[str]*) – Names of the objectives in the dataframe provided.
- **bounds** (*pd.DataFrame, optional*) – A pandas DataFrame containing the upper and lower bounds of the decision variables. Column names have to be same as `variable_names`. Row names have to be “lower_bound” and “upper_bound”.
- **objectives** (*List[Union[ScalarDataObjective, VectorDataObjective,]], optional*) – Objective instances, currently not supported. Defaults to None.
- **variables** (*List[Variable], optional*) – Variable instances. Defaults to None. Currently not supported.
- **constraints** (*List[ScalarConstraint], optional*) – Constraint instances. Defaults to None, which means that there are no constraints.
- **nadir** (*Optional[np.ndarray], optional*) – Nadir of the problem. Defaults to None.
- **ideal** (*Optional[np.ndarray], optional*) – Ideal of the problem. Defaults to None.

Raises

- **ProblemError** – When input data is not a dataframe.
- **ProblemError** – When given objective or variable names are not in dataframe column
- **NotImplementedError** – When objective instances are passed
- **NotImplementedError** – When variable instances are passed

Methods Summary

<code>train(models[, model_parameters, index, data])</code>	Train surrogate models for all the objectives.
<code>train_one_objective(name, model, ...[, ...])</code>	Train one objective at a time, otherwise same is the train method.

Methods Documentation

train(*models, model_parameters=None, index=None, data=None*)

Train surrogate models for all the objectives.

The models should have a fit method and a predict method. The predict method should return predicted values as well as uncertainty value (even if they are none.)

Parameters

- **models** (*Union[BaseRegressor, List[BaseRegressor]]*) – The class for the surrogate modelling algorithm.
- **models_parameters** – Dict or List[Dict] The parameters for the regressors. Should be a dict if a single regressor is provided. If a list of regressors is provided, the parameters should be in a list of dicts, same length as the list of regressors(= number of objs).
- **index** (*List[int], optional*) – The indices of the samples to be used for training the surrogate model. If no values are provided, all samples are used.

- **data** (*pd.DataFrame*, *optional*) – Use this argument if some external data is to be used for training. Defaults to None.

Raises

- **ProblemError** – If VectorDataObjective is used as one of the objective
- **instances. They are not supported yet.** –

train_one_objective(*name*, *model*, *model_parameters*, *index=None*, *data=None*)

Train one objective at a time, otherwise same is the train method.

Parameters

- **name** (*str*) – Name of the objective to be trained.
- **model** (*BaseRegressor*) – The class for the surrogate modelling algorithm.
- **model_parameters** (*Dict*) – ******model_parameters is passed to the model when initialized.
- **index** (*List[int]*, *optional*) – The indices of the samples to be used for training the surrogate model. If no values are provided, all samples are used.
- **data** (*pd.DataFrame*, *optional*) – Use this argument if some external data is to be used for training. Defaults to None.

Raises

- **ProblemError** – If name is not in the list of objective names.
- **ProblemError** – If VectorDataObjective is used as one of the objective instances. They are not supported yet.

ExperimentalProblem

```
class desdeo_problem.problem.ExperimentalProblem(variable_names, objective_names,
                                                uncertainty_names, evaluators=None,
                                                dimensions_data=None, data=None,
                                                objective_functions=None, constraints=None)
```

Bases: *desdeo_problem.problem.Problem.MOProblem*

A problem class for data-based problem. This supports surrogate modelling. Data should be given in the form of a pandas dataframe. self.archive is created to save the true function evaluations and update the surrogate models if needed. self.archive updates whenever a true function evaluation happens.

Parameters

- **data** (*pd.DataFrame*) – The input data. This will be used for training the model.
- **variable_names** (*List[str]*) – Names of the variables in the dataframe provided.
- **objective_names** (*List[str]*) – Names of the objectives in the dataframe provided.
- **objectives** (*List[Union[ScalarDataObjective, VectorDataObjective,]]*, *optional*) – Objective instances, currently not supported. Defaults to None.
- **variables** (*List[Variable]*, *optional*) – Variable instances. Defaults to None. Currently not supported.
- **constraints** (*List[ScalarConstraint]*, *optional*) – Constraint instances. Defaults to None, which means that there are no constraints.

- **nadir** (*Optional*[*np.ndarray*], *optional*) – Nadir of the problem. Defaults to None.
- **ideal** (*Optional*[*np.ndarray*], *optional*) – Ideal of the problem. Defaults to None.

Raises

- **ProblemError** – When input data is not a dataframe.
- **ProblemError** – When given objective or variable names are not in dataframe column
- **NotImplementedError** – When objective instances are passed
- **NotImplementedError** – When variable instances are passed

Note: not properly implemented!

Methods Summary

<code>evaluate</code> (<i>decision_vectors</i> , <i>use_surrogate</i>)	Evaluates the problem using an ensemble of input vectors.
<code>train</code> (<i>models</i> [, <i>model_parameters</i> , <i>index</i> , <i>data</i>])	Train surrogate models for all the objectives.
<code>train_one_objective</code> (<i>name</i> , <i>model</i> , ...[, ...])	Train one objective at a time, otherwise same is the train method.

Methods Documentation

evaluate(*decision_vectors*, *use_surrogate*)

Evaluates the problem using an ensemble of input vectors.

Parameters

- **decision_vectors** (*np.ndarray*) – An 2D array of decision variable input vectors. Each column represent the values of each decision variable.
- **use_surrogate** (*bool*) – A bool to control whether to use the true, potentially expensive function or a surrogate model to evaluate the objectives.

Returns If constraint are defined, returns the objective vector values and corresponding constraint values. Or, if no constraints are defined, returns just the objective vector values with None as the constraint values.

Return type `Tuple[np.ndarray, Union[None, np.ndarray]]`

Raises

- **ProblemError** – The *decision_vectors* have wrong dimensions.
- **ValueError** – If *decision_vectors* violate the lower or upper bounds.

train(*models*, *model_parameters=None*, *index=None*, *data=None*)

Train surrogate models for all the objectives.

The models should have a fit method and a predict method. The predict method should return predicted values as well as uncertainty value (even if they are none.)

Parameters

- **models** (*Union*[*BaseRegressor*, *List*[*BaseRegressor*]]) – The class for the surrogate modelling algorithm.

- **models_parameters** – Dict or List[Dict] The parameters for the regressors. Should be a dict if a single regressor is provided. If a list of regressors is provided, the parameters should be in a list of dicts, same length as the list of regressors(= number of objs).
- **index** (*List[int], optional*) – The indices of the samples to be used for training the surrogate model. If no values are provided, all samples are used.
- **data** (*pd.DataFrame, optional*) – Use this argument if some external data is to be used for training. Defaults to None.

Raises *ProblemError* – If VectorDataObjective is used as one of the objective instances. They are not supported yet.

train_one_objective(*name, model, model_parameters, index=None, data=None*)

Train one objective at a time, otherwise same is the train method.

Parameters

- **name** (*str*) – Name of the objective to be trained.
- **model** (*BaseRegressor*) – The class for the surrogate modelling algorithm.
- **model_parameters** (*Dict*) – ****model_parameters** is passed to the model when initialized.
- **index** (*List[int], optional*) – The indices of the samples to be used for training the surrogate model. If no values are provided, all samples are used.
- **data** (*pd.DataFrame, optional*) – Use this argument if some external data is to be used for training. Defaults to None.

Raises

- *ProblemError* – If name is not in the list of objective names.
- *ProblemError* – If VectorDataObjective is used as one of the objective instances. They are not supported yet.

VariableError

exception `desdeo_problem.problem.VariableError`

Raised when an error is encountered during the handling of the Variable objects.

VariableBuilderError

exception `desdeo_problem.problem.VariableBuilderError`

Raised when an error is encountered during the handling of the Variable objects.

Variable

class `desdeo_problem.problem.Variable`(*name*, *initial_value*, *lower_bound=-inf*, *upper_bound=inf*)

Bases: `object`

Simple variable with a name, initial value and bounds.

Parameters

- **name** (*str*) – Name of the variable
- **initial_value** (*float*) – The initial value of the variable.
- **lower_bound** (*float*, *optional*) – Lower bound of the variable. Defaults to negative infinity.
- **upper_bound** (*float*, *optional*) – Upper bound of the variable. Defaults to positive infinity.

name

Name of the variable.

Type `str`

initial_value

Initial value of the variable.

Type `float`

lower_bound

Lower bound of the variable.

Type `float`

upper_bound

Upper bound of the variable.

Type `float`

current_value

The current value the variable holds.

Type `float`

Raises `VariableError` – Bounds are incorrect.

Attributes Summary

<code>current_value</code>	<code>current_value</code>
<code>initial_value</code>	<code>initial_value</code>
<code>name</code>	<code>name</code>

Methods Summary

<code>get_bounds()</code>	Return the bounds of the variables as a tuple.
---------------------------	--

Attributes Documentation

current_value

current_value

Returns The current value of the variable

Return type float

Type Property

initial_value

initial_value

Returns The initial value of the variable.

Return type float

Type Property

name

name

Returns The name of the variable.

Return type str

Type Property

Methods Documentation

get_bounds()

Return the bounds of the variables as a tuple.

Returns

A tuple consisting of (**lower_bound**, upper_bound)

Return type tuple(float, float)

DiscreteDataProblem

class desdeo_problem.problem.**DiscreteDataProblem**(*data*, *variable_names*, *objective_names*, *ideal*, *nadir*)

Bases: object

A problem class for data-based problems with discrete values.

These data values are computed representing a set of non-dominated points.

Parameters

- **data** (*pd.DataFrame*) – The input data.
- **variable_names** (*List[str]*) – Names of the variables in the dataframe provided.

- **objective_names** (*List[str]*) – Names of the objectices in the dataframe provided.
- **nadir** (*np.ndarray*) – Nadir of the problem.
- **ideal** (*np.ndarray*) – Ideal of the problem.

Methods Summary

<code>find_closest(x)</code>	Find closest point in data to x.
------------------------------	----------------------------------

Methods Documentation

`find_closest(x)`

Find closest point in data to x.

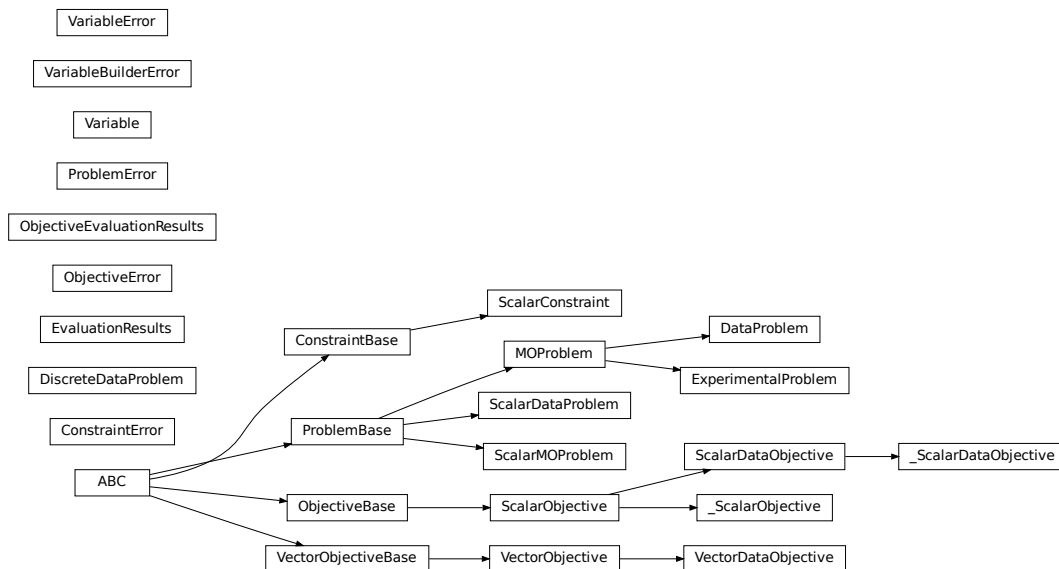
Given a vector of decision variables, finds the closest point in the given data and returns its index. A simple euclidean distance is used.

Parameters **x** (*np.ndarray*) – A 1D vector containing decision variables.

Returns The index of the closest point in the data computed for x.

Return type int

Class Inheritance Diagram



desdeo_problem.testproblems Package

Functions

<code>test_problem_builder(name[, n_of_variables, ...])</code>	Build test problems.
<code>dummy_problem()</code>	An example on how to implement a problem with 3 objectives and 4 variables and no constraints.

test_problem_builder

`desdeo_problem.testproblems.test_problem_builder(name, n_of_variables=None, n_of_objectives=None)`

Build test problems. Currently supported: ZDT1-4, ZDT6, and DTLZ1-7.

Parameters

- **name** (*str*) – Name of the problem in all caps. For example: “ZDT1”, “DTLZ4”, etc.
- **n_of_variables** (*int, optional*) – Number of variables. Required for DTLZ problems, but can be skipped for ZDT problems as they only support one variable value.
- **n_of_objectives** (*int, optional*) – Required for DTLZ problems, but can be skipped for ZDT problems as they only support one variable value.

Raises *ProblemError* – When one of many issues occur while building the MOProblem instance.

Returns The test problem object

Return type *MOProblem*

dummy_problem

`desdeo_problem.testproblems.dummy_problem()`

An example on how to implement a problem with 3 objectives and 4 variables and no constraints.

Returns a problem object.

Return type *MOProblem*

desdeo_problem.surrogatemodels Package

Classes

<code>ModelError</code>	Raised when an error related to the surrogate models classes is encountered.
<code>BaseRegressor()</code>	
<code>GaussianProcessRegressor(**kwargs)</code>	
<code>LipschitzianRegressor([L])</code>	

ModelError

exception `desdeo_problem.surrogatemodels.ModelError`

Raised when an error related to the surrogate models classes is encountered.

BaseRegressor

class `desdeo_problem.surrogatemodels.BaseRegressor`

Bases: `abc.ABC`

Methods Summary

fit(X, y)

predict(X)

rtype `Tuple[ndarray, ndarray]`

Methods Documentation

abstract `fit(X, y)`

abstract `predict(X)`

Return type `Tuple[ndarray, ndarray]`

GaussianProcessRegressor

class `desdeo_problem.surrogatemodels.GaussianProcessRegressor(**kwargs)`

Bases: `sklearn.gaussian_process._gpr.GaussianProcessRegressor`, `desdeo_problem.surrogatemodels.SurrogateModels.BaseRegressor`

Methods Summary

predict(X)

Predict using the Gaussian process regression model.

Methods Documentation

predict(X)

Predict using the Gaussian process regression model.

We can also predict based on an unfitted model by using the GP prior. In addition to the mean of the predictive distribution, optionally also returns its standard deviation (`return_std=True`) or covariance (`return_cov=True`). Note that at most one of the two can be requested.

Parameters

- **X** (array-like of shape $(n_samples, n_features)$ or list of object) – Query points where the GP is evaluated.
- **return_std** (*bool*, *default=False*) – If True, the standard-deviation of the predictive distribution at the query points is returned along with the mean.
- **return_cov** (*bool*, *default=False*) – If True, the covariance of the joint predictive distribution at the query points is returned along with the mean.

Returns

- **y_mean** (*ndarray of shape $(n_samples,)$ or $(n_samples, n_targets)$*) – Mean of predictive distribution a query points.
- **y_std** (*ndarray of shape $(n_samples,)$ or $(n_samples, n_targets)$, optional*) – Standard deviation of predictive distribution at query points. Only returned when *return_std* is True.
- **y_cov** (*ndarray of shape $(n_samples, n_samples)$ or $(n_samples, n_samples, n_targets)$, optional*) – Covariance of joint predictive distribution a query points. Only returned when *return_cov* is True.

LipschitzianRegressor

class desdeo_problem.surrogatemodels.LipschitzianRegressor(*L=None*)

Bases: *desdeo_problem.surrogatemodels.SurrogateModels.BaseRegressor*

Methods Summary

distance(array1, array2)

fit(X, y)

predict(X)

self_distance(arr)

Methods Documentation

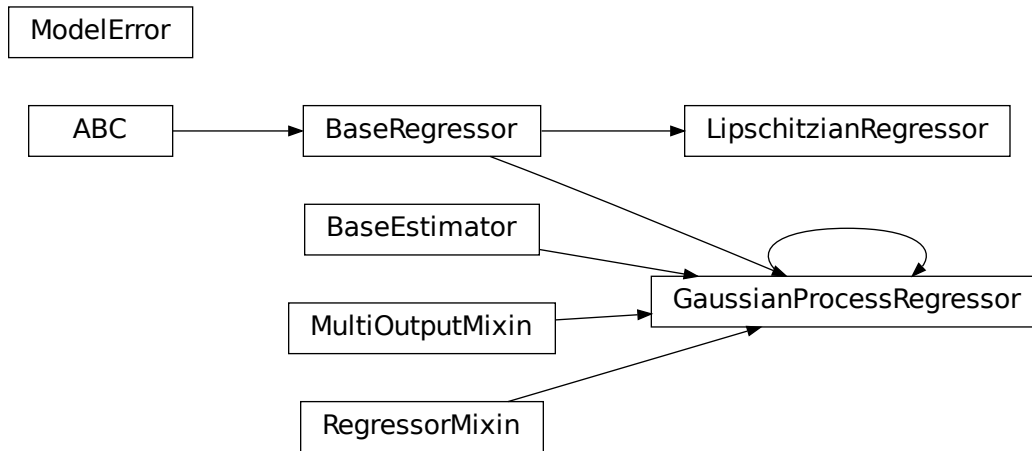
distance(array1, array2)

fit(X, y)

predict(X)

self_distance(arr)

Class Inheritance Diagram



2.2.2 Examples

Analytical problem

Defining a problem with an explicit mathematical representation is straightforward.

As an example, consider the following multiobjective optimization problem:

$$\begin{aligned}
 \min_{\mathbf{x}} \quad & x_1^2 - x_2; x_2^2 - 3x_1 \\
 \text{s.t.} \quad & x_1 + x_2 \leq 10 \\
 & \mathbf{x} \in S,
 \end{aligned} \tag{2.1}$$

where the feasible region is

$$x_i \in [-5, 5] \quad \forall i \in [1, 2]. \tag{2.2}$$

Begin by importing the necessary classes:

```
[1]: from desdeo_problem import Variable, ScalarObjective, ScalarConstraint, ScalarMOPProblem
```

Define the variables:

```
[2]: # Args: name, starting value, lower bound, upper bound
x1 = Variable("x_1", 0, -0.5, 0.5)
x2 = Variable("x_2", 0, -0.5, 0.5)
```

Define the objectives, notice the argument of the callable objective function, it is assumed to be array-like.

```
[3]: # Args: name, callable
obj1 = ScalarObjective("f_1", lambda x: x[:,0]**2 - x[:,1])
obj2 = ScalarObjective("f_2", lambda x: x[:,1]**2 - 3*x[:,0])
```


Define the constraints. Constraint may depend on objective function as well (second argument to the lambda, notice the underscore). In that case, the objectives should not be defined inline, like above, but as their own function definitions. The constraint should be defined so, that when evaluated, it should return a positive value, if the constraint is adhered to, and a negative, if the constraint is breached.

```
[4]: # Args: name, n of variables, n of objectives, callable
cons1 = ScalarConstraint("c_1", 2, 2, lambda x, _: 10 - (x[:,0] + x[:,1]))
```

Finally, put it all together and create the problem.

```
[5]: # Args: list of objctives, variables and constraints
problem = ScalarMOPProblem([obj1, obj2]
                           ,[x1, x2]
                           ,[cons1])
```

Now, the problem is fully specified and can be evaluated and played around with.

```
[6]: import numpy as np

print("N of objectives:", problem.n_of_objectives)
print("N of variables:", problem.n_of_variables)
print("N of constraints:", problem.n_of_constraints)

res1 = problem.evaluate(np.array([2, 4]))
res2 = problem.evaluate(np.array([6, 6]))
res3 = problem.evaluate(np.array([[6, 3], [4,3], [7,4]]))

print("Single feasible decision variables:", res1.objectives, "with constraint values", res1.constraints)
print("Single non-feasible decision variables:", res2.objectives, "with constraint values", res2.constraints)
print("Multiple decision variables:", res3.objectives, "with constraint values", res3.constraints)
```

```
N of objectives: 2
N of variables: 2
N of constraints: 1
Single feasible decision variables: [[ 0. 10.]] with constraint values [[4.]]
Single non-feasible decision variables: [[30. 18.]] with constraint values [[-2.]]
Multiple decision variables: [[33. -9.]
 [13. -3.]
 [45. -5.]] with constraint values [[ 1.]
 [ 3.]
 [-1.]]
```

```
[ ]:
```

The Problem class

An analytical problem is a problem where the mathematical formulation of the various objectives is known, as opposed to a data-driven problem, where one may need to train surrogate models to proceed with optimization.

The `Problem` class is the way to define optimization problems in the DESDEO framework. Once defined, the same `Problem` class instance can be used to solve optimization problems using various EAs from the `desdeo-emo` package, or the more traditional methods from the `desdeo-mcdm` package.

This notebook will help you understand how to instantiate a analytical problem object from scratch. The notebook will also go over other abstractions, namely classes for defining the decision variables, objectives, and the constraints, and will go over the functionalities provided by the abstractions.

Multiobjective Optimization Problem

Let's say that we have the following minimization problem:

$$\begin{aligned}
 \min_{\mathbf{x}} \quad & y_1, y_2, y_3 \\
 & y_1 = x_1 + x_2 + x_3 \\
 & y_2 = x_1 * x_2 * x_3 \\
 & y_3 = x_1 * x_2 + x_3 \\
 \text{s.t.} \quad & -2 \leq x_1 \leq 5 \\
 & -1 \leq x_2 \leq 10 \\
 & -0 \leq x_3 \leq 3 \\
 & x_1 + x_2 + x_3 \leq 10 \\
 & \mathbf{x} \in S,
 \end{aligned} \tag{2.3}$$

Variables

Before instantiating the problem instance, we have to create object to define each of the variables, objectives, and constraints.

The variable objects can be created with the `desdeo_problem.Variable.Variable` class. This object stores the information related to the variable (such as, lower bound, upper bound, and an initial value). This information is used by the methods whenever required (such as when setting box constraints on searching algorithms or recombination operators) and for displaying results to the decision maker. Use this class to create variable objects, one variable at a time.

To define multiple `Variable` instances easily, use the `desdeo_problem.Variable.variable_builder` function. The function takes in all the necessary information for all the variables at once, and returns a `List` of `Variable` instances, one for each decision variable.

Use the `help()` function to know more about any function/class in the `desdeo` framework.

```
[1]: from desdeo_problem import variable_builder
```

```
help(variable_builder)
```

```
Help on function variable_builder in module desdeo_problem.problem.Variable:
```

```
variable_builder(names: List[str], initial_values: Union[List[float], numpy.ndarray],
↳ lower_bounds: Union[List[float], numpy.ndarray] = None, upper_bounds:
↳ Union[List[float], numpy.ndarray] = None) -> List[desdeo_problem.problem.Variable.
↳ Variable]
(continues on next page)
```

(continued from previous page)

Automatically build all variable objects.

Args:

`names` (List[str]): Names of the variables
`initial_values` (np.ndarray): Initial values taken by the variables.
`lower_bounds` (Union[List[float], np.ndarray], optional): Lower bounds of the variables. If None, it defaults to negative infinity. Defaults to None.
`upper_bounds` (Union[List[float], np.ndarray], optional): Upper bounds of the variables. If None, it defaults to positive infinity. Defaults to None.

Raises:

`VariableError`: Lengths of the input arrays are different.

Returns:

List[Variable]: List of variable objects

Let's build the Variable objects

```
[2]: var_names = ["a", "b", "c"] # Make sure that the variable names are meaningful to you.
```

```
initial_values = [1, 1, 1]
lower_bounds = [-2, -1, 0]
upper_bounds = [5, 10, 3]
```

```
variables = variable_builder(var_names, initial_values, lower_bounds, upper_bounds)
```

```
[3]: print("Type of \"variables\": ", type(variables))
print("Length of \"variables\": ", len(variables))
print("Type of the contents of \"variables\": ", type(variables[0]))
```

```
Type of "variables": <class 'list'>
```

```
Length of "variables": 3
```

```
Type of the contents of "variables": <class 'desdeo_problem.problem.Variable.Variable'>
```

Objectives

Objectives are defined using the various objective classes found within the module `desdeo_problem.Objective`. To define an objective class instance, one needs to pass the following:

1. Objective name/s (Required): Name of the objective (or list of names, for multiple objective). This information will be used when displaying results to the user. Hence, these names must be understandable to the user.
2. Evaluator (Required for analytical/simulation based objectives): An evaluator is a python Callable which takes in the decision variables as its input and returns the corresponding objective values. This python function can be used to connect to simulators outside the DESDEO framework.
3. Lower bound (Not required): A lower bound for the objective. This information can be used to generate approximate ideal/nadir point during optimization.
4. Upper bound (Not required): An upper bound for the objective. This information can be used to generate approximate ideal/nadir point during optimization.

5. maximize (Not required): This is a boolean value that determines whether an objective is to be maximized or minimized. This is False by default (i.e. the objective is minimized).

The DESDEO framework has the following classification for objectives, based on the kind of evaluator to be used:

1. “Scalar” objectives: If an evaluator/simulator evaluates only one objective, the objective is defined as a Scalar objective. Use the `desdeo_problem.Objective._ScalarObjective` class to handle such cases.
2. “Vector” objectives: If an evaluator evaluates and returns more than one objective at once, the set of objectives is defined as Vector objective. Use the `desdeo_problem.Objective.VectorObjective` class to handle such cases.

Note: `_ScalarObjective` will be deprecated in the future, and all of its functionality will be handled by the `VectorObjective` class, which will be renamed to, simply, `Objective`.

To define a problem instance, the objectives may be defined as all Scalar objectives, all Vector objectives, or a mix of both, depending upon the case.

Let’s see how to define and use both kinds of Objective classes:

```
[4]: from desdeo_problem import ScalarObjective, VectorObjective

import numpy as np
```

Define the evaluators for the objectives. These evaluators should be python functions that take in the decision variable values and give out the objective value/s. The arguments of these evaluators are **2-D Numpy arrays**.

```
[5]: def obj1_2(x): # This is a "simulator" that returns more than one objective at a time.
↳ Hence, use VectorObjective
    y1 = x[:, 0] + x[:, 1] + x[:, 2]
    y2 = x[:, 0] * x[:, 1] * x[:, 2]
    return (y1, y2)

def obj3(x): # This is a "simulator" that returns only one objective at a time. Hence,
↳ use ScalarObjective
    y3 = x[:, 0] * x[:, 1] + x[:, 2]
    return y3
```

Define the objectives. For this, you need the names of the objectives, and the evaluators defined above. If an evaluator returns multiple objective values, use the `VectorObjective` class to define those objectives. If an evaluator returns objective values for only one objective, either `VectorObjective` or `ScalarObjective` can be used.

If using `VectorObjective`, names should be provided in a list.

Additionally, bounds of the objective values can also be provided.

```
[6]: f1_2 = VectorObjective(["y1", "y2"], obj1_2)
f3 = ScalarObjective("y3", obj3, maximize=True) # Note: f3 = VectorObjective(["y3"],
↳ obj3) will also work.
```

Constraints

Constraint may depend on the decision variable values, as well as the objective function.

The constraint should be defined so, that when evaluated, it should return a positive value, if the constraint is adhered to, and a negative, if the constraint is breached.

```
[7]: from desdeo_problem import ScalarConstraint

const_func = lambda x, y: 10 - (x[:, 0] + x[:, 1] + x[:, 2])

# Args: name, number of variables, number of objectives, callable

cons1 = ScalarConstraint("c_1", 3, 3, const_func)
```

Creating the Problem object

Now that we have all the building blocks, we can create the problem object, using the `desdeo_problem.Problem.MOProblem` class.

Provide objectives, variables and constraints in lists.

```
[8]: from desdeo_problem import MOProblem

prob = MOProblem(objectives=[f1_2, f3], variables=variables, constraints=[cons1])
```

The problem class provides abstractions such as the `evaluate` method. The method evaluates all the objective and constraint values for a given set of decision variables (in a numpy array), using the evaluators.

The abstraction also provides methods such as `train` and `surrogate_evaluate` for data driven problem. These will be tackled in the next notebook.

The output is a `NamedTuple` object. It contains the following elements:

1. `objectives`: Contains the objective values
2. `fitness`: Contains the fitness values. Fitness is either equal to the objective value, or equal to $(-1 * \text{objective value})$, depending upon whether the objective is to be minimized or maximized respectively. The optimization methods in the DESDEO framework internally use this value, rather than the values contained in `output.objectives`
3. `constraints`: Contains constraint violation values.
4. `uncertainty`: Contains the quantification of “uncertainty” of the evaluation

All of these values can be accessed in different ways, as shown below.

Note: Input as list of lists is not supported

```
[9]: data = np.asarray([[1, -1, 0], [5, 5, 2]])
res= prob.evaluate(data)
```

```
[10]: print(res)
# Note the sign reversal in the third objective and third fitness values because of
↪ maximization.
```

```
Evaluation Results Object
Objective values are:
```

(continues on next page)

(continued from previous page)

```

[[ 0. 12. -1.]
 [ 0. 50. 27.]]
Constraint violation values are:
[[10.]
 [-2.]]
Fitness values are:
[[ 0. 12.  1.]
 [ 0. 50. -27.]]
Uncertainty values are:
[[nan nan nan]
 [nan nan nan]]

```

```

[11]: print("The objective values for the given set of decision variables are: \n", res.
      ↪objectives)
      print("The constraint violation for the given set of decision variables are:\n", res.
      ↪constraints)

```

```

The objective values for the given set of decision variables are:
[[ 0. 12. -1.]
 [ 0. 50. 27.]]
The constraint violation for the given set of decision variables are:
[[10.]
 [-2.]]

```

```

[12]: res

```

```

[12]: EvaluationResults(objectives=array([[ 0., 12., -1.],
      [ 0., 50., 27.]]), fitness=array([[ 0., 12.,  1.],
      [ 0., 50., -27.]]), constraints=array([[10.],
      [-2.]]), uncertainty=array([[nan, nan, nan],
      [nan, nan, nan]]))

```

```

[ ]:

```

How to make and use the test problems

Currently supported: * ZDT Problems- ZDT1-4, ZDT6 * DTLZ Problems- DTLZ1-7

Import the test problem builder

```

[1]: from desdeo_problem.testproblems.TestProblems import test_problem_builder

```

Use `test_problem_builder` to build the necessary `MOPProblem` instance, which can be used by methods in `desdeo-emo` and `desdeo-mcdm` to solve multiobjective optimization problems

```

[2]: help(test_problem_builder)

```

```

Help on function test_problem_builder in module desdeo_problem.testproblems.TestProblems:
test_problem_builder(name: str, n_of_variables: int = None, n_of_objectives: int = None) ↪
↪-> desdeo_problem.problem.Problem.MOPProblem

```

(continues on next page)

(continued from previous page)

Build test problems. Currently supported: ZDT1-4, ZDT6, and DTLZ1-7.

Args:

`name` (str): Name of the problem in all caps. For example: "ZDT1", "DTLZ4", etc.
`n_of_variables` (int, optional): Number of variables. Required for DTLZ problems, but can be skipped for ZDT problems as they only support one variable value.
`n_of_objectives` (int, optional): Required for DTLZ problems, but can be skipped for ZDT problems as they only support one variable value.

Raises:

`ProblemError`: When one of many issues occur while building the `MOPProblem` instance.

Returns:

`MOPProblem`: The test problem object

```
[3]: zdt1 = test_problem_builder("ZDT1")
zdt1
```

```
[3]: <desdeo_problem.problem.Problem.MOPProblem at 0x1f33b279ac8>
```

```
[4]: dtlz3 = test_problem_builder("DTLZ3", n_of_objectives= 3, n_of_variables=20)
dtlz3
```

```
[4]: <desdeo_problem.problem.Problem.MOPProblem at 0x1f33b279208>
```

How to use these instances for other purposes, such as generating data:

```
[5]: import numpy as np
```

Generate input data as desired:

```
[6]: number_of_samples = 3
zdt_data = np.random.random((number_of_samples, 30)) # 30 is the number of variables in_
↳ the ZDT1 problem
print(zdt_data)
```

```
[[0.9566665  0.12642707 0.75401858 0.03765694 0.01986561 0.48660617
  0.49280935 0.85344899 0.27133411 0.93323257 0.84133493 0.20475801
  0.92905088 0.06490354 0.8570188  0.83492128 0.62833644 0.99593786
  0.81635487 0.82580931 0.56251793 0.97574662 0.47558831 0.3939823
  0.27397178 0.7496003  0.04909389 0.08239682 0.34656906 0.49915204]
 [0.74919761 0.28625722 0.67908382 0.6106337  0.6950148  0.25785019
  0.61746779 0.76319615 0.89890242 0.75963628 0.98161652 0.67291301
  0.79612155 0.52273917 0.20450823 0.7952598  0.60585745 0.4121897
  0.05809478 0.34800526 0.58840432 0.18724738 0.68237086 0.61657321
  0.3096879  0.32458604 0.16036243 0.82480997 0.17956196 0.01421743]
 [0.9957692  0.19364135 0.11009589 0.63606894 0.92162454 0.95342228
  0.88665615 0.74501953 0.09816078 0.48951933 0.76896919 0.64603171
  0.90088292 0.26154581 0.91006787 0.89883207 0.45426937 0.47012129
  0.01451065 0.40256939 0.2019439  0.45817166 0.56534801 0.18641038
  0.91780371 0.19666782 0.83473067 0.15496044 0.01478023 0.85545543]]
```

```
[7]: dtlz_data = np.random.random((number_of_samples, 20)) # We put the number of variables,
↳ earlier as 20
```

<MOProblem object>.evaluate(data) returns a tuple containing the objective values and constraint violations

```
[8]: zdt_obj_val = zdt1.evaluate(zdt_data)
zdt_obj_val
```

```
[8]: EvaluationResults(objectives=array([[0.9566665 , 3.42361516],
    [0.74919761, 3.55955498],
    [0.9957692 , 3.31853043]]), fitness=array([[0.9566665 , 3.42361516],
    [0.74919761, 3.55955498],
    [0.9957692 , 3.31853043]]), constraints=None, uncertainty=array([[nan, nan],
    [nan, nan],
    [nan, nan]]))
```

There are no constraints in the zdt or dtlz problems, hence cons_val is None

```
[9]: dtlz_obj_val = dtlz3.evaluate(dtlz_data)
dtlz_obj_val
```

```
[9]: EvaluationResults(objectives=array([[1307.49928399, 1214.1266996 , 555.10066581],
    [ 248.94975775, 162.82774202, 2199.31133262],
    [1052.58290325, 1701.71639663, 600.67385014]]), fitness=array([[1307.49928399,
↳ 1214.1266996 , 555.10066581],
    [ 248.94975775, 162.82774202, 2199.31133262],
    [1052.58290325, 1701.71639663, 600.67385014]]), constraints=None,
↳ uncertainty=array([[nan, nan, nan],
    [nan, nan, nan],
    [nan, nan, nan]]))
```

```
[ ]:
```

Data based problem

If the problem to be optimized has already been solved for a representation of its' Pareto efficient front, it can be defined as a ScalarDataProblem.

Suppose we have a problem with 2 decision variables and 4 objectives. In this case, it is the river pollution problem as defined in <https://ieeexplore.ieee.org/document/35354>

The computed Pareto efficient solutions and the corresponding objective vector values have been computed in the file 'riverpollution.dat'. There is a total of 500 entries. Begin by importing relevant classes and loading the data.

```
[2]: # NOTE: This will soon be deprecated.
from desdeo_problem import ScalarDataProblem

import numpy as np

data = np.loadtxt("./data/riverpollution.dat")
```

The first 2 entries of each row are the decision variables, and the last 4 the objective function values.

```
[3]: xs, fs = data[:, 0:2], data[:, 2:]
```


The problem can now be defined:

```
[4]: problem = ScalarDataProblem(xs, fs)
```

That's it. Now the problem is defined and can be further utilized. Notice that there are no constraints. It is assumed that all the entries in the data file are feasible. The warning has to do with the fact, that the data is discrete, therefore the evaluations for specific decision variable values have to be approximated somehow. At the moment, the closest pair of decision variables is searched for in the data. Later on, a surrogate model for the data might be build to have a better approximation.

```
[5]: print("N of objectives:", problem.n_of_objectives)
print("N of variables:", problem.n_of_variables)

print("Single decision vector evaluaion:", problem.evaluate([0.4, 0.5]))

N of objectives: 4
N of variables: 2
Single decision vector evaluaion: (array([-5.6304735 , -2.87892556, -7.06008234,  0.
↪05013393]),)
```

```
[ ]:
```

Data Based Problems

The DESDEO framework provides handling of data-driven optimization problems. Some methods, such as E-NAUTILUS in `desdeo-mcdm`, find the most preferred solution from a provided dataset. Other methods, such as most of the EA's from `desdeo-emo`, require a surrogate model to be trained for each of the objectives. The `desdeo_problem` provides support for both of these cases.

For data based problems, use the data specific objective/problem classes

```
[1]: import pandas as pd
import numpy as np
```

`VectorDataObjective` is an objective class that can handle data, as well as multi-objective evaluators.

The `GaussianProcessRegressor` here is same as the one in `scikit-learn` with one small difference. The `predict` method has been replaced to return uncertainty values (in the form of standard deviation of the prediction) by default. It supports hyperparameters in the same format as the `sklearn` method.

```
[2]: from desdeo_problem import VectorDataObjective as VDO
from desdeo_problem.surrogatemodels.SurrogateModels import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import Matern
```

Creating some random data

'a' and 'b' are randomly generated between 0 and 1.

```
f1 = a + b
f2 = a * b
```

For data-driven problems, make sure that the input dataset is in the pandas DataFrame format, with the column names being the same as the variable/objective names.

```
[3]: data = np.random.rand(100,2)

f1 = (data[:,0]+data[:,1]).reshape(-1,1)
f2 = (data[:,0]*data[:,1]).reshape(-1,1)

data = np.hstack((data, f1, f2))

X = ['a', 'b']
y = ['f1', 'f2']
datapd = pd.DataFrame(data, columns=X+y)
datapd.head()
```

```
[3]:
```

	a	b	f1	f2
0	0.730445	0.051709	0.782154	0.037771
1	0.039999	0.693915	0.733915	0.027756
2	0.024957	0.584356	0.609313	0.014584
3	0.774647	0.427847	1.202494	0.331430
4	0.500549	0.012714	0.513263	0.006364

Using VectorDataObjective class

The VectorDataObjective class takes as its input the data in a dataframe format and the objective names in a list.

```
[4]: obj = VDO(data=datapd, name=y)
```

Training surrogate models

Pass the surrogate modelling technique and the model parameters to the train method of the objective instance.

If only one modelling technique is passed, the `model_parameters` should be a dict (or None) and this will be used for all the objectives.

If multiple modelling techniques are passed, `models` should be the list of modelling techniques, and `model_parameters` should be a list of dicts. The length of these lists should be the same as the number of objectives and each list element will be used to train one objective in order.

```
[5]: obj.train(models=GaussianProcessRegressor, model_parameters={'kernel': Matern(nu=1.5)})
```

Using surrogate models to evaluate objective values

Use the `obj.evaluate` method to get predictions. Note that `use_surrogates` should be true.

```
[6]: print(obj.evaluate(np.asarray([[0.5, 0.3]]), use_surrogate=True))
```

```
Objective Evaluation Results Object
Objective values are:
      f1      f2
0  0.800011  0.150043
```

(continues on next page)

(continued from previous page)

```

Uncertainty values are:
      f1      f2
0  0.000615  0.001536

```

```
[7]: obj._model_trained
```

```
[7]: {'f1': True, 'f2': True}
```

Creating data problem class

Creating the objective class should be bypassed for now, use `DataProblem` class directly with the data in a dataframe.

The `DataProblem` provides a `train` method which trains all the objectives sequentially. The input arguments for this `train` method is the same as that of the `VectorDataObjective` class.

To make sure that the `evaluate` method uses the surrogate models for evaluations, pass the `use_surrogate=True` argument.

```
[8]: from desdeo_problem import DataProblem
```

```
[9]: maximize = pd.DataFrame([[True, False]], columns=['f1', 'f2'])
     prob = DataProblem(data=datapd, objective_names=y, variable_names=X, maximize=maximize)
```

```
[10]: prob.train(GaussianProcessRegressor)
```

```
[11]: print(prob.evaluate(np.asarray([[0.1,0.8], [0.5,0.3]]), use_surrogate=True))
```

```

Evaluation Results Object
Objective values are:
[[0.89999924  0.08000021]
 [0.8         0.14999895]]
Constraint violation values are:
None
Fitness values are:
[[-0.89999924  0.08000021]
 [-0.8         0.14999895]]
Uncertainty values are:
[[0.00346248  0.00346248]
 [0.00374312  0.00374312]]

```

Lipschitzian models

```
[12]: from desdeo_problem.surrogatemodels.lipschitzian import LipschitzianRegressor
```

```
[13]: prob = DataProblem(data=datapd, objective_names=y, variable_names=X)
```

```
[14]: prob.train(LipschitzianRegressor)
```

```
[15]: print(prob.evaluate(np.asarray([[0.1,0.8], [0.5,0.3]]), use_surrogate=True))
```

```
Evaluation Results Object
Objective values are:
[[0.9      0.08734629]
 [0.8      0.16056562]]
Constraint violation values are:
None
Fitness values are:
[[0.9      0.08734629]
 [0.8      0.16056562]]
Uncertainty values are:
[[3.33066907e-16 1.74576926e-02]
 [6.10622664e-16 7.72293212e-02]]
```

```
[ ]:
```

How to use Lipschitzian modelling

```
[1]: import numpy as np
import pandas as pd
from desdeo_problem.surrogatemodels.lipschitzian import LipschitzianRegressor
import matplotlib.pyplot as plt
```

```
[2]: def y_func(x):
      return(np.sin(x) + np.sin(2*x))
```

```
[3]: num_points = 20
x = np.linspace(0, 2 * np.pi, num_points).reshape(-1,1)
y = y_func(x)
data = pd.DataFrame(np.hstack((x,y)), columns = ['x','y'])
```

```
[4]: model = LipschitzianRegressor()
```

```
[5]: model.fit(X=data['x'], y = data['y'])
```

```
[6]: model.L
```

```
[6]: 2.8392177826759726
```

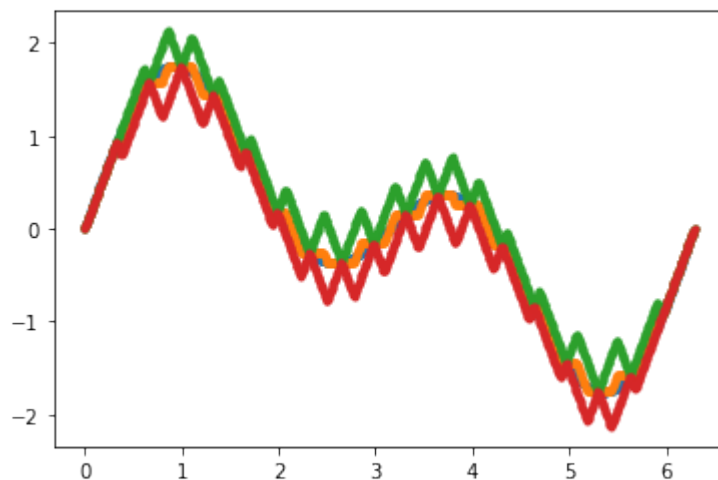
```
[7]: model.y
```

```
[7]: array([[ 0.00000000e+00],
         [ 9.38912182e-01],
         [ 1.58361298e+00],
         [ 1.75293980e+00],
         [ 1.44534766e+00],
         [ 8.31989903e-01],
         [ 1.80049416e-01],
         [-2.60860582e-01],
         [-3.61219085e-01],
         [-1.60104879e-01],
         [ 1.60104879e-01],
         [ 3.61219085e-01],
         [ 2.60860582e-01],
         [-1.80049416e-01],
         [-8.31989903e-01],
         [-1.44534766e+00],
         [-1.75293980e+00],
         [-1.58361298e+00],
         [-9.38912182e-01],
         [-7.34788079e-16]])
```

```
[8]: x_new = np.linspace(0, 2*np.pi, 1000).reshape(-1,1)
     y_new_true = y_func(x_new)
     y_mean, y_delta = model.predict(x_new)
```

```
[9]: plt.scatter(x_new, y_new_true, marker='.')
     plt.scatter(x_new, y_mean, marker='.')
     plt.scatter(x_new, y_mean + y_delta, marker='.')
     plt.scatter(x_new, y_mean - y_delta, marker='.')
```

```
[9]: <matplotlib.collections.PathCollection at 0x2c4d3b5d9c8>
```



```
[10]: def y_func2d(x):
       return(np.sin(x[:,0]) + np.sin(2*x[:,1]))
```

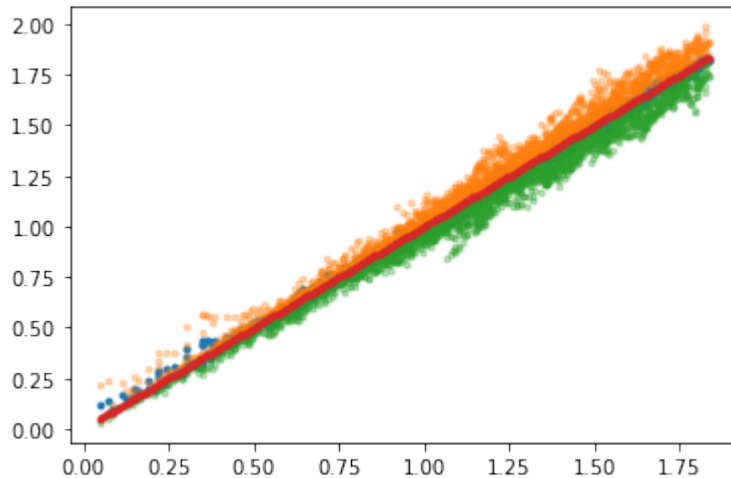
```
[11]: x = np.random.rand(200,2)
      y = y_func2d(x).reshape(-1,1)
      data = pd.DataFrame(np.hstack((x,y)), columns = ['x1', 'x2', 'y'])
```

```
[12]: model = LipschitzianRegressor()
      model.fit(X=data[['x1', 'x2']], y = data['y'])
```

```
[13]: x_new = np.random.rand(2000,2)
      y_new_true = y_func2d(x_new).reshape(-1,1)
      y_predict, y_delta = model.predict(x_new)
```

```
[14]: line = np.linspace(y_new_true.min(),y_new_true.max(),200)
      plt.scatter(y_new_true, y_predict.reshape(-1,1), marker=".")
      plt.scatter(y_new_true, (y_predict+y_delta).reshape(-1,1), marker=".", alpha=0.3)
      plt.scatter(y_new_true, (y_predict-y_delta).reshape(-1,1), marker=".", alpha=0.3)
      plt.scatter(line,line, marker='.')
```

```
[14]: <matplotlib.collections.PathCollection at 0x2c4d4c56c08>
```



```
[ ]:
```

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

`desdeo_problem.problem`, 5

`desdeo_problem.surrogatemodels`, 41

`desdeo_problem.testproblems`, 41

Symbols

<code>__ScalarDataObjective</code>	(class in <code>desdeo_problem.problem</code>), 17	<code>__n_of_objectives</code>	(<code>desdeo_problem.problem.ScalarMOPProblem</code> attribute), 22
<code>__ScalarObjective</code>	(class in <code>desdeo_problem.problem</code>), 14	<code>__n_of_variables</code>	(<code>desdeo_problem.problem.MOPProblem</code> attribute), 29
<code>__constraints</code>	(<code>desdeo_problem.problem.MOPProblem</code> attribute), 29	<code>__n_of_variables</code>	(<code>desdeo_problem.problem.ProblemBase</code> attribute), 19
<code>__constraints</code>	(<code>desdeo_problem.problem.ScalarDataProblem</code> attribute), 27	<code>__n_of_variables</code>	(<code>desdeo_problem.problem.ScalarMOPProblem</code> attribute), 22
<code>__constraints</code>	(<code>desdeo_problem.problem.ScalarMOPProblem</code> attribute), 23	<code>__nadir</code>	(<code>desdeo_problem.problem.MOPProblem</code> attribute), 29
<code>__decision_vectors</code>	(<code>desdeo_problem.problem.ProblemBase</code> attribute), 20	<code>__nadir</code>	(<code>desdeo_problem.problem.ScalarMOPProblem</code> attribute), 22
<code>__epsilon</code>	(<code>desdeo_problem.problem.ScalarDataProblem</code> attribute), 26	<code>__name</code>	(<code>desdeo_problem.problem.ScalarConstraint</code> attribute), 8
<code>__evaluator</code>	(<code>desdeo_problem.problem.ScalarConstraint</code> attribute), 8	<code>__name</code>	(<code>desdeo_problem.problem.ScalarObjective</code> attribute), 12
<code>__evaluator</code>	(<code>desdeo_problem.problem.ScalarObjective</code> attribute), 12	<code>__objective_vectors</code>	(<code>desdeo_problem.problem.ProblemBase</code> attribute), 20
<code>__ideal</code>	(<code>desdeo_problem.problem.MOPProblem</code> attribute), 29	<code>__objectives</code>	(<code>desdeo_problem.problem.MOPProblem</code> attribute), 29
<code>__ideal</code>	(<code>desdeo_problem.problem.ScalarMOPProblem</code> attribute), 23	<code>__objectives</code>	(<code>desdeo_problem.problem.ScalarMOPProblem</code> attribute), 23
<code>__lower_bound</code>	(<code>desdeo_problem.problem.ScalarObjective</code> attribute), 12	<code>__upper_bound</code>	(<code>desdeo_problem.problem.ScalarObjective</code> attribute), 12
<code>__model_exists</code>	(<code>desdeo_problem.problem.ScalarDataProblem</code> attribute), 27	<code>__value</code>	(<code>desdeo_problem.problem.ScalarObjective</code> attribute), 12
<code>__n_decision_vars</code>	(<code>desdeo_problem.problem.ScalarConstraint</code> attribute), 8	<code>__variables</code>	(<code>desdeo_problem.problem.MOPProblem</code> attribute), 29
<code>__n_objective_funs</code>	(<code>desdeo_problem.problem.ScalarConstraint</code> attribute), 8	<code>__model</code>	(<code>desdeo_problem.problem.ScalarDataObjective</code> attribute), 16
<code>__n_of_constraints</code>	(<code>desdeo_problem.problem.ScalarMOPProblem</code> attribute), 22	<code>__model</code>	(<code>desdeo_problem.problem.VectorDataObjective</code> attribute), 18
<code>__n_of_objectives</code>	(<code>desdeo_problem.problem.MOPProblem</code> attribute), 29	<code>__model_trained</code>	(<code>desdeo_problem.problem.VectorDataObjective</code> attribute), 18
<code>__n_of_objectives</code>	(<code>desdeo_problem.problem.ProblemBase</code> attribute), 19		

B

BaseRegressor (class in *desdeo_problem.surrogatemodels*), 42

C

constraint_function_factory() (in module *desdeo_problem.problem*), 6

ConstraintBase (class in *desdeo_problem.problem*), 9

ConstraintError, 9

constraints (*desdeo_problem.problem.EvaluationResults* attribute), 22

constraints (*desdeo_problem.problem.MOPProblem* attribute), 30

constraints (*desdeo_problem.problem.ScalarDataProblem* attribute), 27

constraints (*desdeo_problem.problem.ScalarMOPProblem* attribute), 24

current_value (*desdeo_problem.problem.Variable* attribute), 38, 39

D

DataProblem (class in *desdeo_problem.problem*), 33

decision_vectors (*desdeo_problem.problem.ProblemBase* attribute), 20

decision_vectors (*desdeo_problem.problem.ScalarDataProblem* attribute), 26

desdeo_problem.problem module, 5

desdeo_problem.surrogatemodels module, 41

desdeo_problem.testproblems module, 41

DiscreteDataProblem (class in *desdeo_problem.problem*), 39

distance() (*desdeo_problem.surrogatemodels.LipschitzianRegressor* method), 43

dummy_problem() (in module *desdeo_problem.testproblems*), 41

E

epsilon (*desdeo_problem.problem.ScalarDataProblem* attribute), 27

evaluate() (*desdeo_problem.problem.ConstraintBase* method), 10

evaluate() (*desdeo_problem.problem.ExperimentalProblem* method), 36

evaluate() (*desdeo_problem.problem.MOPProblem* method), 31

evaluate() (*desdeo_problem.problem.ObjectiveBase* method), 10

evaluate() (*desdeo_problem.problem.ProblemBase* method), 21

evaluate() (*desdeo_problem.problem.ScalarConstraint* method), 9

evaluate() (*desdeo_problem.problem.ScalarDataProblem* method), 28

evaluate() (*desdeo_problem.problem.ScalarMOPProblem* method), 25

evaluate() (*desdeo_problem.problem.VectorObjectiveBase* method), 11

evaluate_constraint_values() (*desdeo_problem.problem.MOPProblem* method), 31

evaluate_constraint_values() (*desdeo_problem.problem.ProblemBase* method), 21

evaluate_constraint_values() (*desdeo_problem.problem.ScalarDataProblem* method), 28

evaluate_constraint_values() (*desdeo_problem.problem.ScalarMOPProblem* method), 25

evaluate_fitness() (*desdeo_problem.problem.MOPProblem* method), 32

evaluate_objectives() (*desdeo_problem.problem.MOPProblem* method), 32

EvaluationResults (class in *desdeo_problem.problem*), 21

evaluator (*desdeo_problem.problem.ScalarConstraint* attribute), 8

evaluator (*desdeo_problem.problem.ScalarObjective* attribute), 13

evaluator (*desdeo_problem.problem.VectorObjective* attribute), 15

ExperimentalProblem (class in *desdeo_problem.problem*), 35

F

find_closest() (*desdeo_problem.problem.DiscreteDataProblem* method), 40

fit() (*desdeo_problem.surrogatemodels.BaseRegressor* method), 42

fit() (*desdeo_problem.surrogatemodels.LipschitzianRegressor* method), 43

fitness (*desdeo_problem.problem.EvaluationResults* attribute), 22

G

GaussianProcessRegressor (class in *desdeo_problem.surrogatemodels*), 42

get_bounds() (*desdeo_problem.problem.Variable* method), 39

get_objective_names() (*desdeo_problem.problem.MOPProblem* method), 32
 get_objective_names() (*desdeo_problem.problem.ScalarMOPProblem* method), 25
 get_uncertainty_names() (*desdeo_problem.problem.ScalarMOPProblem* method), 25
 get_variable_bounds() (*desdeo_problem.problem.MOPProblem* method), 32
 get_variable_bounds() (*desdeo_problem.problem.ProblemBase* method), 21
 get_variable_bounds() (*desdeo_problem.problem.ScalarDataProblem* method), 28
 get_variable_bounds() (*desdeo_problem.problem.ScalarMOPProblem* method), 25
 get_variable_lower_bounds() (*desdeo_problem.problem.MOPProblem* method), 33
 get_variable_lower_bounds() (*desdeo_problem.problem.ScalarMOPProblem* method), 26
 get_variable_names() (*desdeo_problem.problem.MOPProblem* method), 33
 get_variable_names() (*desdeo_problem.problem.ScalarMOPProblem* method), 26
 get_variable_upper_bounds() (*desdeo_problem.problem.MOPProblem* method), 33
 get_variable_upper_bounds() (*desdeo_problem.problem.ScalarMOPProblem* method), 26

I

ideal (*desdeo_problem.problem.ProblemBase* attribute), 19
 ideal (*desdeo_problem.problem.ScalarDataProblem* attribute), 27
 ideal (*desdeo_problem.problem.ScalarMOPProblem* attribute), 24
 ideal_fitness (*desdeo_problem.problem.ProblemBase* attribute), 19
 initial_value (*desdeo_problem.problem.Variable* attribute), 38, 39

L

LipschitzianRegressor (*class in desdeo_problem.problem.SurrogateModels*), 43

lower_bound (*desdeo_problem.problem.ScalarObjective* attribute), 13
 lower_bound (*desdeo_problem.problem.Variable* attribute), 38
 lower_bounds (*desdeo_problem.problem.VectorObjective* attribute), 15

M

maximize (*desdeo_problem.problem.ScalarObjective* attribute), 12
 ModelError, 42
 module
 desdeo_problem.problem, 5
 desdeo_problem.surrogatemodels, 41
 desdeo_problem.testproblems, 41
 MOPProblem (*class in desdeo_problem.problem*), 29

N

n_decision_vars (*desdeo_problem.problem.ScalarConstraint* attribute), 8
 n_objective_funs (*desdeo_problem.problem.ScalarConstraint* attribute), 8
 n_of_constraints (*desdeo_problem.problem.MOPProblem* attribute), 30
 n_of_constraints (*desdeo_problem.problem.ScalarMOPProblem* attribute), 24
 n_of_fitnesses (*desdeo_problem.problem.MOPProblem* attribute), 30
 n_of_objectives (*desdeo_problem.problem.MOPProblem* attribute), 31
 n_of_objectives (*desdeo_problem.problem.ProblemBase* attribute), 20
 n_of_objectives (*desdeo_problem.problem.ScalarMOPProblem* attribute), 24
 n_of_objectives (*desdeo_problem.problem.VectorObjective* attribute), 15
 n_of_variables (*desdeo_problem.problem.MOPProblem* attribute), 31
 n_of_variables (*desdeo_problem.problem.ProblemBase* attribute), 20
 n_of_variables (*desdeo_problem.problem.ScalarMOPProblem* attribute), 26

- attribute*), 24
 nadir (*desdeo_problem.problem.ProblemBase attribute*), 19
 nadir (*desdeo_problem.problem.ScalarDataProblem attribute*), 27
 nadir (*desdeo_problem.problem.ScalarMOPProblem attribute*), 24
 nadir_fitness (*desdeo_problem.problem.ProblemBase attribute*), 19
 name (*desdeo_problem.problem.ScalarConstraint attribute*), 9
 name (*desdeo_problem.problem.ScalarObjective attribute*), 13
 name (*desdeo_problem.problem.Variable attribute*), 38, 39
 name (*desdeo_problem.problem.VectorObjective attribute*), 15
 number_of_objectives() (*desdeo_problem.problem.MOPProblem static method*), 33
- ## O
- objective_vectors (*desdeo_problem.problem.ScalarDataProblem attribute*), 26
 ObjectiveBase (*class in desdeo_problem.problem*), 10
 ObjectiveError, 11
 ObjectiveEvaluationResults (*class in desdeo_problem.problem*), 11
 objectives (*desdeo_problem.problem.EvaluationResults attribute*), 21
 objectives (*desdeo_problem.problem.MOPProblem attribute*), 31
 objectives (*desdeo_problem.problem.ObjectiveEvaluationResults attribute*), 11
 objectives (*desdeo_problem.problem.ScalarMOPProblem attribute*), 24
- ## P
- predict() (*desdeo_problem.surrogatemodels.BaseRegressor method*), 42
 predict() (*desdeo_problem.surrogatemodels.GaussianProcessRegressor method*), 42
 predict() (*desdeo_problem.surrogatemodels.LipschitzianRegressor method*), 43
 ProblemBase (*class in desdeo_problem.problem*), 19
 ProblemError, 19
- ## S
- ScalarConstraint (*class in desdeo_problem.problem*), 7
 ScalarDataObjective (*class in desdeo_problem.problem*), 16
 ScalarDataProblem (*class in desdeo_problem.problem*), 26
 ScalarMOPProblem (*class in desdeo_problem.problem*), 22
 ScalarObjective (*class in desdeo_problem.problem*), 12
 self_distance() (*desdeo_problem.surrogatemodels.LipschitzianRegressor method*), 43
- ## T
- test_problem_builder() (*in module desdeo_problem.testproblems*), 41
 train() (*desdeo_problem.problem.DataProblem method*), 34
 train() (*desdeo_problem.problem.ExperimentalProblem method*), 36
 train() (*desdeo_problem.problem.ScalarDataObjective method*), 17
 train() (*desdeo_problem.problem.VectorDataObjective method*), 18
 train_one_objective() (*desdeo_problem.problem.DataProblem method*), 35
 train_one_objective() (*desdeo_problem.problem.ExperimentalProblem method*), 37
- ## U
- uncertainty (*desdeo_problem.problem.EvaluationResults attribute*), 22
 uncertainty (*desdeo_problem.problem.ObjectiveEvaluationResults attribute*), 11
 update_ideal() (*desdeo_problem.problem.MOPProblem method*), 33
 upper_bound (*desdeo_problem.problem.ScalarObjective attribute*), 13
 upper_bound (*desdeo_problem.problem.Variable attribute*), 38
 upper_bounds (*desdeo_problem.problem.VectorObjective attribute*), 15
- ## V
- value (*desdeo_problem.problem.ScalarObjective attribute*), 13
 values (*desdeo_problem.problem.VectorObjective attribute*), 15
 Variable (*class in desdeo_problem.problem*), 38
 variable_builder() (*in module desdeo_problem.problem*), 6
 variable_names (*desdeo_problem.problem.ScalarDataObjective attribute*), 16

variable_names (desdeo_problem.problem.VectorDataObjective attribute), 17

VariableBuilderError, 37

VariableError, 37

variables (desdeo_problem.problem.MOProblem attribute), 31

variables (desdeo_problem.problem.ScalarMOProblem attribute), 24

VectorDataObjective (class in desdeo_problem.problem), 17

VectorObjective (class in desdeo_problem.problem), 14

VectorObjectiveBase (class in desdeo_problem.problem), 11

X

X (desdeo_problem.problem.ScalarDataObjective attribute), 16

X (desdeo_problem.problem.VectorDataObjective attribute), 17

Y

y (desdeo_problem.problem.ScalarDataObjective attribute), 16

y (desdeo_problem.problem.VectorDataObjective attribute), 17