
EMA workbench Documentation

Release 2.4.0

J.H. Kwakkel

Apr 07, 2023

GETTING STARTED

1	Exploratory Modelling and Analysis (EMA) Workbench	1
	Python Module Index	205
	Index	207

EXPLORATORY MODELLING AND ANALYSIS (EMA) WORKBENCH

Exploratory Modeling and Analysis (EMA) is a research methodology that uses computational experiments to analyze complex and uncertain systems (Bankes, 1993). That is, exploratory modeling aims at offering computational decision support for decision making under [deep uncertainty](#) and [Robust Decision Making](#).

The EMA workbench aims at providing support for performing exploratory modeling with models developed in various modelling packages and environments. Currently, the workbench offers connectors to [Vensim](#), [Netlogo](#), and Excel.

The EMA workbench offers support for designing experiments, performing the experiments - including support for parallel processing on both a single machine as well as on clusters-, and analysing the results. To get started, take a look at the high level overview, the tutorial, or dive straight into the details of the API. For a comparison between the workbench and [rhodium](#), see [this discussion](#).

1.1 A High Level Overview

- *Exploratory modeling framework*
- *Connectors*
- *Analysis*

1.1.1 Exploratory modeling framework

The core package contains the core functionality for setting up, designing, and performing series of computational experiments on one or more models simultaneously.

- Model (`ema_workbench.em_framework.model`): an abstract base class for specifying the interface to the model on which you want to perform exploratory modeling.
- Samplers (`ema_workbench.em_framework.samplers`): the various sampling techniques that are readily available in the workbench.
- Uncertainties (`ema_workbench.em_framework.parameters`): various types of parameter classes that can be used to specify the uncertainties and/or levers on the model
- Outcomes (`ema_workbench.em_framework.outcomes`): various types of outcome classes
- Evaluators (`ema_workbench.em_framework.evaluators`): various evaluators for running experiments in sequence or in parallel.

1.1.2 Connectors

The connectors package contains connectors to some existing simulation modeling environments. For each of these, a standard `ModelStructureInterface` class is provided that users can use as a starting point for specifying the interface to their own model.

- Vensim connector (`ema_workbench.connectors.vensim`): This enables controlling (e.g. setting parameters, simulation setup, run, get output, etc .) a simulation model that is built in Vensim software, and conducting an EMA study based on this model.
- Excel connector (`ema_workbench.connectors.excel`): This enables controlling models build in Excel.
- NetLogo connector (`ema_workbench.connectors.netlogo`): This enables controlling (e.g. setting parameters, simulation setup, run, get output, etc .) a simulation model that is built in NetLogo software, and conducting an EMA study based on this model.
- Simio connector (`ema_workbench.connectors.simio_connector`): This enables controlling models built in Simio
- Pysd connector (`ema_workbench.connectors.pysd_connector`)

1.1.3 Analysis

The analysis package contains a variety of analysis and visualization techniques for analyzing the results from the exploratory modeling. The analysis scripts are tailored for use in combination with the workbench, but they can also be used on their own with data generated in some other manner.

- Patient Rule Induction Method (`ema_workbench.analysis.prim`)
- Classification Trees (`ema_workbench.analysis.cart`)
- Logistic Regression (`ema_workbench.analysis.logistic_regression`)
- Dimensional Stacking (`ema_workbench.analysis.dimensional_stacking`)
- Feature Scoring (`ema_workbench.analysis.feature_scoring`)
- Regional Sensitivity Analysis (`ema_workbench.analysis.regional_sa`)
- various plotting functions for time series data (`ema_workbench.analysis.plotting`)
- pair wise plots (`ema_workbench.analysis.pairs_plotting`)
- parallel coordinate plots (`ema_workbench.analysis.parcoords`)
- support for converting figures to black and white (`ema_workbench.analysis.b_an_w_plotting`)

1.2 Installing the workbench

The 2.x version of the workbench requires Python 3.8 or higher.

1.2.1 Regular installations

A stable version of the workbench can be installed via pip.

```
pip install ema_workbench
```

This installs the EMAworkbench together with all the bare necessities to run Python models.

If you want to upgrade the workbench from a previous version, add `-U` (or `--upgrade`) to the pip command.

```
pip install -U ema_workbench
```

We have a few more install options available, which install optional dependencies not always necessary but either nice to have or for specific functions. These can be installed with so called “extras” using pip.

Therefore we recommended installing with:

```
pip install -U ema_workbench[recommended]
```

Which currently includes everything needed to use the workbench in Jupyter notebooks, with interactive graphs and to successfully complete the tests with pytest.

However, the EMAworkbench can connect to many other simulation software, such as Netlogo, Simio, Vensim (pysd) and Vadere. For these there are also extras available:

```
pip install -U ema_workbench[netlogo,simio,pysd]
```

Note that the Netlogo and Simio extras need Windows as OS.

These extras can be combined. If you’re going to work with Netlogo for example, you can do:

```
pip install -U ema_workbench[recommended,netlogo]
```

You can use `all` to install all dependencies, except the connectors. Prepare for a large install.

```
pip install -U ema_workbench[all]
```

These are all the extras that are available:

- `jupyter` installs ["jupyter", "jupyter_client", "ipython", "ipykernel"]
- `dev` installs ["pytest", "jupyter_client", "ipyparallel"]
- `cov` installs ["pytest-cov", "coverage", "coveralls"]
- `docs` installs ["sphinx", "nbsphinx", "myst", "pyscaffold"]
- `graph` installs ["altair", "pydot", "graphviz"]
- `parallel` installs ["ipyparallel", "traitlets"]
- `netlogo` installs ["jpype-1", "pynetlogo"]
- `pysd` installs ["pysd"]
- `simio` installs ["pythonnet"]

Then `recommended` is currently equivalent to `jupyter`, `dev`, `graph` and `all` installs everything, except the connectors. These are defined in the `pyproject.toml` file.

1.2.2 Developer installations

As a developer you will want an edible install, in which you can modify the installation itself. This can be done by adding `-e` (for edible) to the pip command.

```
pip install -e ema_workbench
```

The latest commit on the master branch can be installed with:

```
pip install -e git+https://github.com/quaquel/EMAworkbench#egg=ema-workbench
```

Or any other (development) branch on this repo or your own fork:

```
pip install -e git+https://github.com/YOUR_FORK/EMAworkbench@YOUR_BRANCH#egg=ema-  
↪workbench
```

The code is also available from [github](#).

1.2.3 Limitations

Some connectors have specific limitations, listed below.

- Vensim only works on Windows. If you have 64-bit Vensim, you need 64-bit Python. If you have 32-bit Vensim, you will need 32-bit Python.
- Excel also only works on Windows.

1.3 Alternative packages

So how does the workbench differ from other open source tools available for exploratory modeling? In Python there is [rhodium](#), in R there is the [open MORDM](#) toolkit, and there is also [OpenMole](#). Below we discuss the key differences with rhodium. Given a lack of familiarity with the other tools, we wont comment on those.

1.3.1 The workbench versus rhodium

For Python, the main alternative tool is [rhodium](#), which is part of [Project Platypus](#). Project Platypus is a collection of libraries for doing many objective optimization ([platypus-opt](#)), setting up and performing simulation experiments ([rhodium](#)), and scenario discovery using the Patient Rule Induction Method ([prim](#)). The relationship between the workbench and the tools that form project platypus is a bit messy. For example, the workbench too relies on [platypus-opt](#) for many objective optimization, the [PRIM](#) package is a, by now very dated, fork of the prim code in the workbench, and both [rhodium](#) and the workbench rely on [SALib](#) for global sensitivity analysis. Moreover, the API of [rhodium](#) was partly inspired by an older version of the workbench, while new ideas from the rhodium API have in turned resulting in profound changes in the API of the workbench.

Currently, the workbench is still actively being developed. It is also not just used in teaching but also for research, and in practice by various organization globally. Moreover, the workbench is quite a bit more developed when it comes to providing off the shelf connectors for some popular modeling and simulation tools. Basically, everything that can be done with project Platypus can be done with the workbench and then the workbench offers additional functionality, a more up to date code base, and active support.

1.4 Changelog

All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](#), and this project adheres to [Semantic Versioning](#).

1.4.1 2.3.0

Highlights

This release adds a new algorithm for [output space exploration](#). The way in which convergence tracking for optimization is supported has been overhauled completely, see the updated [directed search](#) user guide for full details. The documentation has moreover been expanded with a [comparison to Rhodium](#).

With this new release, the installation process has been improved by reducing the number of required dependencies. Recommended packages and connectors can now be installed as *extras* using pip, for example `pip install -U ema-workbench[recommended,netlogo]`. See the [updated installation instructions](#) for all options and details.

The 2.3.x release series supports Python 3.8 to 3.11. It can be installed as usual via PyPI, with:

```
pip install --upgrade ema-workbench
```

What's Changed

New features added

- Output space exploration by @quaquel in <https://github.com/quaquel/EMAworbench/pull/170>
- Convergence tracking by @quaquel in <https://github.com/quaquel/EMAworbench/pull/193>

Enhancements made

- Switch to using format string in prim logging by @quaquel in <https://github.com/quaquel/EMAworbench/pull/161>
- Replace `setup.py` with `pyproject.toml` and implement optional dependencies by @EwoutH in <https://github.com/quaquel/EMAworbench/pull/166>

Bugs fixed

- use masked arrays for storing outcomes by @quaquel in <https://github.com/quaquel/EMAworbench/pull/176>
- Fix error for negative `n_processes` input in `MultiprocessingEvaluator` by @EwoutH in <https://github.com/quaquel/EMAworbench/pull/189>
- `optimization.py`: Fix “`epsilons`” keyword argument in `_optimize()` by @EwoutH in <https://github.com/quaquel/EMAworbench/pull/150>

Documentation improvements

- Create initial CONTRIBUTING.md documentation by @EwoutH in <https://github.com/quaquel/EMAworkbench/pull/162>
- Create Read the Docs yaml configuration by @EwoutH in <https://github.com/quaquel/EMAworkbench/pull/173>
- update to outcomes documentation by @quaquel in <https://github.com/quaquel/EMAworkbench/pull/183>
- Improved directed search tutorial by @quaquel in <https://github.com/quaquel/EMAworkbench/pull/194>
- Update Contributing.md with instructions how to merge PRs by @EwoutH in <https://github.com/quaquel/EMAworkbench/pull/200>
- Update Readme with an introduction and documentation, installation and contribution sections by @EwoutH in <https://github.com/quaquel/EMAworkbench/pull/199>
- Rhodium docs by @quaquel in <https://github.com/quaquel/EMAworkbench/pull/184>
- Fix spelling mistakes by @EwoutH in <https://github.com/quaquel/EMAworkbench/pull/195>

Maintenance

- Replace depreciated shade keyword in Seaborn kdeplot with fill by @EwoutH in <https://github.com/quaquel/EMAworkbench/pull/169>
- CI: Add pip dependency caching, don't run on doc changes, update setup-python to v4 by @EwoutH in <https://github.com/quaquel/EMAworkbench/pull/174>
- Formatting: Format with Black, increase max line length to 100, combine multi-line blocks by @EwoutH in <https://github.com/quaquel/EMAworkbench/pull/178>
- Add pre-commit configuration and auto update CI by @EwoutH in <https://github.com/quaquel/EMAworkbench/pull/181>
- Fix Matplotlib, ipyparallel and dict deprecation warnings by @EwoutH in <https://github.com/quaquel/EMAworkbench/pull/202>
- CI: Start testing on Python 3.11 by @EwoutH in <https://github.com/quaquel/EMAworkbench/pull/156>
- Replace deprecated saltelli with sobol SALib 1.4.6+. by @quaquel in <https://github.com/quaquel/EMAworkbench/pull/211>

Other changes

- Adds CITATION.cff by @quaquel in <https://github.com/quaquel/EMAworkbench/pull/209>

Full Changelog: <https://github.com/quaquel/EMAworkbench/compare/2.2.0...2.3>

1.4.2 2.2.0

Highlights

With the 2.2 release, the EMAworkbench can now connect to **Vadere** models on pedestrian dynamics. When inspecting a Prim Box peeling trajectory, multiple points on the peeling trajectory can be inspected simultaneously by inputting a list of integers into `PrimBox.inspect()`.

When running experiments with multiprocessing using the **MultiprocessingEvaluator**, the number of processes can now be controlled using a negative integer as input for `n_processes` (for example, -2 on a 12-thread CPU results

in 10 threads used). Also, it will now default to max. 61 processes on windows machines due to limitations inherent in Windows in dealing with higher processor counts. Code quality, CI, and error reporting also have been improved. And finally, generating these release notes is now automated.

What's Changed

New features added

- Vadere model connector by @floristevito in <https://github.com/quaquel/EMAworkbench/pull/145>

Enhancements made

- Improve code quality with static analysis by @EwoutH in <https://github.com/quaquel/EMAworkbench/pull/119>
- `prim.py`: Make `PrimBox.peeling_trajectory["id"]` int instead of float by @EwoutH in <https://github.com/quaquel/EMAworkbench/pull/121>
- `analysis`: Allow optional annotation of `plot_tradeoff` graphs by @EwoutH in <https://github.com/quaquel/EMAworkbench/pull/123>
- `evaluators.py`: Allow `MultiprocessingEvaluator` to initialize with `cpu_count` minus N processes by @EwoutH in <https://github.com/quaquel/EMAworkbench/pull/140>
- `PrimBox.inspect()` now can also take a list of integers (aside from a single int) to inspect multiple points at once by @quaquel in <https://github.com/quaquel/EMAworkbench/commit/6d83a6c33442ad4dce0974a384b03a225aaf830d> (see also issue <https://github.com/quaquel/EMAworkbench/issues/124>)

Bugs fixed

- fixed typo in `lake_model.py` by @JeffreyDillonLyons in <https://github.com/quaquel/EMAworkbench/pull/136>

Documentation improvements

- Docs: `Installation.rst`: Add how to install master or custom branch by @EwoutH in <https://github.com/quaquel/EMAworkbench/pull/122>
- Docs: Replace all http links with secure https URLs by @EwoutH in <https://github.com/quaquel/EMAworkbench/pull/134>
- Maintain release notes at `CHANGELOG.md` and include them in `Readthedocs` by @quaquel in <https://github.com/quaquel/EMAworkbench/commit/ebdbc9f5c77693fc75911ead472b420065dfe2aa>
- Fix badge links in readme by @quaquel in <https://github.com/quaquel/EMAworkbench/commit/28569bdc149c070c329589969179>

Maintenance

- `feature_scoring`: fix Regressor criterion depreciation by @EwoutH in <https://github.com/quaquel/EMAWorkbench/pull/125>
- `feature_scoring.py`: Change `max_features` in `get_rf_feature_scores` to `"sqrt"` by @EwoutH in <https://github.com/quaquel/EMAWorkbench/pull/129>
- CI: Use Pytest instead of Nose, update default build to Python 3.10 by @EwoutH in <https://github.com/quaquel/EMAWorkbench/pull/131>
- Release CI: Only upload packages if on main repo by @EwoutH in <https://github.com/quaquel/EMAWorkbench/pull/132>
- CI: Split off flake8 linting in a separate job by @EwoutH in <https://github.com/quaquel/EMAWorkbench/pull/133>
- CI: Add weekly scheduled jobs and manual trigger by @EwoutH in <https://github.com/quaquel/EMAWorkbench/pull/137>
- `setup.py`: Add `project_urls` for documentation and issue tracker links by @EwoutH in <https://github.com/quaquel/EMAWorkbench/pull/142>
- set scikit-learn requirement `>= 1.0.0` by @rhapsys in <https://github.com/quaquel/EMAWorkbench/pull/144>
- Create `release.yml` file for automatic release notes generation by @EwoutH in <https://github.com/quaquel/EMAWorkbench/pull/152>
- instantiating an Evaluator without one or more AbstractModel instances now raises a type error by @quaquel in <https://github.com/quaquel/EMAWorkbench/commit/a83533aa8166ca2414137cdfc3125a53ee3697ec>
- removes deprecated `DataFrame.append` by replacing it with `DataFrame.concat` (see the conversation on issue <https://github.com/quaquel/EMAWorkbench/issues/126>):
 - from `feature_scoring` by @quaquel in <https://github.com/quaquel/EMAWorkbench/commit/8b8bfe41733e49b75c01e34b75563>
 - from `logistic_regression.py` by @quaquel in <https://github.com/quaquel/EMAWorkbench/commit/255e3d6d9639dfe6fd4e797>
- removes NumPy datatypes deprecated in 1.20 by @quaquel in <https://github.com/quaquel/EMAWorkbench/commit/e8fbf6fc64f14b>
- replace deprecated `scipy.stats.kde` with `scipy.stats` by @quaquel in <https://github.com/quaquel/EMAWorkbench/commit/b5a9ca9677>

New Contributors

- @JeffreyDillonLyons made their first contribution in <https://github.com/quaquel/EMAWorkbench/pull/136>
- @rhapsys made their first contribution in <https://github.com/quaquel/EMAWorkbench/pull/144>
- @floristevito made their first contribution in <https://github.com/quaquel/EMAWorkbench/pull/145>

Full Changelog: <https://github.com/quaquel/EMAWorkbench/compare/2.1.2...2.2.0>

1.5 Tutorials

The code of these examples can be found in the examples package. The first three examples are meant to illustrate the basics of the EMA workbench. How to implement a model, specify its uncertainties and outcomes, and run it. The fourth example is a more extensive illustration based on Pruyt & Hamarat (2010). It shows some more advanced possibilities of the EMA workbench, including one way of handling policies.

- *A simple model in Python*
- *A simple model in Vensim*
- *A simple model in Excel*
- *A more elaborate example: Mexican Flu*

1.5.1 A simple model in Python

The simplest case is where we have a model available through a python function. For example, imagine we have the simple model.

```
def some_model(x1=None, x2=None, x3=None):
    return {'y': x1*x2+x3}
```

In order to control this model from the workbench, we can make use of the `Model`. We can instantiate a model object, by passing it a name, and the function.

```
model = Model('simpleModel', function=some_model) #instantiate the model
```

Next, we need to specify the uncertainties and the outcomes of the model. In this case, the uncertainties are x1, x2, and x3, while the outcome is y. Both uncertainties and outcomes are attributes of the model object, so we can say

```
1 #specify uncertainties
2 model.uncertainties = [RealParameter("x1", 0.1, 10),
3                        RealParameter("x2", -0.01,0.01),
4                        RealParameter("x3", -0.01,0.01)]
5 #specify outcomes
6 model.outcomes = [ScalarOutcome('y')]
```

Here, we specify that x1 is some value between 0.1, and 10, while both x2 and x3 are somewhere between -0.01 and 0.01. Having implemented this model, we can now investigate the model behavior over the set of uncertainties by simply calling

```
results = perform_experiments(model, 100)
```

The function `perform_experiments()` takes the model we just specified and will execute 100 experiments. By default, these experiments are generated using a Latin Hypercube sampling, but Monte Carlo sampling and Full factorial sampling are also readily available. Read the documentation for `perform_experiments()` for more details.

The complete code:

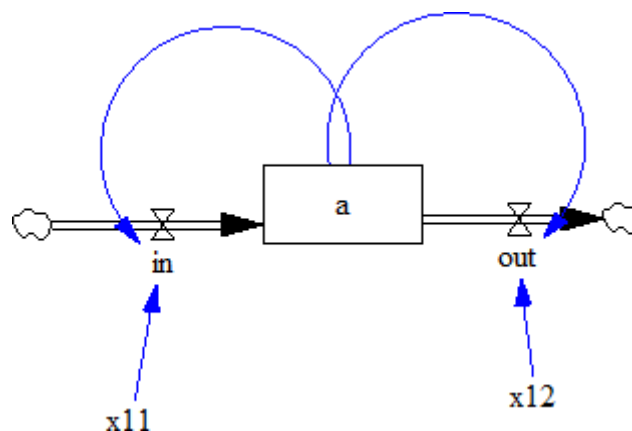
```

1  """
2  Created on 20 dec. 2010
3
4  This file illustrated the use the EMA classes for a contrived example
5  It's main purpose has been to test the parallel processing functionality
6
7  .. codeauthor:: jhkwakkel <j.h.kwakkel (at) tudelft (dot) nl>
8  """
9  from ema_workbench import Model, RealParameter, ScalarOutcome, ema_logging, perform_
10     ↪ experiments
11
12  def some_model(x1=None, x2=None, x3=None):
13      return {"y": x1 * x2 + x3}
14
15
16  if __name__ == "__main__":
17      ema_logging.LOG_FORMAT = "[% (name)s/% (levelname)s/% (processName)s] %(message)s"
18      ema_logging.log_to_stderr(ema_logging.INFO)
19
20      model = Model("simpleModel", function=some_model) # instantiate the model
21
22      # specify uncertainties
23      model.uncertainties = [
24          RealParameter("x1", 0.1, 10),
25          RealParameter("x2", -0.01, 0.01),
26          RealParameter("x3", -0.01, 0.01),
27      ]
28      # specify outcomes
29      model.outcomes = [ScalarOutcome("y")]
30
31      results = perform_experiments(model, 100)

```

1.5.2 A simple model in Vensim

Imagine we have a very simple Vensim model:



For this example, we assume that 'x11' and 'x12' are uncertain. The state variable 'a' is the outcome of interest. Similar to the previous example, we have to first instantiate a vensim model object, in this case `VensimModel`. To this end, we need to specify the directory in which the vensim file resides, the name of the vensim file and the name of the model.

```
wd = r'./models/vensim example'
model = VensimModel("simpleModel", wd=wd, model_file=r'\model.vpm')
```

Next, we can specify the uncertainties and the outcomes.

```
1 model.uncertainties = [RealParameter("x11", 0, 2.5),
2                         RealParameter("x12", -2.5, 2.5)]
3
4
5 model.outcomes = [TimeSeriesOutcome('a')]
```

Note that we are using a `TimeSeriesOutcome`, because vensim results are time series. We can now simply run this model by calling `perform_experiments()`.

```
with MultiprocessingEvaluator(model) as evaluator:
    results = evaluator.perform_experiments(1000)
```

We now use a evaluator, which ensures that the code is executed in parallel.

Is it generally good practice to first run a model a small number of times sequentially prior to running in parallel. In this way, bugs etc. can be spotted more easily. To further help with keeping track of what is going on, it is also good practice to make use of the logging functionality provided by the workbench

```
ema_logging.log_to_stderr(ema_logging.INFO)
```

Typically, this line appears at the start of the script. When executing the code, messages on progress or on errors will be shown.

The complete code

```
1  """
2  Created on 3 Jan. 2011
3
4  This file illustrated the use the EMA classes for a contrived vensim
5  example
6
7
8  .. codeauthor:: jhkwakkel <j.h.kwakkel (at) tudelft (dot) nl>
9                  chamarat <c.hamarat (at) tudelft (dot) nl>
10
11  """
12  from ema_workbench import TimeSeriesOutcome, perform_experiments, RealParameter, ema_
13  ↪ logging
14
15  from ema_workbench.connectors.vensim import VensimModel
16
17  if __name__ == "__main__":
18      # turn on logging
19      ema_logging.log_to_stderr(ema_logging.INFO)
```

(continues on next page)

(continued from previous page)

```

19  # instantiate a model
20  wd = "./models/vensim example"
21  vensimModel = VensimModel("simpleModel", wd=wd, model_file="model.vpm")
22  vensimModel.uncertainties = [RealParameter("x11", 0, 2.5), RealParameter("x12", -2.5,
↪ 2.5)]
23
24  vensimModel.outcomes = [TimeSeriesOutcome("a")]
25
26  results = perform_experiments(vensimModel, 1000)

```

1.5.3 A simple model in Excel

In order to perform EMA on an Excel model, one can use the `ExcelModel`. This base class makes use of naming cells in Excel to refer to them directly. That is, we can assume that the names of the uncertainties correspond to named cells in Excel, and similarly, that the names of the outcomes correspond to named cells or ranges of cells in Excel. When using this class, make sure that the decimal separator and thousands separator are set correctly in Excel. This can be checked via `file > options > advanced`. These separators should follow the [anglo saxon convention](#).

```

1  """
2  Created on 27 Jul. 2011
3
4  This file illustrates the use the EMA classes for a model in Excel.
5
6  It used the excel file provided by
7  `A. Sharov <https://home.comcast.net/~sharov/PopEcol/lec10/fullmod.html>`
8
9  This excel file implements a simple predator prey model.
10
11  .. codeauthor:: jhkwakkel <j.h.kwakkel (at) tudelft (dot) nl>
12  """
13  from ema_workbench import RealParameter, TimeSeriesOutcome, ema_logging, perform_
↪ experiments
14
15  from ema_workbench.connectors.excel import ExcelModel
16  from ema_workbench.em_framework.evaluators import MultiprocessingEvaluator
17
18  if __name__ == "__main__":
19      ema_logging.log_to_stderr(level=ema_logging.INFO)
20
21      model = ExcelModel("predatorPrey", wd="./models/excelModel", model_file="excel_
↪ example.xlsx")
22      model.uncertainties = [
23          RealParameter("K2", 0.01, 0.2),
24          # we can refer to a cell in the normal way
25          # we can also use named cells
26          RealParameter("KKK", 450, 550),
27          RealParameter("rP", 0.05, 0.15),
28          RealParameter("aaa", 0.00001, 0.25),
29          RealParameter("tH", 0.45, 0.55),
30          RealParameter("kk", 0.1, 0.3),
31      ]

```

(continues on next page)

(continued from previous page)

```

32
33 # specification of the outcomes
34 model.outcomes = [
35     TimeSeriesOutcome("B4:B1076"),
36     # we can refer to a range in the normal way
37     TimeSeriesOutcome("P_t"),
38 ] # we can also use named range
39
40 # name of the sheet
41 model.default_sheet = "Sheet1"
42
43 with MultiprocessingEvaluator(model) as evaluator:
44     results = perform_experiments(model, 100, reporting_interval=1,
    ↪ evaluator=evaluator)

```

The example is relatively straight forward. We instantiate an excel model, we specify the uncertainties and the outcomes. We also need to specify the sheet in excel on which the model resides. Next we can call `perform_experiments()`.

Warning: when using named cells. Make sure that the names are defined at the sheet level and not at the workbook level

1.5.4 A more elaborate example: Mexican Flu

This example is derived from [Pruyt & Hamarat \(2010\)](#). This paper presents a small exploratory System Dynamics model related to the dynamics of the 2009 flu pandemic, also known as the Mexican flu, swine flu, or A(H1N1)v. The model was developed in May 2009 in order to quickly foster understanding about the possible dynamics of this new flu variant and to perform rough-cut policy explorations. Later, the model was also used to further develop and illustrate Exploratory Modelling and Analysis.

Mexican Flu: the basic model

In the first days, weeks and months after the first reports about the outbreak of a new flu variant in Mexico and the USA, much remained unknown about the possible dynamics and consequences of the at the time plausible/imminent epidemic/pandemic of the new flu variant, first known as Swine or Mexican flu and known today as Influenza A(H1N1)v.

The exploratory model presented here is small, simple, high-level, data-poor (no complex/special structures nor detailed data beyond crude guestimates), and history-poor.

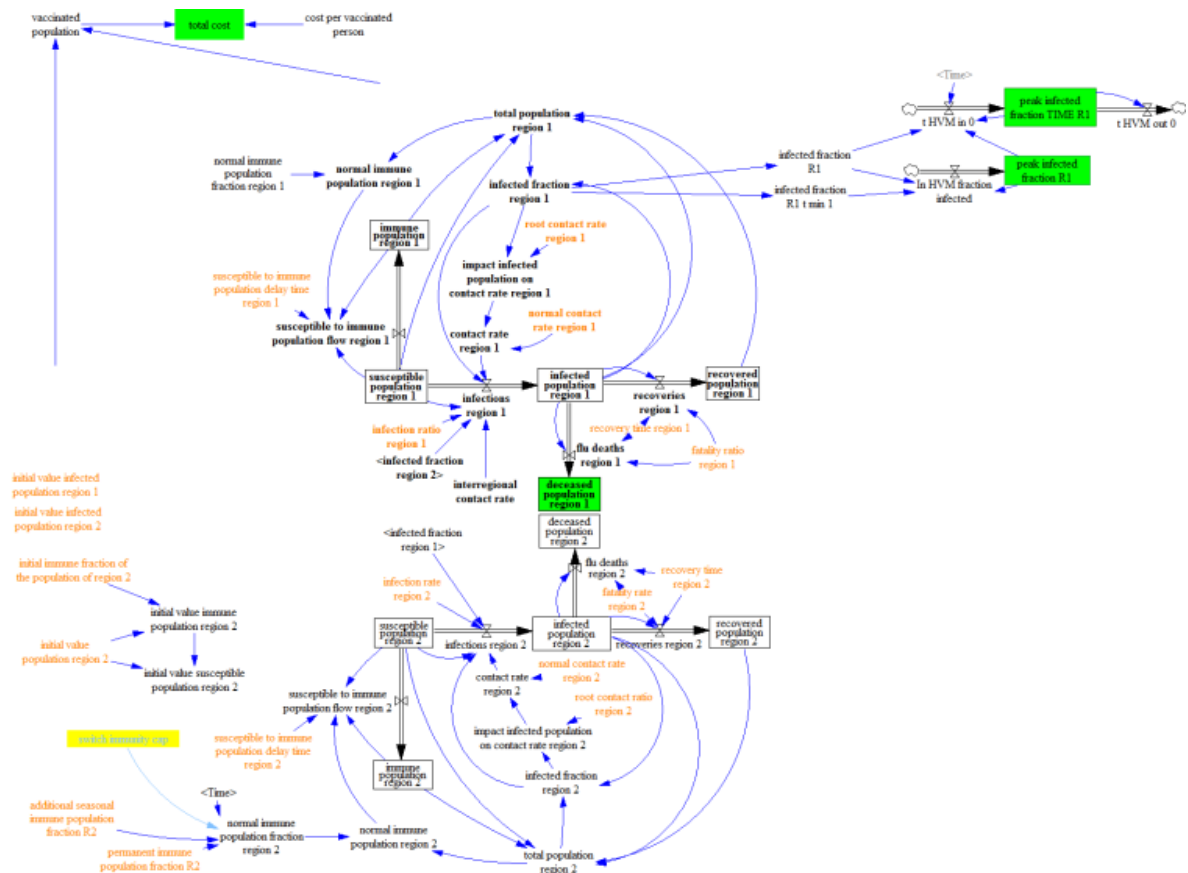
The modelled world is divided in three regions: the Western World, the densely populated Developing World, and the scarcely populated Developing World. Only the two first regions are included in the model because it is assumed that the scarcely populated regions are causally less important for dynamics of flu pandemics. Below, the figure shows the basic stock-and-flow structure. For a more elaborate description of the model, see [Pruyt & Hamarat \(2010\)](#).

Given the various uncertainties about the exact characteristics of the flu, including its fatality rate, the contact rate, the susceptibility of the population, etc. the flu case is an ideal candidate for EMA. One can use EMA to explore the kinds of dynamics that can occur, identify undesirable dynamic, and develop policies targeted at the undesirable dynamics.

In the original paper, [Pruyt & Hamarat \(2010\)](#). recoded the model in Python and performed the analysis in that way. Here we show how the EMA workbench can be connected to Vensim directly.

The flu model was build in Vensim. We can thus use `VensimModelS` as a base class.

We are interested in two outcomes:



- **deceased population region 1:** the total number of deaths over the duration of the simulation.
- **peak infected fraction:** the fraction of the population that is infected.

These are added to `self.outcomes`, using the `TimeSeriesOutcome` class.

The table below is adapted from [Pruyt & Hamarat \(2010\)](#). It shows the uncertainties, and their bounds. These are added to `self.uncertainties` as `ParameterUncertainty` instances.

Parameter	Lower Limit	Upper Limit
additional seasonal immune population fraction region 1	0.0	0.5
additional seasonal immune population fraction region 2	0.0	0.5
fatality ratio region 1	0.0001	0.1
fatality ratio region 2	0.0001	0.1
initial immune fraction of the population of region 1	0.0	0.5
initial immune fraction of the population of region 2	0.0	0.5
normal interregional contact rate	0.0	0.9
permanent immune population fraction region 1	0.0	0.5
permanent immune population fraction region 2	0.0	0.5
recovery time region 1	0.2	0.8
recovery time region 2	0.2	0.8
root contact rate region 1	1.0	10.0
root contact rate region 2	1.0	10.0
infection ratio region 1	0.0	0.1
infection ratio region 2	0.0	0.1
normal contact rate region 1	10	200
normal contact rate region 2	10	200

Together, this results in the following code:

```

1  """
2  Created on 20 May, 2011
3
4  This module shows how you can use vensim models directly
5  instead of coding the model in Python. The underlying case
6  is the same as used in fluExample
7
8  .. codeauthor:: jhkwakkel <j.h.kwakkel (at) tudelft (dot) nl>
9                  epruyt <e.pruyt (at) tudelft (dot) nl>
10
11  """
12  from ema_workbench import (
13      RealParameter,
14      TimeSeriesOutcome,
15      ema_logging,
16      perform_experiments,
17      MultiprocessingEvaluator,
18  )
19
20  from ema_workbench.connectors.vensim import VensimModel
21
22  if __name__ == "__main__":
23      ema_logging.log_to_stderr(ema_logging.INFO)

```

(continues on next page)

(continued from previous page)

```

24     model = VensimModel("fluCase", wd="./models/flu", model_file="FLUvensimV1basecase.vpm
↪")
25
26     # outcomes
27     model.outcomes = [
28         TimeSeriesOutcome("deceased population region 1"),
29         TimeSeriesOutcome("infected fraction R1"),
30     ]
31
32     # Plain Parametric Uncertainties
33     model.uncertainties = [
34         RealParameter("additional seasonal immune population fraction R1", 0, 0.5),
35         RealParameter("additional seasonal immune population fraction R2", 0, 0.5),
36         RealParameter("fatality ratio region 1", 0.0001, 0.1),
37         RealParameter("fatality rate region 2", 0.0001, 0.1),
38         RealParameter("initial immune fraction of the population of region 1", 0, 0.5),
39         RealParameter("initial immune fraction of the population of region 2", 0, 0.5),
40         RealParameter("normal interregional contact rate", 0, 0.9),
41         RealParameter("permanent immune population fraction R1", 0, 0.5),
42         RealParameter("permanent immune population fraction R2", 0, 0.5),
43         RealParameter("recovery time region 1", 0.1, 0.75),
44         RealParameter("recovery time region 2", 0.1, 0.75),
45         RealParameter("susceptible to immune population delay time region 1", 0.5, 2),
46         RealParameter("susceptible to immune population delay time region 2", 0.5, 2),
47         RealParameter("root contact rate region 1", 0.01, 5),
48         RealParameter("root contact ratio region 2", 0.01, 5),
49         RealParameter("infection ratio region 1", 0, 0.15),
50         RealParameter("infection rate region 2", 0, 0.15),
51         RealParameter("normal contact rate region 1", 10, 100),
52         RealParameter("normal contact rate region 2", 10, 200),
53     ]
54
55     nr_experiments = 10
56     with MultiprocessingEvaluator(model) as evaluator:
57         results = perform_experiments(model, nr_experiments, evaluator=evaluator)

```

We have now instantiated the model, specified the uncertain factors and outcomes and run the model. We now have generated a dataset of results and can proceed to analyse the results using various analysis scripts. As a first step, one can look at the individual runs using a line plot using `lines()`. See plotting for some more visualizations using results from performing EMA on `FluModel`.

```

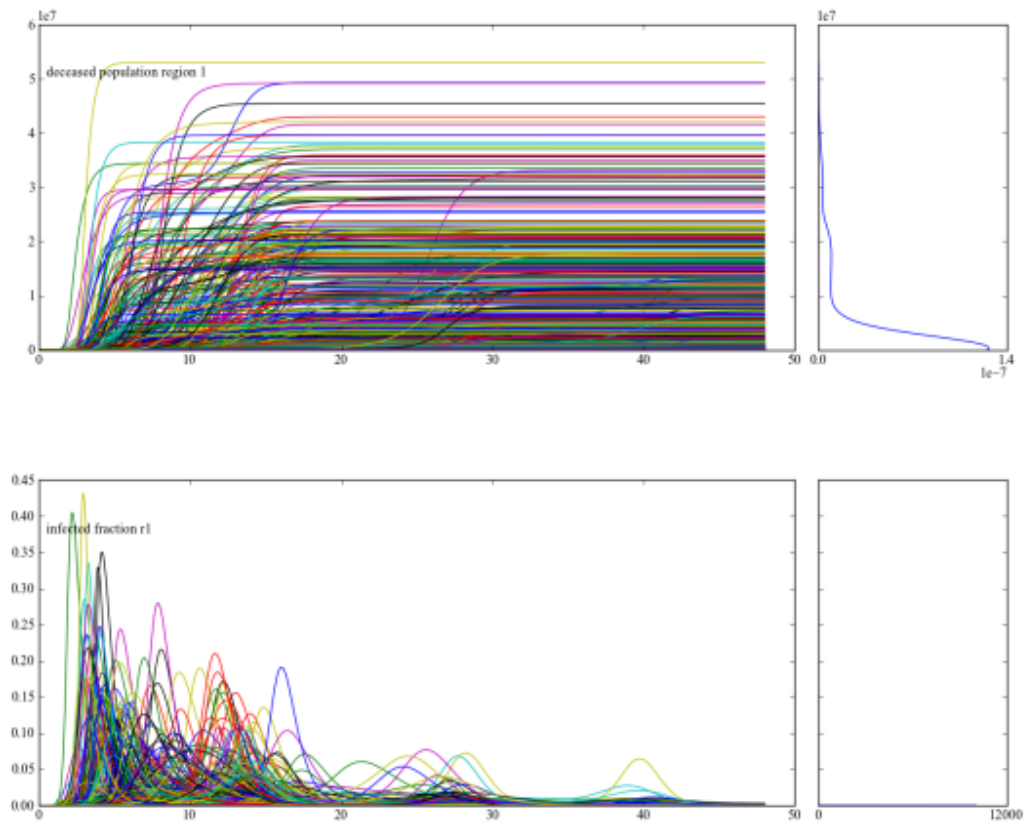
1  import matplotlib.pyplot as plt
2  from ema_workbench.analysis.plotting import lines
3
4  figure = lines(results, density=True) #show lines, and end state density
5  plt.show() #show figure

```

generates the following figure:

From this figure, one can deduce that across the ensemble of possible futures, there is a subset of runs with a substantial amount of deaths. We can zoom in on those cases, identify their conditions for occurring, and use this insight for policy design.

For further analysis, it is generally convenient, to generate the results for a series of experiments and save these results.



One can then use these saved results in various analysis scripts.

```
from ema_workbench import save_results
save_results(results, './1000 runs.tar.gz')
```

The above code snippet shows how we can use `save_results()` for saving the results of our experiments. `save_results()` stores the as csv files in a tarball.

Mexican Flu: policies

For this paper, policies were developed by using the system understanding of the analysts.

static policy

adaptive policy

running the policies

In order to be able to run the models with the policies and to compare their results with the no policy case, we need to specify the policies

```
1 #add policies
2 policies = [Policy('no policy',
3                   model_file=r'/FLUvensimV1basecase.vpm'),
4             Policy('static policy',
5                   model_file=r'/FLUvensimV1static.vpm'),
6             Policy('adaptive policy',
7                   model_file=r'/FLUvensimV1dynamic.vpm')
8             ]
```

In this case, we have chosen to have the policies implemented in separate vensim files. Policies require a name, and can take any other keyword arguments you like. If the keyword matches an attribute on the model object, it will be updated, so `model_file` is an attribute on the vensim model. When executing the policies, we update this attribute for each policy. We can pass these policies to `perform_experiment()` as an additional keyword argument

```
results = perform_experiments(model, 1000, policies=policies)
```

We can now proceed in the same way as before, and perform a series of experiments. Together, this results in the following code:

```
1 """
2 Created on 20 May, 2011
3
4 This module shows how you can use vensim models directly
5 instead of coding the model in Python. The underlying case
6 is the same as used in fluExample
7
8 .. codeauthor:: jhkwakkel <j.h.kwakkel (at) tudelft (dot) nl>
9                epruyt <e.pruyt (at) tudelft (dot) nl>
10 """
11 import numpy as np
12
```

(continues on next page)

(continued from previous page)

```

13 from ema_workbench import (
14     RealParameter,
15     TimeSeriesOutcome,
16     ema_logging,
17     ScalarOutcome,
18     perform_experiments,
19 )
20 from ema_workbench.connectors.vensim import VensimModel
21 from ema_workbench.em_framework.parameters import Policy
22
23 if __name__ == "__main__":
24     ema_logging.log_to_stderr(ema_logging.INFO)
25
26     model = VensimModel("fluCase", wd=r"./models/flu", model_file=r"FLUvensimV1basecase.
↳vpm")
27
28     # outcomes
29     model.outcomes = [
30         TimeSeriesOutcome("deceased population region 1"),
31         TimeSeriesOutcome("infected fraction R1"),
32         ScalarOutcome(
33             "max infection fraction", variable_name="infected fraction R1", function=np.
↳max
34         ),
35     ]
36
37     # Plain Parametric Uncertainties
38     model.uncertainties = [
39         RealParameter("additional seasonal immune population fraction R1", 0, 0.5),
40         RealParameter("additional seasonal immune population fraction R2", 0, 0.5),
41         RealParameter("fatality ratio region 1", 0.0001, 0.1),
42         RealParameter("fatality rate region 2", 0.0001, 0.1),
43         RealParameter("initial immune fraction of the population of region 1", 0, 0.5),
44         RealParameter("initial immune fraction of the population of region 2", 0, 0.5),
45         RealParameter("normal interregional contact rate", 0, 0.9),
46         RealParameter("permanent immune population fraction R1", 0, 0.5),
47         RealParameter("permanent immune population fraction R2", 0, 0.5),
48         RealParameter("recovery time region 1", 0.1, 0.75),
49         RealParameter("recovery time region 2", 0.1, 0.75),
50         RealParameter("susceptible to immune population delay time region 1", 0.5, 2),
51         RealParameter("susceptible to immune population delay time region 2", 0.5, 2),
52         RealParameter("root contact rate region 1", 0.01, 5),
53         RealParameter("root contact ratio region 2", 0.01, 5),
54         RealParameter("infection ratio region 1", 0, 0.15),
55         RealParameter("infection rate region 2", 0, 0.15),
56         RealParameter("normal contact rate region 1", 10, 100),
57         RealParameter("normal contact rate region 2", 10, 200),
58     ]
59
60     # add policies
61     policies = [
62         Policy("no policy", model_file=r"FLUvensimV1basecase.vpm"),

```

(continues on next page)

(continued from previous page)

```

63     Policy("static policy", model_file=r"FLUvensimV1static.vpm"),
64     Policy("adaptive policy", model_file=r"FLUvensimV1dynamic.vpm"),
65 ]
66
67 results = perform_experiments(model, 1000, policies=policies)

```

comparison of results

Using the following script, we reproduce figures similar to the 3D figures in [Pruyt & Hamarat \(2010\)](#). But using `pairs_scatter()`. It shows for the three different policies their behavior on the total number of deaths, the height of the highest peak of the pandemic, and the point in time at which this peak was reached.

```

1  """
2  Created on 20 sep. 2011
3
4  .. codeauthor:: jhkwakkel <j.h.kwakkel (at) tudelft (dot) nl>
5  """
6  import matplotlib.pyplot as plt
7  import numpy as np
8
9  from ema_workbench import load_results, ema_logging
10 from ema_workbench.analysis.pairs_plotting import pairs_lines, pairs_scatter, pairs_
    ↪ density
11
12 ema_logging.log_to_stderr(level=ema_logging.DEFAULT_LEVEL)
13
14 # load the data
15 fh = "./data/1000 flu cases no policy.tar.gz"
16 experiments, outcomes = load_results(fh)
17
18 # transform the results to the required format
19 # that is, we want to know the max peak and the casualties at the end of the
20 # run
21 tr = {}
22
23 # get time and remove it from the dict
24 time = outcomes.pop("TIME")
25
26 for key, value in outcomes.items():
27     if key == "deceased population region 1":
28         tr[key] = value[:, -1] # we want the end value
29     else:
30         # we want the maximum value of the peak
31         max_peak = np.max(value, axis=1)
32         tr["max peak"] = max_peak
33
34         # we want the time at which the maximum occurred
35         # the code here is a bit obscure, I don't know why the transpose
36         # of value is needed. This however does produce the appropriate results
37         logical = value.T == np.max(value, axis=1)
38         tr["time of max"] = time[logical.T]

```

(continues on next page)

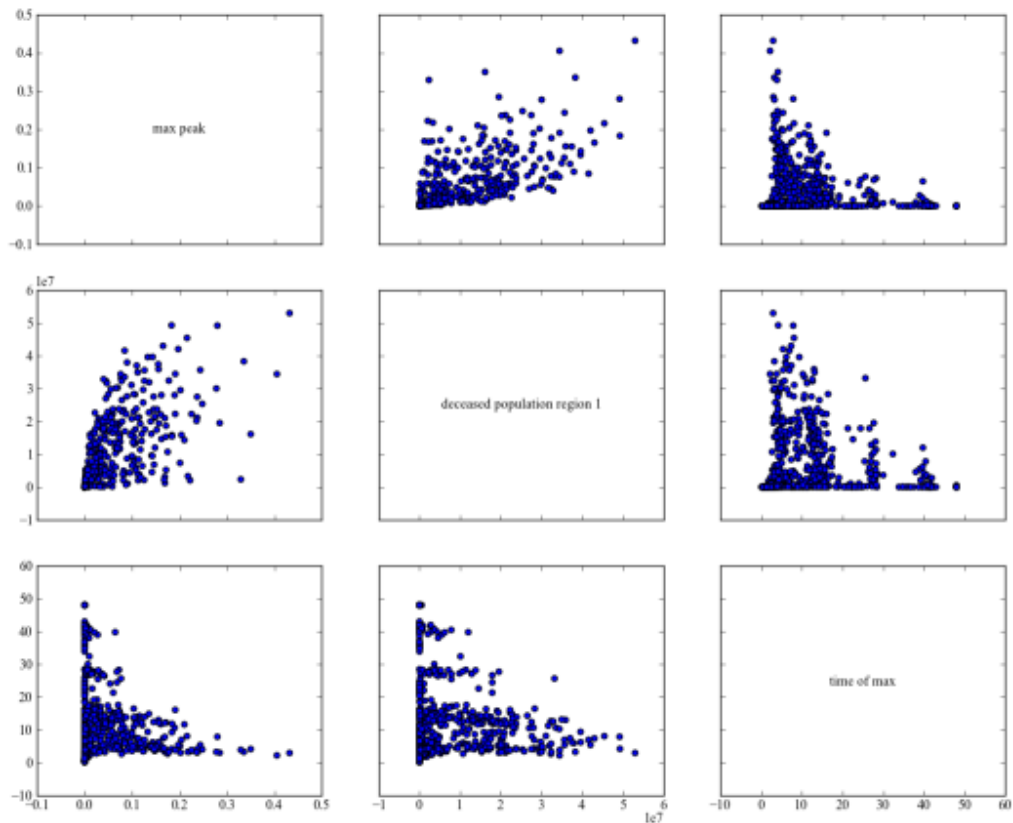
(continued from previous page)

```

39 pairs_scatter(experiments, tr, filter_scalar=False)
40 pairs_lines(experiments, outcomes)
41 pairs_density(experiments, tr, filter_scalar=False)
42 plt.show()

```

no policy

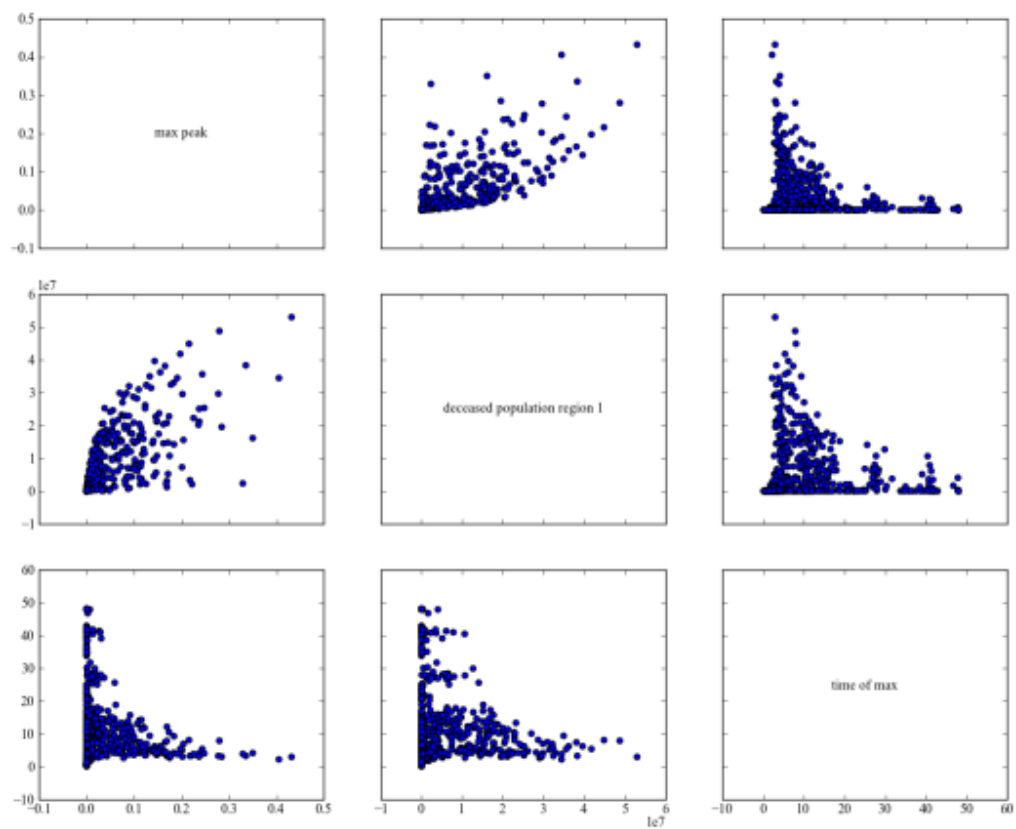


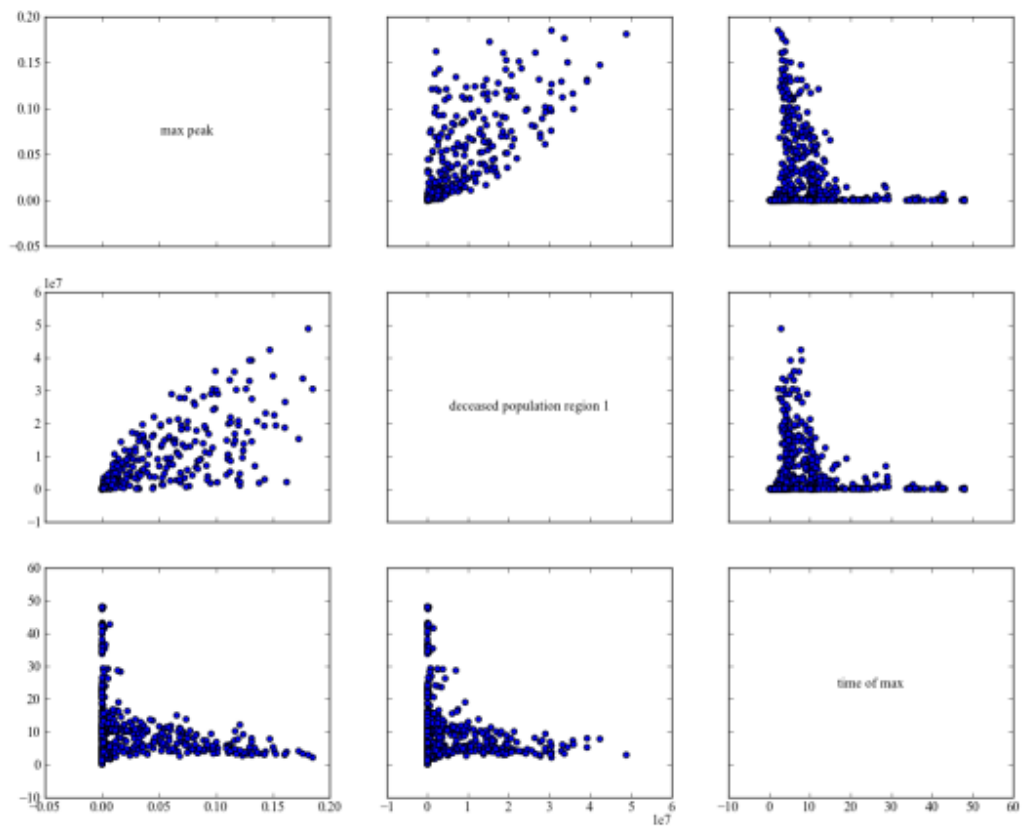
static policy

adaptive policy

1.6 General Introduction

Since 2010, I have been working on the development of an open source toolkit for supporting decision-making under deep uncertainty. This toolkit is known as the exploratory modeling workbench. The motivation for this name is that in my opinion all model-based deep uncertainty approaches are forms of exploratory modeling as first introduced by





[Banks \(1993\)](#). The design of the workbench has undergone various changes over time, but it has started to stabilize in the fall of 2016. In the summer of 2017, I published a paper detailing the workbench ([Kwakkel, 2017](#)). There is an in depth example in the paper, but for the documentation I want to provide a more tutorial style description of the functionality of the workbench.

As a starting point, I will use the Direct Policy Search example that is available for Rhodium ([Quinn et al 2017](#)). A quick note on terminology is in order here. I have a background in transport modeling. Here we often use discrete event simulation models. These are intrinsically stochastic models. It is standard practice to run these models several times and take descriptive statistics over the set of runs. In discrete event simulation, and also in the context of agent based modeling, this is known as running replications. The workbench adopts this terminology and draws a sharp distinction between designing experiments over a set of deeply uncertain factors, and performing replications of this experiment to cope with stochastic uncertainty.

```
[1]: import math

# more or less default imports when using
# the workbench
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

from scipy.optimize import brentq

def get_antropogenic_release(xt, c1, c2, r1, r2, w1):
    """
    Parameters
    -----
    xt : float
        pollution in lake at time t
    c1 : float
        center rbf 1
    c2 : float
        center rbf 2
    r1 : float
        radius rbf 1
    r2 : float
        radius rbf 2
    w1 : float
        weight of rbf 1

    Returns
    -----
    float

    note:: w2 = 1 - w1

    """

    rule = w1 * (abs(xt - c1) / r1) ** 3 + (1 - w1) * (abs(xt - c2) / r2) ** 3
    at1 = max(rule, 0.01)
    at = min(at1, 0.1)
```

(continues on next page)

(continued from previous page)

```

    return at

def lake_model(
    b=0.42,
    q=2.0,
    mean=0.02,
    stdev=0.001,
    delta=0.98,
    alpha=0.4,
    nsamples=100,
    myears=100,
    c1=0.25,
    c2=0.25,
    r1=0.5,
    r2=0.5,
    w1=0.5,
    seed=None,
):
    """runs the lake model for nsamples stochastic realisation using
    specified random seed.

    Parameters
    -----
    b : float
        decay rate for P in lake (0.42 = irreversible)
    q : float
        recycling exponent
    mean : float
        mean of natural inflows
    stdev : float
        standard deviation of natural inflows
    delta : float
        future utility discount rate
    alpha : float
        utility from pollution
    nsamples : int, optional
    myears : int, optional
    c1 : float
    c2 : float
    r1 : float
    r2 : float
    w1 : float
    seed : int, optional
        seed for the random number generator

    Returns
    -----
    tuple

    """

```

(continues on next page)

(continued from previous page)

```

np.random.seed(seed)
Pcrit = brentq(lambda x: x**q / (1 + x**q) - b * x, 0.01, 1.5)

X = np.zeros((myyears,))
average_daily_P = np.zeros((myyears,))
reliability = 0.0
inertia = 0
utility = 0

for _ in range(nsamples):
    X[0] = 0.0
    decision = 0.1

    decisions = np.zeros(myyears)
    decisions[0] = decision

    natural_inflows = np.random.lognormal(
        math.log(mean**2 / math.sqrt(stdev**2 + mean**2)),
        math.sqrt(math.log(1.0 + stdev**2 / mean**2)),
        size=myyears,
    )

    for t in range(1, myyears):
        # here we use the decision rule
        decision = get_antropogenic_release(X[t - 1], c1, c2, r1, r2, w1)
        decisions[t] = decision

        X[t] = (
            (1 - b) * X[t - 1]
            + X[t - 1] ** q / (1 + X[t - 1] ** q)
            + decision
            + natural_inflows[t - 1]
        )
        average_daily_P[t] += X[t] / nsamples

    reliability += np.sum(X < Pcrit) / (nsamples * myyears)
    inertia += np.sum(np.absolute(np.diff(decisions) < 0.02)) / (nsamples * myyears)
    utility += np.sum(alpha * decisions * np.power(delta, np.arange(myyears))) /
↪nsamples
max_P = np.max(average_daily_P)
return max_P, utility, inertia, reliability

```

1.6.1 Connecting a python function to the workbench

Now we are ready to connect this model to the workbench. We have to specify the uncertainties, the outcomes, and the policy levers. For the uncertainties and the levers, we can use real valued parameters, integer valued parameters, and categorical parameters. For outcomes, we can use either scalar, single valued outcomes or time series outcomes. For convenience, we can also explicitly control constants in case we want to have them set to a value different from their default value.

```
[2]: from ema_workbench import RealParameter, ScalarOutcome, Constant, Model
    from dps_lake_model import lake_model

    model = Model("lakeproblem", function=lake_model)

    # specify uncertainties
    model.uncertainties = [
        RealParameter("b", 0.1, 0.45),
        RealParameter("q", 2.0, 4.5),
        RealParameter("mean", 0.01, 0.05),
        RealParameter("stdev", 0.001, 0.005),
        RealParameter("delta", 0.93, 0.99),
    ]

    # set levers
    model.levers = [
        RealParameter("c1", -2, 2),
        RealParameter("c2", -2, 2),
        RealParameter("r1", 0, 2),
        RealParameter("r2", 0, 2),
        RealParameter("w1", 0, 1),
    ]

    # specify outcomes
    model.outcomes = [
        ScalarOutcome("max_P"),
        ScalarOutcome("utility"),
        ScalarOutcome("inertia"),
        ScalarOutcome("reliability"),
    ]

    # override some of the defaults of the model
    model.constants = [
        Constant("alpha", 0.41),
        Constant("nsamples", 150),
        Constant("myears", 100),
    ]
```

1.6.2 Performing experiments

Now that we have specified the model with the workbench, we are ready to perform experiments on it. We can use evaluators to distribute these experiments either over multiple cores on a single machine, or over a cluster using `ipyparallel`. Using any parallelization is an advanced topic, in particular if you are on a windows machine. The code as presented here will run fine in parallel on a mac or Linux machine. If you are trying to run this in parallel using multiprocessing on a windows machine, from within a jupyter notebook, it won't work. The solution is to move the `lake_model` and `get_antropogenic_release` to a separate python module and import the lake model function into the notebook.

Another common practice when working with the exploratory modeling workbench is to turn on the logging functionality that it provides. This will report on the progress of the experiments, as well as provide more insight into what is happening in particular in case of errors.

If we want to perform experiments on the model we have just defined, we can use the `perform_experiments` method on the evaluator, or the stand alone `perform_experiments` function. We can perform experiments over the uncertainties and/or over the levers. Any given parameterization of the levers is known as a policy, while any given parametrization over the uncertainties is known as a scenario. Any policy is evaluated over each of the scenarios. So if we want to use 100 scenarios and 10 policies, this means that we will end up performing $100 * 10 = 1000$ experiments. By default, the workbench uses Latin hypercube sampling for both sampling over levers and sampling over uncertainties. However, the workbench also offers support for full factorial, partial factorial, and Monte Carlo sampling, as well as wrappers for the various sampling schemes provided by [SALib](#).

The `ema_workbench` offers support for parallelization of the execution of the experiments using either the [multiprocessing](#) or [ipyparallel](#). There are various OS specific concerns you have to keep in mind when using either of these libraries. Please have a look at the documentation of these libraries, before using them.

```
[3]: from ema_workbench import MultiprocessingEvaluator, ema_logging, perform_experiments

ema_logging.log_to_stderr(ema_logging.INFO)

with MultiprocessingEvaluator(model) as evaluator:
    results = evaluator.perform_experiments(scenarios=1000, policies=10)

[MainProcess/INFO] pool started with 12 workers
[MainProcess/INFO] performing 1000 scenarios * 10 policies * 1 model(s) = 10000_
↪experiments
100%| 10000/10000 [02:18<00:00, 72.21it/s]
[MainProcess/INFO] experiments finished
[MainProcess/INFO] terminating pool
```

1.6.3 Processing the results of the experiments

By default, the return of `perform_experiments` is a tuple of length 2. The first item in the tuple is the experiments. The second item is the outcomes. Experiments and outcomes are aligned by index. The experiments are stored in a Pandas DataFrame, while the outcomes are a dict with the name of the outcome as key, and the values are in a numpy array.

```
[4]: experiments, outcomes = results
print(experiments.shape)
print(list(outcomes.keys()))

(10000, 13)
['max_P', 'utility', 'inertia', 'reliability']
```

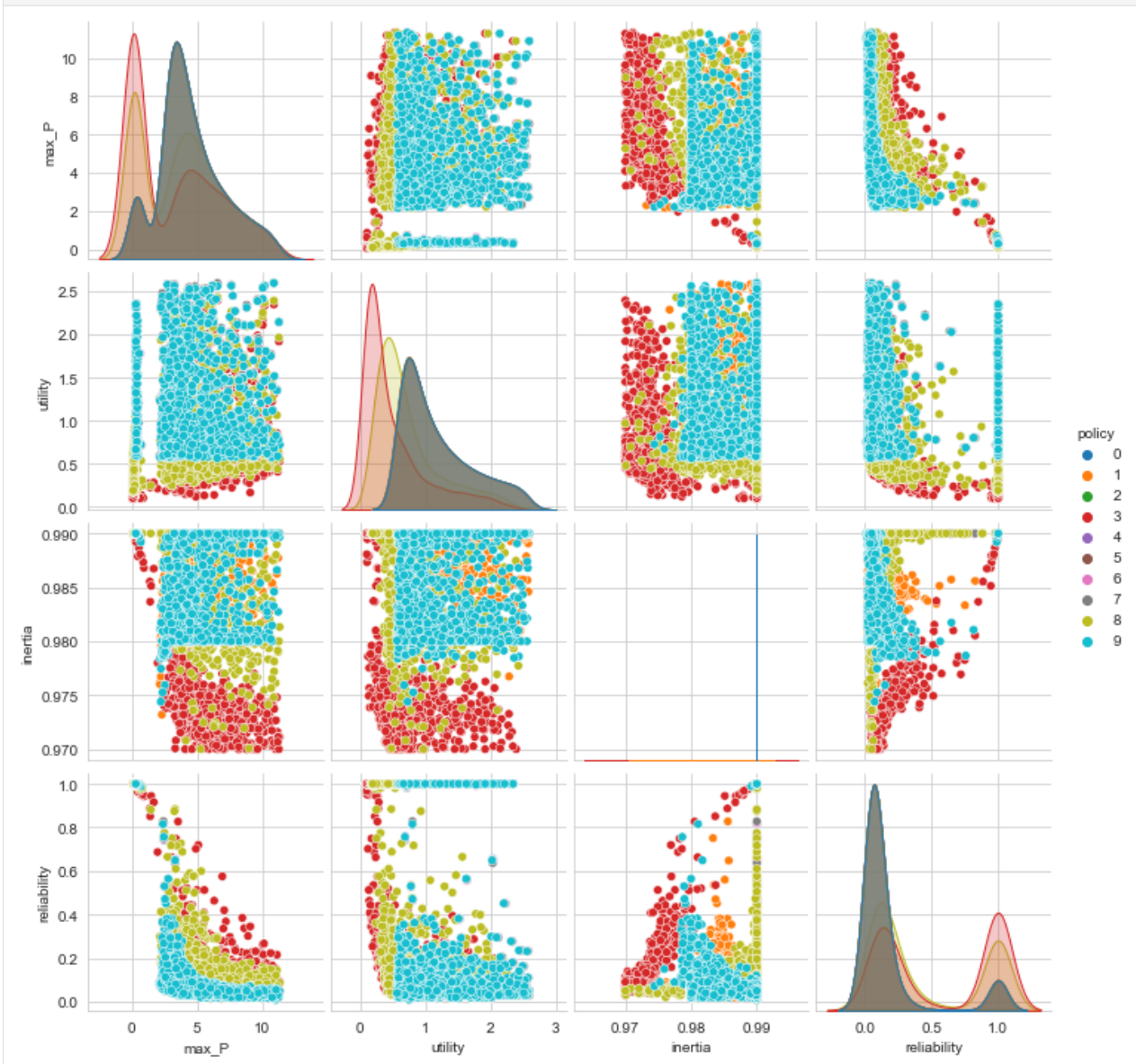

We can easily visualize these results. The workbench comes with various convenience plotting functions built on top of matplotlib and seaborn. We can however also easily use seaborn or matplotlib directly. For example, we can create a pairplot using seaborn where we group our outcomes by policy. For this, we need to create a dataframe with the outcomes and a policy column. By default the name of a policy is a string representation of the dict with lever names and values. We can replace this easily with a number as shown below.

```
[5]: policies = experiments["policy"]
    for i, policy in enumerate(np.unique(policies)):
        experiments.loc[policies == policy, "policy"] = str(i)

    data = pd.DataFrame(outcomes)
    data["policy"] = policies
```

Next, all that is left is to use `seaborn's pairplot` function to visualize the data.

```
[6]: sns.pairplot(data, hue="policy", vars=list(outcomes.keys()))
    plt.show()
```



```
[ ]:
```

1.7 Open exploration

In this second tutorial, I will showcase how to use the `ema_workbench` for performing open exploration. This tutorial will continue with the same example as used in the previous tutorial.

1.7.1 some background

In exploratory modeling, we are interested in understanding how regions in the uncertainty space and/or the decision space map to the whole outcome space, or partitions thereof. There are two general approaches for investigating this mapping. The first one is through systematic sampling of the uncertainty or decision space. This is sometimes also known as open exploration. The second one is to search through the space in a directed manner using some type of optimization approach. This is sometimes also known as directed search.

The workbench support both open exploration and directed search. Both can be applied to investigate the mapping of the uncertainty space and/or the decision space to the outcome space. In most applications, search is used for finding promising mappings from the decision space to the outcome space, while exploration is used to stress test these mappings under a whole range of possible resolutions to the various uncertainties. This need not be the case however. Optimization can be used to discover the worst possible scenario, while sampling can be used to get insight into the sensitivity of outcomes to the various decision levers.

1.7.2 open exploration

To showcase the open exploration functionality, let's start with a basic example using the Direct Policy Search (DPS) version of the lake problem (Quinn et al 2017). This is the same model as we used in the general introduction. Note that for convenience, I have moved the code for the model to a module called `dps_lake_model.py`, which I import here for further use.

We are going to simultaneously sample over uncertainties and decision levers. We are going to generate 1000 scenarios and 5 policies, and see how they jointly affect the outcomes. A *scenario* is understood as a point in the uncertainty space, while a *policy* is a point in the decision space. The combination of a scenario and a policy is called *experiment*. The uncertainty space is spanned by uncertainties, while the decision space is spanned by levers.

Both uncertainties and levers are instances of *RealParameter* (a continuous range), *IntegerParameter* (a range of integers), or *CategoricalParameter* (an unordered set of things). By default, the workbench will use Latin Hypercube sampling for generating both the scenarios and the policies. Each policy will be always evaluated over all scenarios (i.e. a full factorial over scenarios and policies).

```
[1]: from ema_workbench import RealParameter, ScalarOutcome, Constant, Model
    from dps_lake_model import lake_model

    import numpy as np
    import pandas as pd
    import seaborn as sns
    import matplotlib.pyplot as plt

    model = Model("lakeproblem", function=lake_model)

    # specify uncertainties
```

(continues on next page)

(continued from previous page)

```

model.uncertainties = [
    RealParameter("b", 0.1, 0.45),
    RealParameter("q", 2.0, 4.5),
    RealParameter("mean", 0.01, 0.05),
    RealParameter("stdev", 0.001, 0.005),
    RealParameter("delta", 0.93, 0.99),
]

# set levers
model.levers = [
    RealParameter("c1", -2, 2),
    RealParameter("c2", -2, 2),
    RealParameter("r1", 0, 2),
    RealParameter("r2", 0, 2),
    RealParameter("w1", 0, 1),
]

# specify outcomes
model.outcomes = [
    ScalarOutcome("max_P"),
    ScalarOutcome("utility"),
    ScalarOutcome("inertia"),
    ScalarOutcome("reliability"),
]

# override some of the defaults of the model
model.constants = [
    Constant("alpha", 0.41),
    Constant("nsamples", 150),
    Constant("myears", 100),
]

```

```

[2]: from ema_workbench import MultiprocessingEvaluator, ema_logging, perform_experiments

ema_logging.log_to_stderr(ema_logging.INFO)

with MultiprocessingEvaluator(model, n_processes=7) as evaluator:
    experiments, outcomes = evaluator.perform_experiments(scenarios=1000, policies=5)

[MainProcess/INFO] pool started with 7 workers
[MainProcess/INFO] performing 1000 scenarios * 5 policies * 1 model(s) = 5000 experiments
100%| 5000/5000 [01:02<00:00, 80.06it/s]
[MainProcess/INFO] experiments finished
[MainProcess/INFO] terminating pool

```

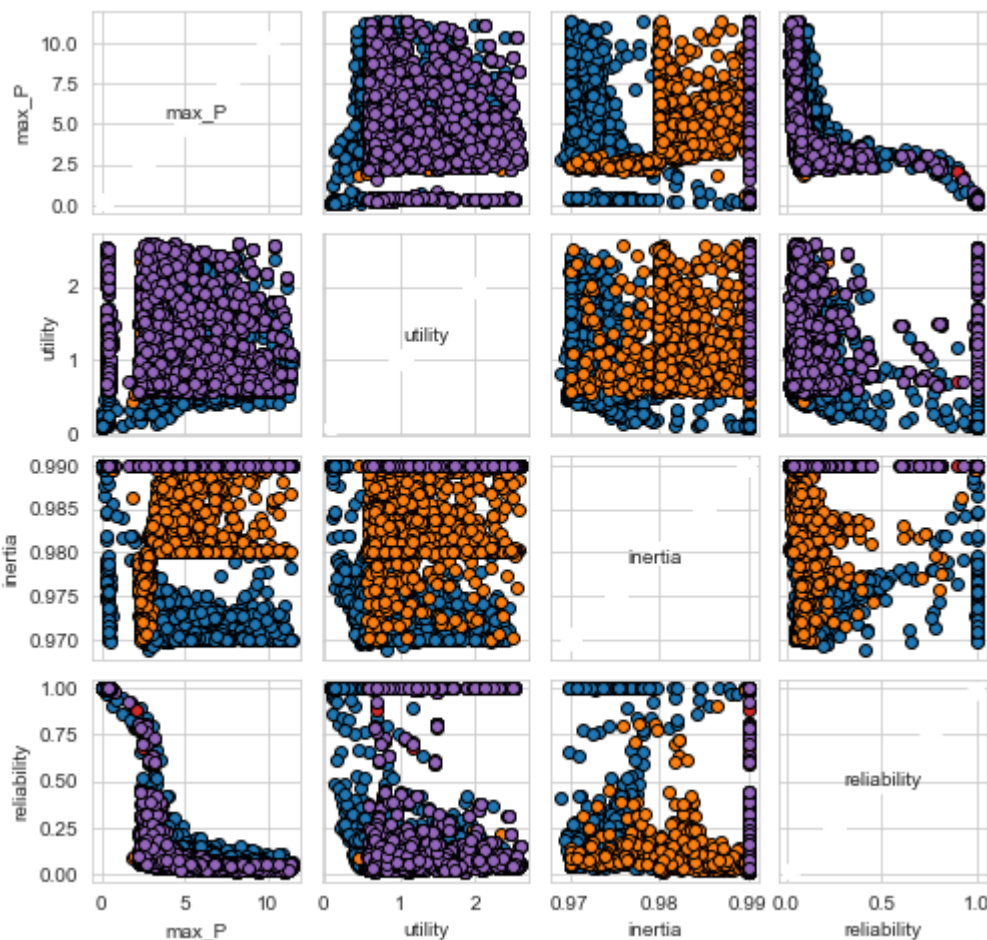
Visual analysis

Having generated these results, the next step is to analyze them and see what we can learn from the results. The workbench comes with a variety of techniques for this analysis. A simple first step is to make a few quick visualizations of the results. The workbench has convenience functions for this, but it is also possible to create your own visualizations using the scientific Python stack.

```
[3]: from ema_workbench.analysis import pairs_plotting

fig, axes = pairs_plotting.pairs_scatter(experiments, outcomes, group_by="policy",
    legend=False)
fig.set_size_inches(8, 8)
plt.show()
```

[MainProcess/INFO] no time dimension found in results



Often, it is convenient to separate the process of performing the experiments from the analysis. To make this possible, the workbench offers convenience functions for storing results to disc and loading them from disc. The workbench will store the results in a tarball with .csv files and separate metadata files. This is a convenient format that has proven sufficient over the years.

```
from ema_workbench import save_results
save_results(results, '1000 scenarios 5 policies.tar.gz')
```

(continues on next page)

(continued from previous page)

```
from ema_workbench import load_results
results = load_results('1000 scenarios 5 policies.tar.gz')
```

1.7.3 advanced analysis

In addition to visual analysis, the workbench comes with a variety of techniques to perform a more in-depth analysis of the results. In addition, other analyses can simply be performed by utilizing the scientific python stack. The workbench comes with

- **Scenario Discovery**, a model driven approach to scenario development
- **Feature Scoring**, a poor man's alternative to global sensitivity analysis
- **Dimensional stacking**, a quick visual approach drawing on feature scoring to enable scenario discovery. This approach has received limited attention in the literature. The implementation in the workbench replaces the rule mining approach with a feature scoring approach.
- **Regional sensitivity analysis**

Scenario Discovery

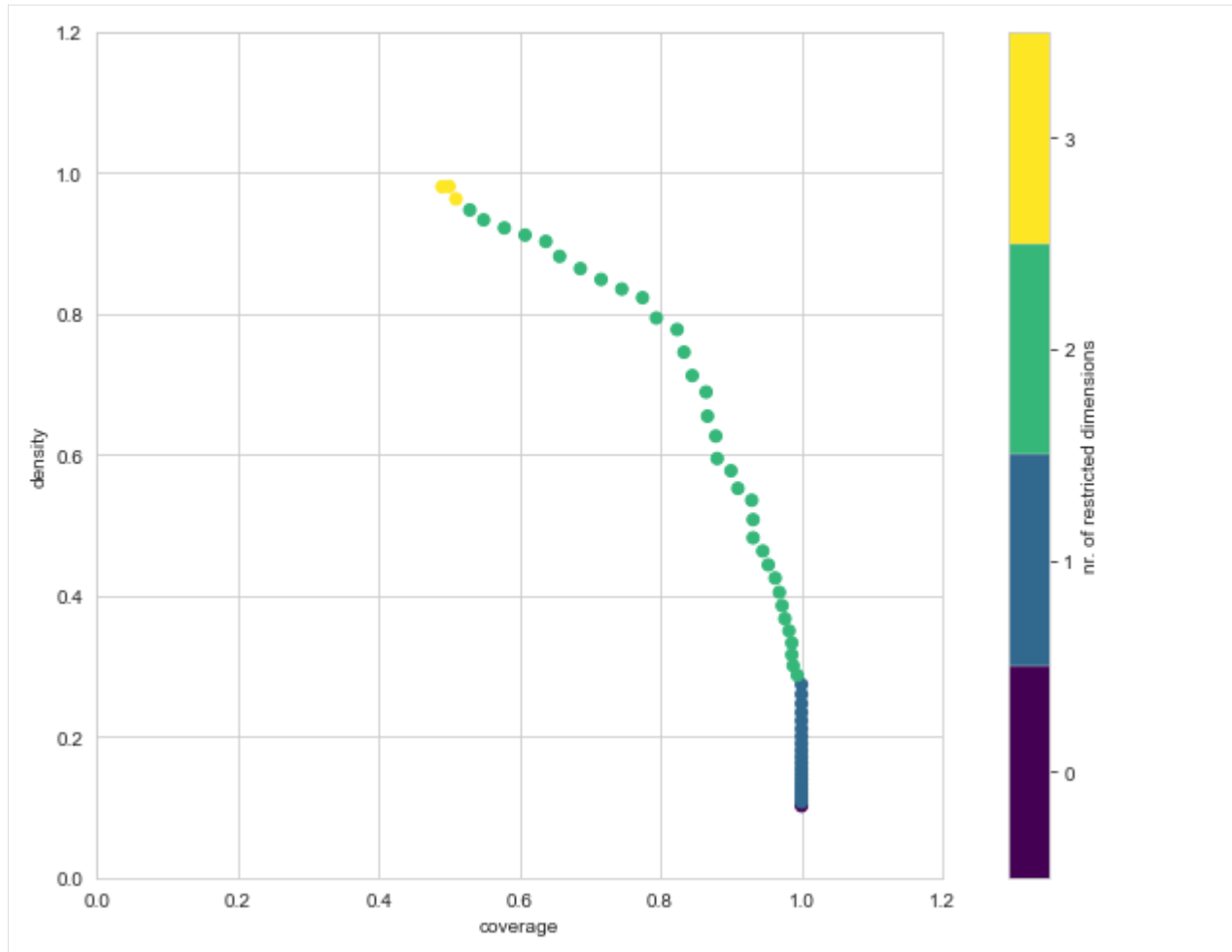
A detailed discussion on scenario discovery can be found in an [earlier blogpost](#). For completeness, I provide a code snippet here. Compared to the previous blog post, there is one small change. The library `mpld3` is currently not being maintained and broken on Python 3.5. and higher. To still utilize the interactive exploration of the trade offs within the notebook, one could use the interactive back-end ([% matplotlib notebook](#)).

```
[4]: from ema_workbench.analysis import prim
```

```
x = experiments
y = outcomes["max_P"] < 0.8
prim_alg = prim.Prim(x, y, threshold=0.8)
box1 = prim_alg.find_box()
```

```
[MainProcess/INFO] model dropped from analysis because only a single category
[MainProcess/INFO] 5000 points remaining, containing 510 cases of interest
[MainProcess/INFO] mean: 0.9807692307692307, mass: 0.052, coverage: 0.5, density: 0.
↪9807692307692307 restricted_dimensions: 3
```

```
[5]: box1.show_tradeoff()
plt.show()
```

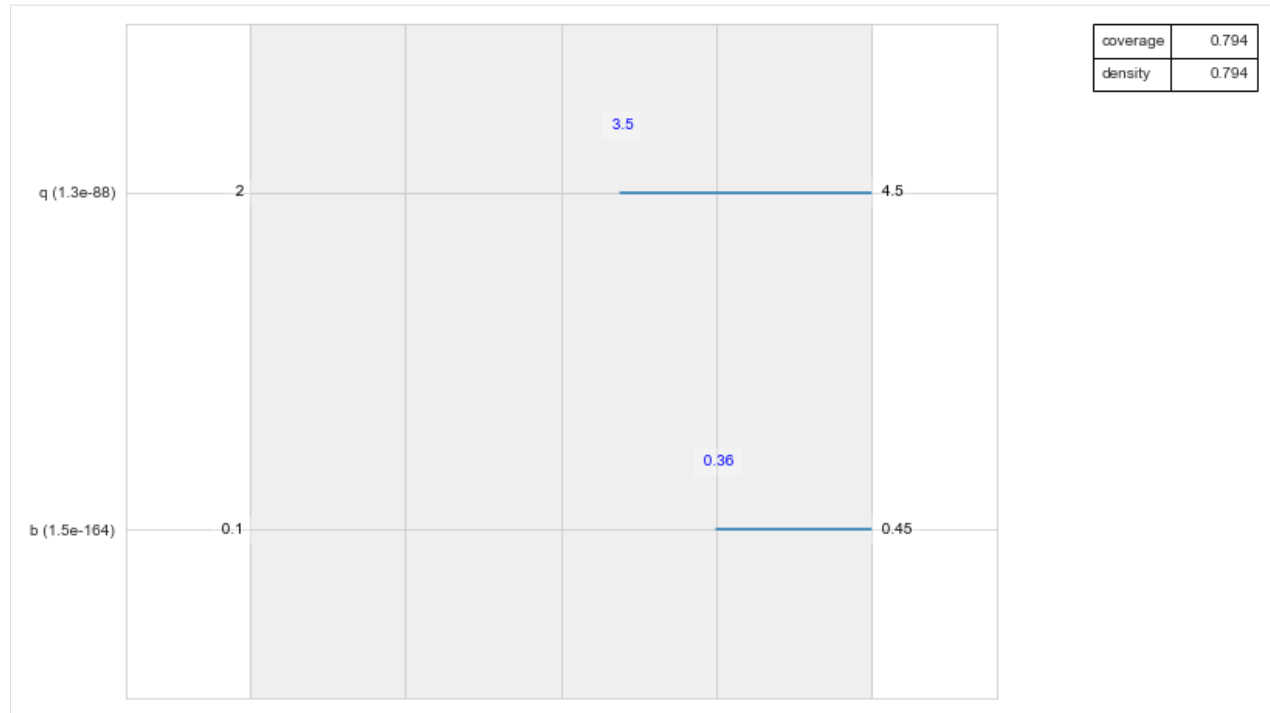


We can inspect any of the points on the trade off curve using the inspect method. As shown, we can show the results either in a table format or in a visual format.

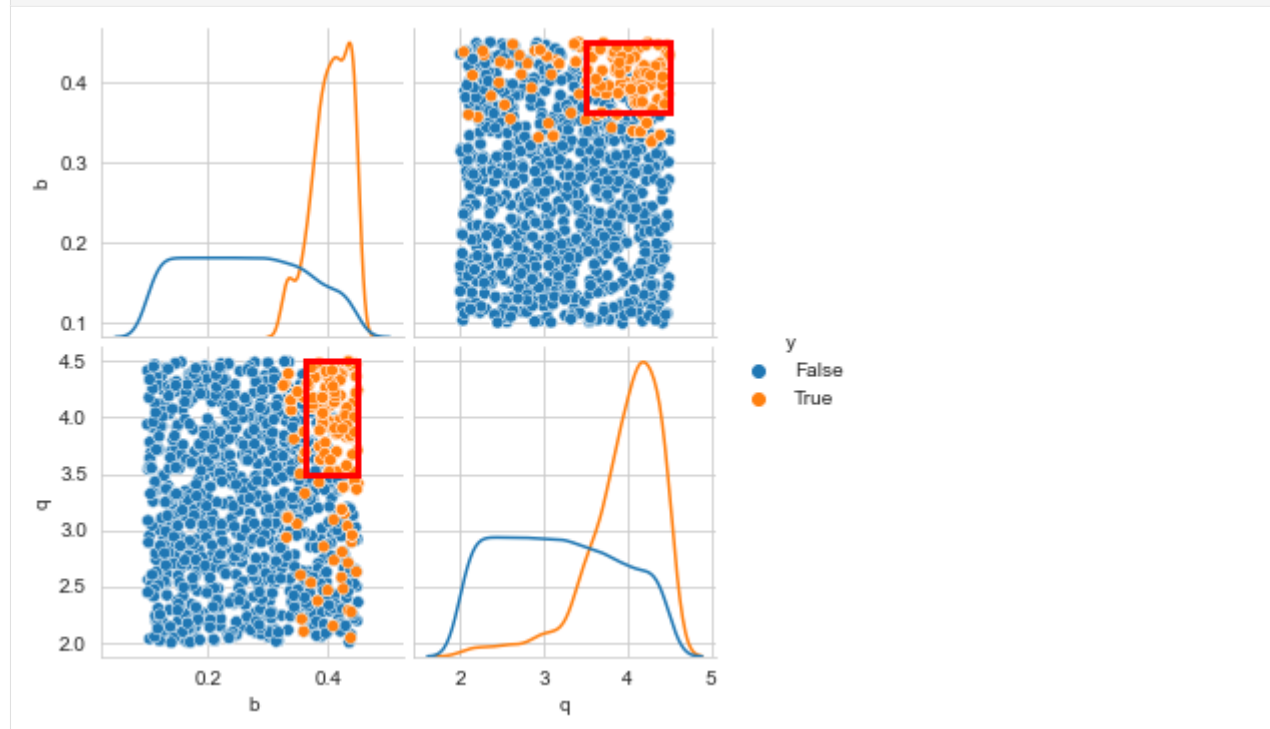
```
[6]: box1.inspect(43)
      box1.inspect(43, style="graph")
      plt.show()
```

```
coverage    0.794118
density      0.794118
id           43
mass         0.102
mean         0.794118
res_dim      2
Name: 43, dtype: object
```

```
      box 43
      min    max          qp values
b  0.362596  0.449742 [1.5372285485377317e-164, -1.0]
q  3.488834  4.498362 [1.3004137205191903e-88, -1.0]
```



```
[7]: box1.show_pairs_scatter(43)
plt.show()
```



feature scoring

Feature scoring is a family of techniques often used in machine learning to identify the most relevant features to include in a model. This is similar to one of the use cases for global sensitivity analysis, namely factor prioritisation. The main advantage of feature scoring techniques is that they impose no specific constraints on the experimental design, while they can handle real valued, integer valued, and categorical valued parameters. The workbench supports multiple techniques, the most useful of which generally is extra trees (Geurts et al. 2006).

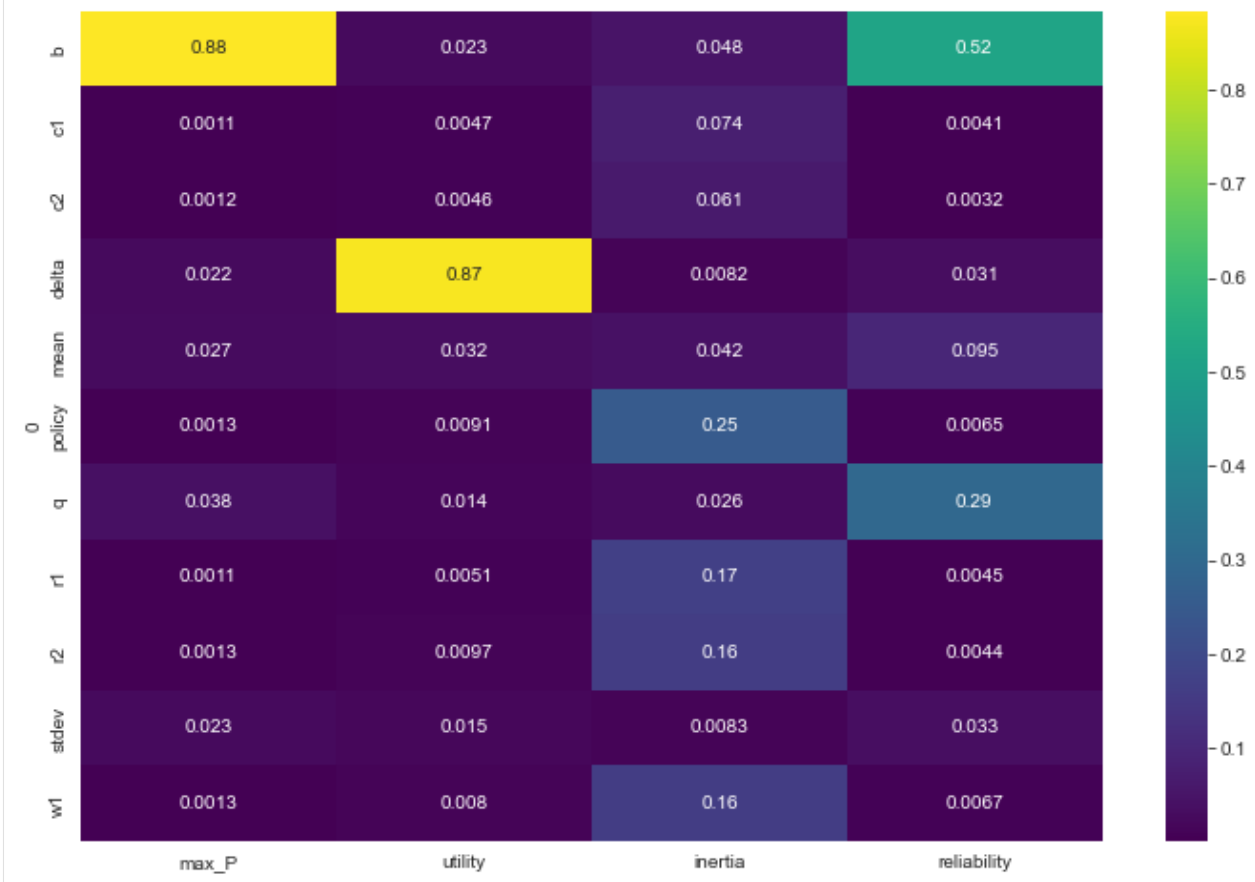
For this example, we run feature scoring for each outcome of interest. We can also run it for a specific outcome if desired. Similarly, we can choose if we want to run in regression mode or classification mode. The later is applicable if the outcome is a categorical variable and the results should be interpreted similar to regional sensitivity analysis results. For more details, see the documentation.

```
[8]: from ema_workbench.analysis import feature_scoring
```

```
x = experiments
y = outcomes
```

```
fs = feature_scoring.get_feature_scores_all(x, y)
sns.heatmap(fs, cmap="viridis", annot=True)
plt.show()
```

```
[MainProcess/INFO] model dropped from analysis because only a single category
[MainProcess/INFO] model dropped from analysis because only a single category
[MainProcess/INFO] model dropped from analysis because only a single category
[MainProcess/INFO] model dropped from analysis because only a single category
```



From the results, we see that \max_P is primarily influenced by b , while utility is driven by δ , for inertia and reliability the situation is a little bit less clear cut.

The foregoing feature scoring was using the raw values of the outcomes. However, in scenario discovery applications, we are typically dealing with a binary classification. This might produce slightly different results as demonstrated below

```
[11]: from ema_workbench.analysis import RuleInductionType

x = experiments
y = outcomes["max_P"] < 0.8

fs, alg = feature_scoring.get_ex_feature_scores(x, y, mode=RuleInductionType.
↳CLASSIFICATION)
fs.sort_values(ascending=False, by=1)
```

[MainProcess/INFO] model dropped from analysis because only a single category

```
[11]:      1
0
b      0.458094
q      0.310989
mean   0.107732
delta  0.051212
stdev  0.047653
policy 0.005475
w1     0.004815
r2     0.003658
r1     0.003607
c1     0.003467
c2     0.003299
```

Here we ran extra trees feature scoring on a binary vector for \max_P . the b parameter is still important, similar to in the previous case, but the introduction of the binary classification now also highlights some additional parameters as being potentially relevant.

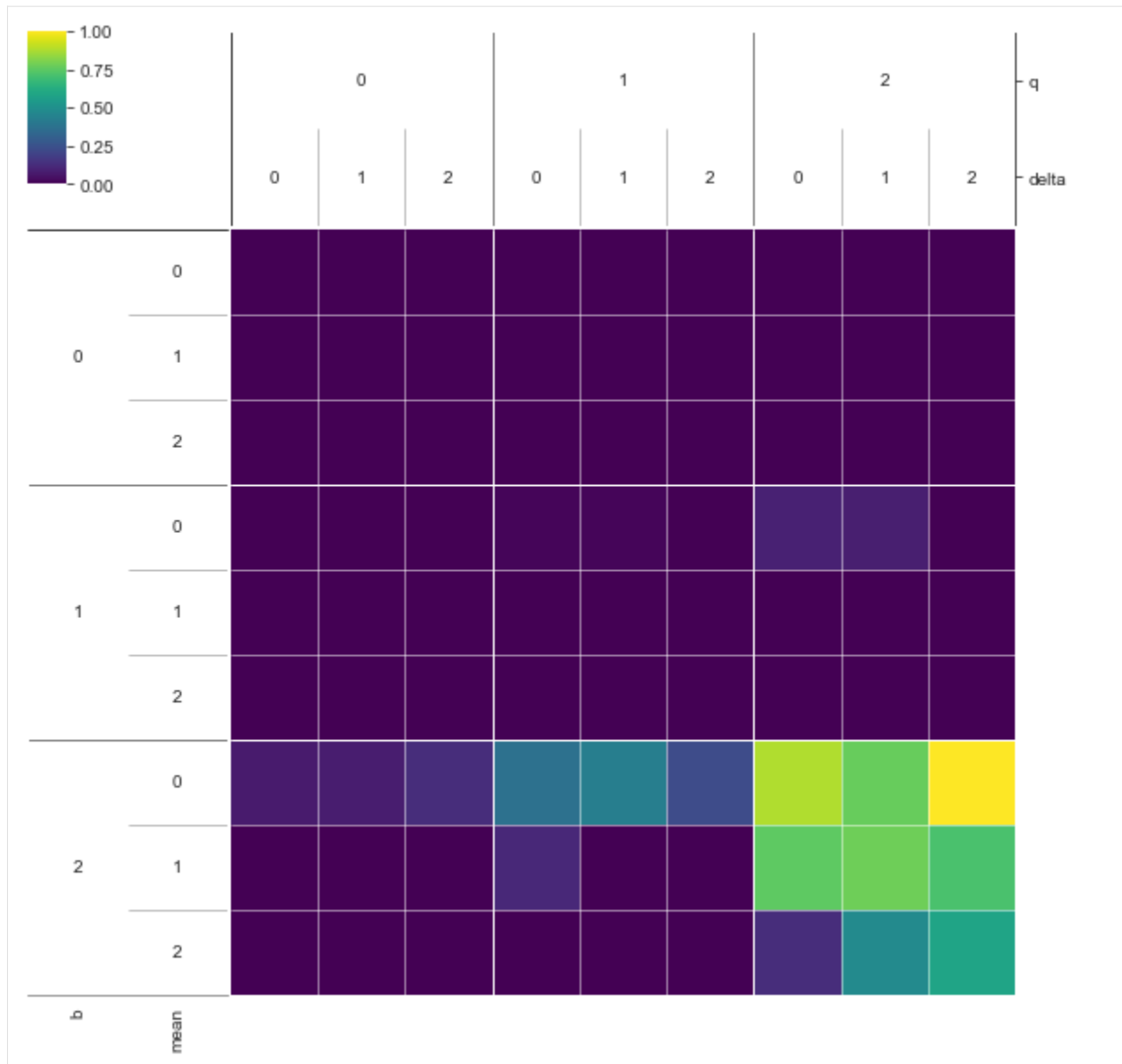
dimensional stacking

Dimensional stacking was suggested as a more visual approach to scenario discovery. It involves two steps: identifying the most important uncertainties that affect system behavior, and creating a pivot table using the most influential uncertainties. In order to do this, we first need, as in scenario discovery, specify the outcomes that are of interest. The creating of the pivot table involves binning the uncertainties. More details can be found in [Suzuki et al. \(2015\)](#) or by looking through the code in the workbench. Compared to Suzuki et al, the workbench uses feature scoring for determining the most influential uncertainties. The code is set up in a modular way so other approaches to global sensitivity analysis can easily be used as well if so desired.

```
[12]: from ema_workbench.analysis import dimensional_stacking

x = experiments
y = outcomes["max_P"] < 0.8
dimensional_stacking.create_pivot_plot(x, y, 2, nbins=3)
plt.show()
```

[MainProcess/INFO] model dropped from analysis because only a single category



We can see from this visual that if B is high, while Q is high, we have a high concentration of cases where pollution stays below 0.8. The mean and stdev have some limited additional influence. By playing around with an alternative number of bins, or different number of layers, patterns can be coarsened or refined.

regional sensitivity analysis

A fourth approach for supporting scenario discovery is to perform a regional sensitivity analysis. The workbench implements a visual approach based on plotting the empirical CDF given a classification vector. Please look at section 3.4 in [Pianosi et al \(2016\)](#) for more details.

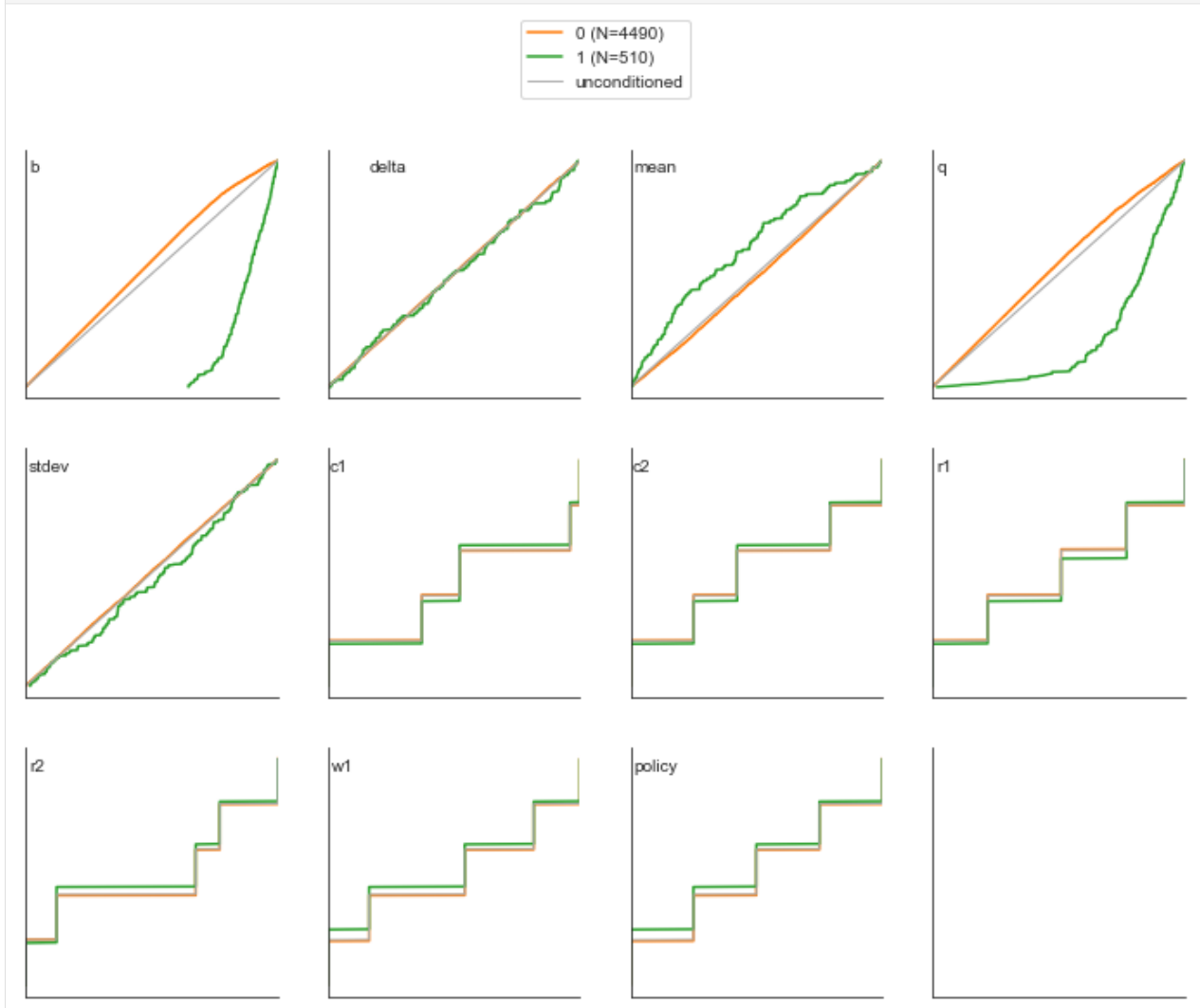
```
[13]: from ema_workbench.analysis import regional_sa
      from numpy.lib import recfunctions as rf

      sns.set_style("white")
```

(continues on next page)

(continued from previous page)

```
# model is the same across experiments
x = experiments.copy()
x = x.drop("model", axis=1)
y = outcomes["max_P"] < 0.8
fig = regional_sa.plot_cdfs(x, y)
sns.despine()
plt.show()
```



The above results clearly show that both B and Q are important. to a lesser extend, the mean also is relevant.

1.7.4 More advanced sampling techniques

The workbench can also be used for more advanced sampling techniques. To achieve this, it relies on [SALib](#). On the workbench side, the only change is to specify the sampler we want to use. Next, we can use SALib directly to perform the analysis. To help with this, the workbench provides a convenience function for generating the problem dict which SALib provides. The example below focusses on performing SOBOL on the uncertainties, but we could do the exact same thing with the levers instead. The only changes required would be to set `lever_sampling` instead of `uncertainty_sampling`, and get the SALib problem dict based on the levers.

```
[17]: from SALib.analyze import sobol
      from ema_workbench import Samplers
      from ema_workbench.em_framework.splib_samplers import get_SALib_problem

      with MultiprocessingEvaluator(model) as evaluator:
          sa_results = evaluator.perform_experiments(scenarios=1000, uncertainty_
          ↪sampling=Samplers.SOBOL)

      experiments, outcomes = sa_results

      problem = get_SALib_problem(model.uncertainties)
      Si = sobol.analyze(problem, outcomes["max_P"], calc_second_order=True, print_to_
      ↪console=False)

[MainProcess/INFO] pool started with 12 workers
/Users/jhkwakkel/opt/anaconda3/lib/python3.9/site-packages/SALib/sample/saltelli.py:94:
↪UserWarning:
    Convergence properties of the Sobol' sequence is only valid if
    `N` (1000) is equal to `2^n`.

    warnings.warn(msg)
[MainProcess/INFO] performing 12000 scenarios * 1 policies * 1 model(s) = 12000
↪experiments
100%| 12000/12000 [02:37<00:00, 76.17it/s]
[MainProcess/INFO] experiments finished
[MainProcess/INFO] terminating pool
```

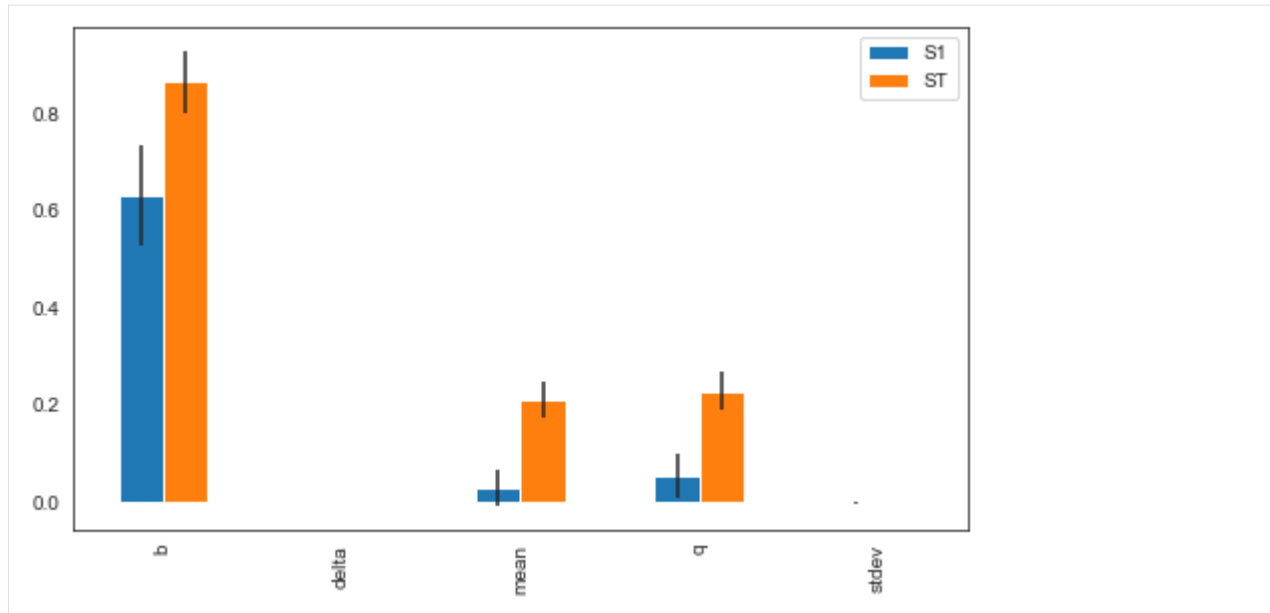
We have now completed the sobol analysis and have calculated the metrics. What remains is to visualize the metrics. Which can be done as shown below, focussing on `St` and `S1`. The error bars indicate the confidence intervals.

```
[18]: scores_filtered = {k: Si[k] for k in ["ST", "ST_conf", "S1", "S1_conf"]}
      Si_df = pd.DataFrame(scores_filtered, index=problem["names"])

      sns.set_style("white")
      fig, ax = plt.subplots(1)

      indices = Si_df[["S1", "ST"]]
      err = Si_df[["S1_conf", "ST_conf"]]

      indices.plot.bar(yerr=err.values.T, ax=ax)
      fig.set_size_inches(8, 6)
      fig.subplots_adjust(bottom=0.3)
      plt.show()
```



[]:

1.8 Directed search

This is the third tutorial in a series showcasing the functionality of the Exploratory Modeling workbench. Exploratory modeling entails investigating the way in which uncertainty and/or policy levers map to outcomes. To investigate these mappings, we can either use sampling based strategies (open exploration) or optimization based strategies (directed search).

In this tutorial, I will demonstrate in more detail how to use the workbench for directed search. We are using the same example as in the previous tutorials. When using optimization, it is critical that you specify for each Scalar Outcome the direction in which it should move. There are three possibilities: info which is ignored, maximize, and minimize. If the kind keyword argument is not specified, it defaults to info.

```
[1]: from ema_workbench import RealParameter, ScalarOutcome, Constant, Model
from dps_lake_model import lake_model

model = Model("lakeproblem", function=lake_model)

# specify uncertainties
model.uncertainties = [
    RealParameter("b", 0.1, 0.45),
    RealParameter("q", 2.0, 4.5),
    RealParameter("mean", 0.01, 0.05),
    RealParameter("stdev", 0.001, 0.005),
    RealParameter("delta", 0.93, 0.99),
]

# set levers
model.levers = [
    RealParameter("c1", -2, 2),
```

(continues on next page)

(continued from previous page)

```

    RealParameter("c2", -2, 2),
    RealParameter("r1", 0, 2),
    RealParameter("r2", 0, 2),
    RealParameter("w1", 0, 1),
]

# specify outcomes
model.outcomes = [
    ScalarOutcome("max_P", ScalarOutcome.MINIMIZE),
    ScalarOutcome("utility", ScalarOutcome.MAXIMIZE),
    ScalarOutcome("inertia", ScalarOutcome.MAXIMIZE),
    ScalarOutcome("reliability", ScalarOutcome.MAXIMIZE),
]

# override some of the defaults of the model
model.constants = [
    Constant("alpha", 0.41),
    Constant("nsamples", 150),
    Constant("myears", 100),
]

```

Using directed search with the `ema_workbench` requires `platypus-opt`. Please check the installation suggestions provided in the readme of the github repository. I personally either install from github directly

```
pip git+https://github.com/Project-Platypus/Platypus.git
```

or through pip

```
pip install platypus-opt
```

One note of caution: don't install `platypus`, but `platypus-opt`. There exists a python package on pip called `platypus`, but that is quite a different kind of library.

There are three ways in which we can use optimization in the workbench: 1. Search over the decision levers, conditional on a reference scenario 2. Search over the uncertain factors, conditional on a reference policy 3. Search over the decision levers given a set of scenarios

1.8.1 Search over levers

Directed search is most often used to search over the decision levers in order to find good candidate strategies. This is for example the first step in the [Many Objective Robust Decision Making process](#). This is straightforward to do with the workbench using the `optimize` method.

Note that I have kept the number of functional evaluations (`nfe`) very low. In real applications this should be substantially higher and be based on convergence considerations which are demonstrated below.

```

[2]: from ema_workbench import MultiprocessingEvaluator, ema_logging

ema_logging.log_to_stderr(ema_logging.INFO)

with MultiprocessingEvaluator(model) as evaluator:
    results = evaluator.optimize(nfe=250, searchover="levers", epsilons=[0.1] *
    ↪ len(model.outcomes))

```

```
[MainProcess/INFO] pool started with 10 workers
298it [00:03, 75.48it/s]
[MainProcess/INFO] optimization completed, found 5 solutions
[MainProcess/INFO] terminating pool
```

The results from `optimize` is a `DataFrame` with the decision variables and outcomes of interest. This dataframe contains both the decision variables (`c1-w1`) and the outcomes (`max_P-reliability`). Each row is thus a single unique solution.

Note also that there might be a difference between the specified number of nfe (250 in this case) and the actual number of nfe. The default algorithm is population based and the nfe-based stopping condition is only checked after evaluating an entire generation.

[3]: results

```
[3]:
```

	c1	c2	r1	r2	w1	max_P	utility \
0	-0.682559	0.026569	1.738125	1.035683	0.514804	2.283760	1.655153
1	-1.667796	-0.624652	0.914360	0.582130	0.243352	2.283627	1.778130
2	1.869933	1.822573	0.798277	0.886831	0.686324	1.961235	0.615122
3	0.442190	1.680330	0.989541	1.958483	0.979940	0.190612	0.523998
4	0.096997	0.329009	0.500270	0.946994	0.344109	0.092322	0.233183

	inertia	reliability
0	0.99	0.100133
1	0.99	0.070000
2	0.99	0.070000
3	0.99	1.000000
4	0.99	1.000000

Specifying constraints

It is possible to specify a constrained optimization problem. A model can have one or more constraints. A constraint can be applied to the model input parameters (both uncertainties and levers), and/or outcomes. A constraint is essentially a function that should return the distance from the feasibility threshold. The distance should be 0 if the constraint is met.

In the example below, we add a constraint saying that the value of the outcome `max_P` should be below 1. Given that this is a very simple constraint, I chose to use a lambda function to implement it. For more complicated constraints, you can also define your own function and pass that instead.

[4]: `from ema_workbench import Constraint`

```
constraints = [Constraint("max pollution", outcome_names="max_P", function=lambda x:
↳max(0, x - 1))]
```

[5]: `from ema_workbench import MultiprocessingEvaluator`
`from ema_workbench import ema_logging`

```
ema_logging.log_to_stderr(ema_logging.INFO)
```

```
with MultiprocessingEvaluator(model) as evaluator:
    results = evaluator.optimize(
        nfe=250, searchover="levers", epsilons=[0.1] * len(model.outcomes),
↳constraints=constraints
    )
```

```
[MainProcess/INFO] pool started with 10 workers
298it [00:03, 76.22it/s]
[MainProcess/INFO] optimization completed, found 2 solutions
[MainProcess/INFO] terminating pool
```

```
[6]: results
```

```
[6]:
```

	c1	c2	r1	r2	w1	max_P	utility \
0	0.517330	0.130095	1.648766	1.005694	0.616171	0.093374	0.232173
1	0.034839	0.763078	1.654288	1.185964	0.783527	0.190797	0.510802

	inertia	reliability
0	0.99	1.0
1	0.99	1.0

The new results with this simple constraint contains only a few solutions. This suggests that it is difficult to find solutions that are able to meet this constraint.

Seed analysis

Genetic algorithms rely on randomness in the search process. This implies that a single run of the algorithm does not tell you that much because the results might be due to randomness. It is thus best practice to always repeat the optimization for several different seeds and next merge the results across the different optimizations. This is supported by the workbench. We first need to run multiple optimization like this:

```
[7]: results = []
with MultiprocessingEvaluator(model) as evaluator:
    # we run 5 separate optimizations
    for _ in range(5):
        result = evaluator.optimize(
            nfe=5000, searchover="levers", epsilons=[0.05] * len(model.outcomes)
        )
        results.append(result)
```

```
[MainProcess/INFO] pool started with 10 workers
5036it [00:37, 133.58it/s]
[MainProcess/INFO] optimization completed, found 25 solutions
5058it [00:36, 140.03it/s]
[MainProcess/INFO] optimization completed, found 41 solutions
5049it [00:35, 140.43it/s]
[MainProcess/INFO] optimization completed, found 26 solutions
5054it [00:35, 140.87it/s]
[MainProcess/INFO] optimization completed, found 16 solutions
5052it [00:36, 140.17it/s]
[MainProcess/INFO] optimization completed, found 30 solutions
[MainProcess/INFO] terminating pool
```

We now have the results for each of the five different runs of the optimization. The next step is to combine these into a single comprehensive set. Since by default the workbench uses *-NSGAI*, it makes sense to also merge the results across the seeds using an *-nondominated* sort.

```
[11]: from ema_workbench.em_framework.optimization import epsilon_nondominated, to_problem
```

(continues on next page)

(continued from previous page)

```
problem = to_problem(model, searchover="levers")
epsilons = [0.05] * len(model.outcomes)
merged_archives = epsilon_nondominated(results, epsilons, problem)
```

```
[9]: merged_archives.shape
```

```
[9]: (34, 9)
```

As can be seen, the new merged archive contains 34 solutions. Which is quite a bit smaller than the sum of solutions across the 5 seeds. This implies that many of the solutions found in each seed land in the same -gridcell.

Tracking convergence

An important part of using many-objective evolutionary algorithms is to carefully monitor whether they have converged to the best possible solutions. Various different metrics can be used for this. Some of these metrics must be collected at runtime, such as [-progress](#). Others, like [hypervolume](#), are better calculated after the optimization has been completed. The metrics that are better calculated after the optimization require, however, storing the state of the archive over the course of the optimization. Both types of metrics are supported by the workbench.

```
[23]: from ema_workbench.em_framework.optimization import ArchiveLogger, EpsilonProgress
```

```
# we need to store our results for each seed
results = []
convergences = []

with MultiprocessingEvaluator(model) as evaluator:
    # we run again for 5 seeds
    for i in range(5):
        # we create 2 coverage tracker metrics
        # the archive logger writes the archive to disk for every x nfe
        # the epsilon progress tracks during runtime
        convergence_metrics = [
            ArchiveLogger(
                "./archives",
                [l.name for l in model.levers],
                [o.name for o in model.outcomes],
                base_filename=f"{i}.tar.gz",
            ),
            EpsilonProgress(),
        ]

        result, convergence = evaluator.optimize(
            nfe=50000,
            searchover="levers",
            epsilons=[0.05] * len(model.outcomes),
            convergence=convergence_metrics,
        )

        results.append(result)
        convergences.append(convergence)
```

```
[MainProcess/INFO] pool started with 10 workers
50061it [06:17, 132.46it/s]
[MainProcess/INFO] optimization completed, found 39 solutions
50117it [06:23, 130.56it/s]
[MainProcess/INFO] optimization completed, found 31 solutions
50081it [06:11, 134.92it/s]
[MainProcess/INFO] optimization completed, found 32 solutions
50085it [05:57, 139.98it/s]
[MainProcess/INFO] optimization completed, found 37 solutions
50062it [1:12:08, 11.57it/s]
[MainProcess/INFO] optimization completed, found 33 solutions
[MainProcess/INFO] terminating pool
```

Various metrics are provided by platypus. For details on these metrics see *e.g.*, [Zatarain-Salazar et al \(2016\)](#) and [Gupta et al \(2020\)](#) for hypervolume, generational distance and additive ϵ -indicator; [Hadka and Reed \(2012\)](#) for spacing; and [Hadka and Reed \(2013\)](#) for ϵ -pgress. To use these metrics, we first need to load the archives into memory. Next, these metrics need a set of platypus solutions, instead of the dataframes that the workbench has stored. Moreover, most of these metrics need a reference set. The reference set, typically, is the union of best solutions found across the seeds and next filtered using an -nondominated sort. All these steps are supported by the workbench as shown below.

```
[24]: all_archives = []

for i in range(5):
    archives = ArchiveLogger.load_archives(f"./archives/{i}.tar.gz")
    all_archives.append(archives)

[25]: from ema_workbench import (
        HypervolumeMetric,
        GenerationalDistanceMetric,
        EpsilonIndicatorMetric,
        InvertedGenerationalDistanceMetric,
        SpacingMetric,
    )
    from ema_workbench.em_framework.optimization import to_problem

    problem = to_problem(model, searchover="levers")

    reference_set = epsilon_nondominated(results, [0.05] * len(model.outcomes), problem)

    hv = HypervolumeMetric(reference_set, problem)
    gd = GenerationalDistanceMetric(reference_set, problem, d=1)
    ei = EpsilonIndicatorMetric(reference_set, problem)
    ig = InvertedGenerationalDistanceMetric(reference_set, problem, d=1)
    sm = SpacingMetric(problem)

    metrics_by_seed = []
    for archives in all_archives:
        metrics = []
        for nfe, archive in archives.items():
            scores = {
                "generational_distance": gd.calculate(archive),
                "hypervolume": hv.calculate(archive),
```

(continues on next page)

(continued from previous page)

```

        "epsilon_indicator": ei.calculate(archive),
        "inverted_gd": ig.calculate(archive),
        "spacing": sm.calculate(archive),
        "nfe": int(nfe),
    }
    metrics.append(scores)
metrics = pd.DataFrame.from_dict(metrics)

# sort metrics by number of function evaluations
metrics.sort_values(by="nfe", inplace=True)
metrics_by_seed.append(metrics)

```

Now that we have calculated all our metrics, we can visualize them using matplotlib.

```

[26]: sns.set_style("white")
fig, axes = plt.subplots(nrows=6, figsize=(8, 12), sharex=True)

ax1, ax2, ax3, ax4, ax5, ax6 = axes

for metrics, convergence in zip(metrics_by_seed, convergences):
    ax1.plot(metrics.nfe, metrics.hypervolume)
    ax1.set_ylabel("hypervolume")

    ax2.plot(convergence.nfe, convergence.epsilon_progress)
    ax2.set_ylabel("$\epsilon$ progress")

    ax3.plot(metrics.nfe, metrics.generational_distance)
    ax3.set_ylabel("generational distance")

    ax4.plot(metrics.nfe, metrics.epsilon_indicator)
    ax4.set_ylabel("epsilon indicator")

    ax5.plot(metrics.nfe, metrics.inverted_gd)
    ax5.set_ylabel("inverted generational distance")

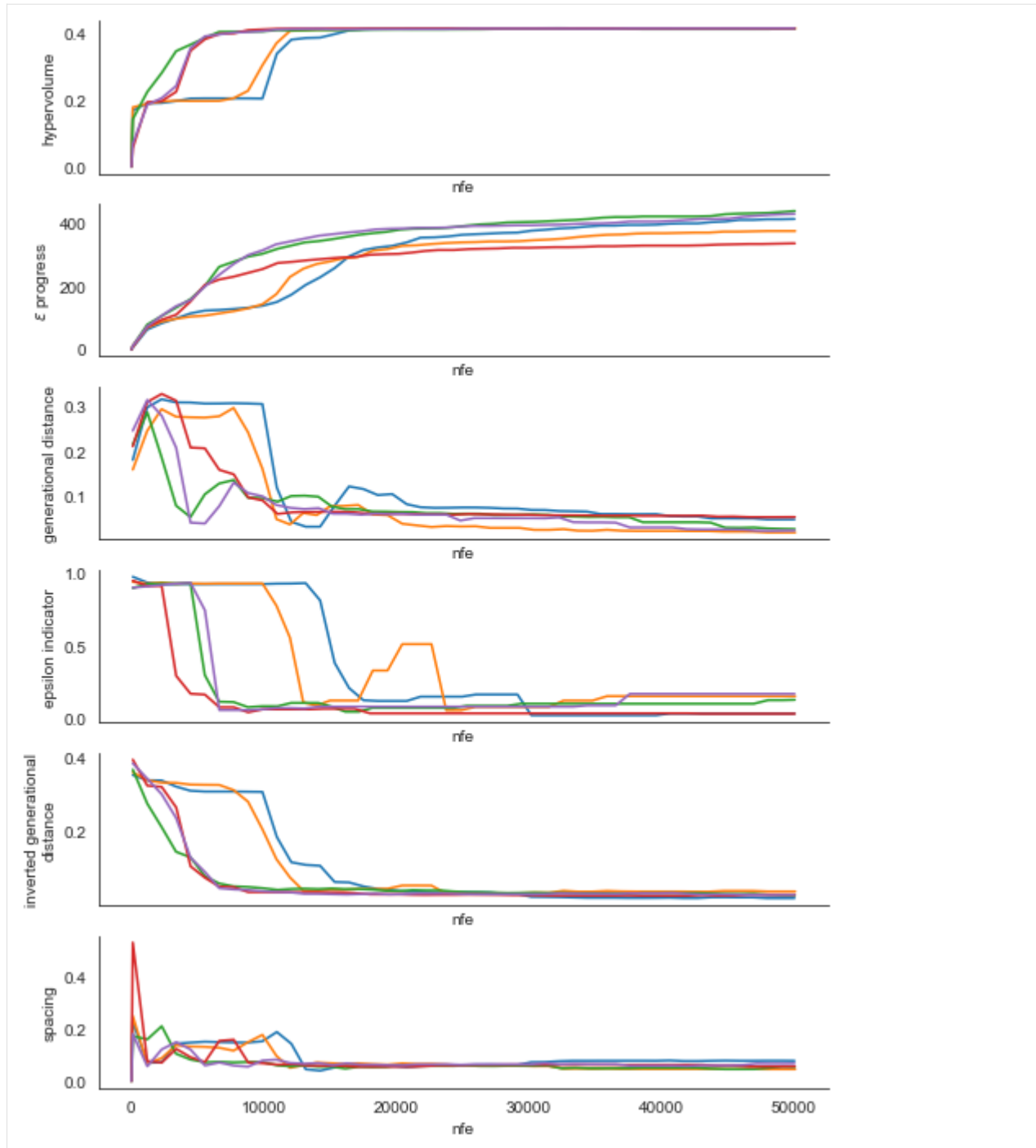
    ax6.plot(metrics.nfe, metrics.spacing)
    ax6.set_ylabel("spacing")

ax6.set_xlabel("nfe")

sns.despine(fig)

plt.show()

```



As we can see, for this problem across all the metrics and for each of the seeds we see that the line stabilizes which is indicative of convergence. The fact that the lines for each of the seeds stabilizes at roughly the same value moreover suggests that the seeds are converging to similar sets of solutions.

Changing the reference scenario

The workbench offers control over the reference scenario or policy under which you are performing the optimization. This makes it easy to apply multi-scenario MORDM (Watson & Kasprzyk, 2017; Eker & Kwakkel, 2018; Bartholomew & Kwakkel, 2020). Alternatively, you can also use it to change the policy for which you are applying worst case scenario discovery (see below).

```
reference = Scenario('reference', b=0.4, q=2, mean=0.02, stdev=0.01)

with MultiprocessingEvaluator(lake_model) as evaluator:
    results = evaluator.optimize(searchover='levers', nfe=1000,
                                epsilon=[0.1, ]*len(lake_model.outcomes),
                                reference=reference)
```

1.8.2 Search over uncertainties: worst case discovery

Up till now, the focus has been on applying search to find promising candidate strategies. That is, we search through the lever space. However, there might also be good reasons to search through the uncertainty space. For example to search for worst case scenarios (Halim et al, 2015). This is easily achieved as shown below. We change the kind attribute on each outcome so that we search for the worst outcome and specify that we would like to search over the uncertainties instead of the levers.

Any of the foregoing additions such as constraints or convergence works as shown above. Note that if you would like to to change the reference policy, reference should be a Policy object rather than a Scenario object.

```
[7]: # change outcomes so direction is undesirable
minimize = ScalarOutcome.MINIMIZE
maximize = ScalarOutcome.MAXIMIZE

for outcome in model.outcomes:
    if outcome.kind == minimize:
        outcome.kind = maximize
    else:
        outcome.kind = minimize

with MultiprocessingEvaluator(model) as evaluator:
    results = evaluator.optimize(
        nfe=1000, searchover="uncertainties", epsilon=[0.1] * len(model.outcomes)
    )
```

```
[MainProcess/INFO] pool started with 10 workers
1090it [00:09, 116.02it/s]
[MainProcess/INFO] optimization completed, found 8 solutions
[MainProcess/INFO] terminating pool
```

1.8.3 Parallel coordinate plots

The workbench comes with support for making parallel axis plots through the `parcoords` module. This module offers a parallel axes object on which we can plot data.

The typical workflow is to first instantiate this parallel axes object given a pandas dataframe with the upper and lower limits for each axes. Next, one or more datasets can be plotted on this axes. Any dataframe passed to the plot method will be normalized using the limits passed first. We can also invert any of the axes to ensure that the desirable direction is the same for all axes.

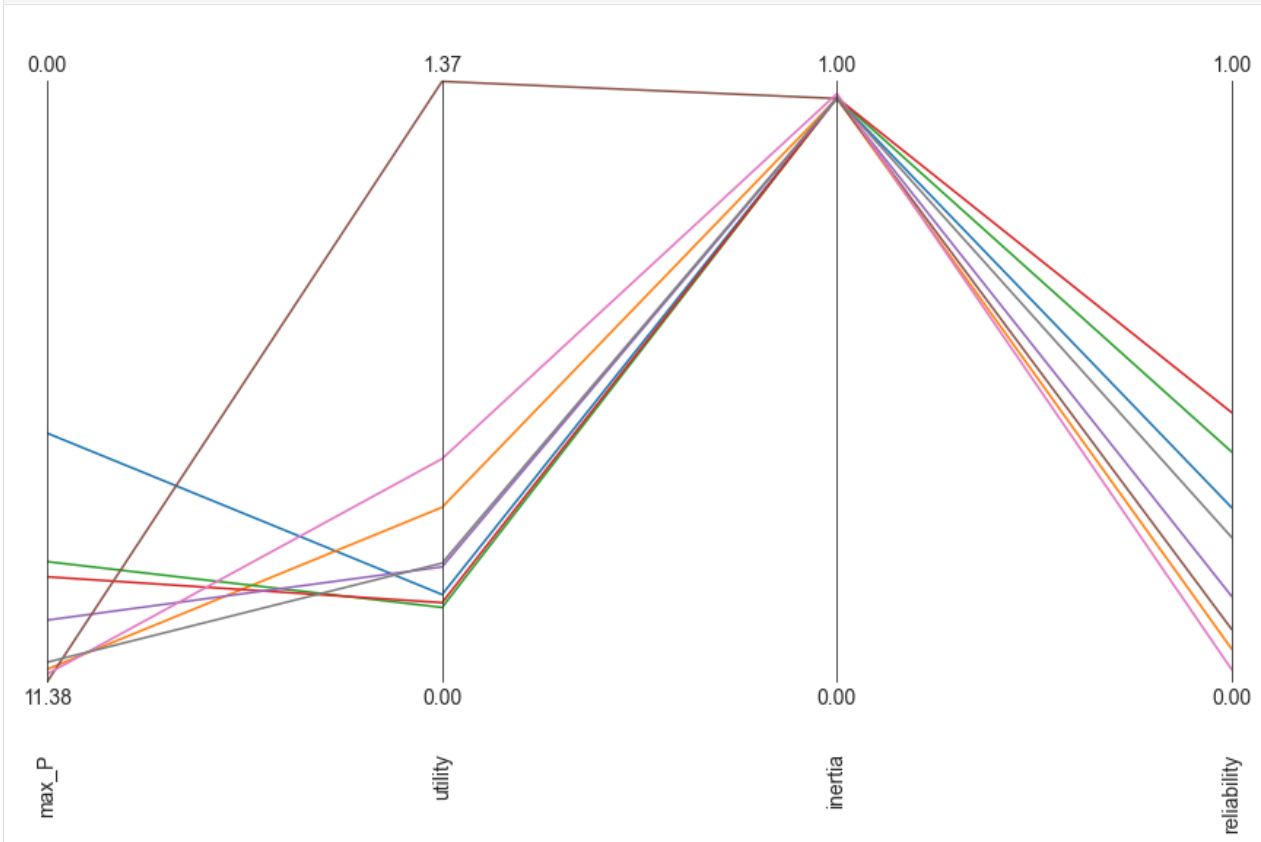
```
[9]: from ema_workbench.analysis import parcoords

data = results.loc[:, [o.name for o in model.outcomes]]

# get the minimum and maximum values as present in the dataframe
limits = parcoords.get_limits(data)

# we know that the lowest possible value for all objectives is 0
limits.loc[0, ["utility", "inertia", "reliability", "max_P"]] = 0
# inertia and reliability are defined on unit interval, so their theoretical maximum is 1
limits.loc[1, ["inertia", "reliability"]] = 1

paraxes = parcoords.ParallelAxes(limits)
paraxes.plot(data)
paraxes.invert_axis("max_P")
plt.show()
```



The above parallel coordinate plot shows the results from the worst case discovery. The worst possible solution would

be a straight line at the bottom. A key insight from this result is that there seems to be a trade-off between max_P and utility. This can be inferred from the crossing lines between these two axes.

1.8.4 Robust Search

In the foregoing, we have been using optimization over levers or uncertainties, while assuming a reference scenario or policy. However, we can also formulate a robust many objective optimization problem (Kwakkel et al. 2015; Bartholomew et al. 2020), where we are going to search over the levers for solutions that have robust performance over a set of scenarios. To do this with the workbench, there are several steps that one has to take.

First, we need to specify our robustness metrics. A robustness metric takes as input the performance of a candidate policy over a set of scenarios and returns a single robustness score. For a more in depth overview, see McPhail et al. (2018). In case of the workbench, we can use the `ScalarOutcome` class for this. We need to specify the name of the robustness metric a function that takes as input a numpy array and returns a single number, and the model outcome to which this function should be applied.

Below, we use a percentile based robustness metric, which we apply to each model outcome.

```
[20]: import functools

# our robustness functions
percentile10 = functools.partial(np.percentile, q=10)
percentile90 = functools.partial(np.percentile, q=90)

# convenient short hands
MAXIMIZE = ScalarOutcome.MAXIMIZE
MINIMIZE = ScalarOutcome.MINIMIZE

robustnes_functions = [
    ScalarOutcome(
        "90th percentile max_p", kind=MINIMIZE, variable_name="max_P",
        ↪function=percentile90
    ),
    ScalarOutcome(
        "10th percentile reliability",
        kind=MAXIMIZE,
        variable_name="reliability",
        function=percentile10,
    ),
    ScalarOutcome(
        "10th percentile inertia", kind=MAXIMIZE, variable_name="inertia",
        ↪function=percentile10
    ),
    ScalarOutcome(
        "10th percentile utility", kind=MAXIMIZE, variable_name="utility",
        ↪function=percentile10
    ),
]
```

Next, we have to generate the scenarios we want to use. Below we generate 10 scenarios, which we will keep fixed over the optimization. The exact number of scenarios to use is to be established through trial and error. Typically it involves balancing computational costs of more scenarios, with the stability of the robustness metric over the number of scenarios

```
[23]: from ema_workbench.em_framework import sample_uncertainties

n_scenarios = 10
scenarios = sample_uncertainties(model, n_scenarios)
```

With the robustness metrics specified, and the scenarios, sampled, we can now perform robust many-objective optimization. Below is the code that one would run. Note that this is computationally very expensive since each candidate solution is going to be run for ten scenarios before we can calculate the robustness for each outcome of interest.

```
[24]: from ema_workbench.em_framework import ArchiveLogger

nfe = int(5e4)
with MultiprocessingEvaluator(model) as evaluator:
    robust_results = evaluator.robust_optimize(
        robustnes_functions,
        scenarios,
        nfe=nfe,
        epsilons=[
            0.025,
        ]
        * len(robustnes_functions),
    )

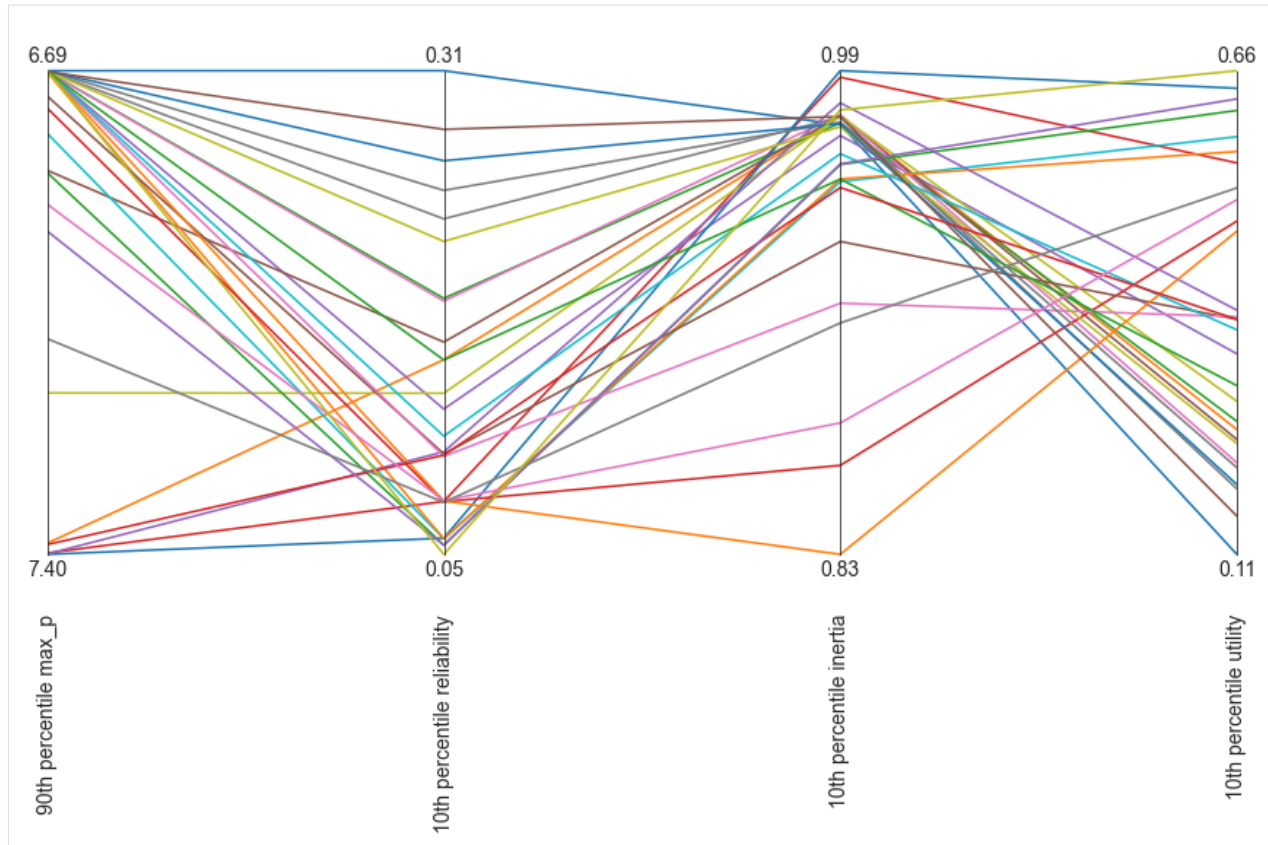
[MainProcess/INFO] pool started with 10 workers
50039it [2:10:57, 6.37it/s]
[MainProcess/INFO] optimization completed, found 29 solutions
[MainProcess/INFO] terminating pool
```

We can now again visualize the results using a parallel coordinate plot. Note that we are visualizing the robustness scores rather than the outcomes of interest as specified in the underlying model.

```
[29]: from ema_workbench.analysis import parcoords

data = robust_results.loc[:, [o.name for o in robustnes_functions]]
limits = parcoords.get_limits(data)

paraxes = parcoords.ParallelAxes(limits)
paraxes.plot(data)
paraxes.invert_axis("90th percentile max_p")
plt.show()
```

What we can see in this parallel axis plot is that there is a clear tradeoff between robust high reliability and robust low maximum pollution. Likewise with inertia and utility. This can be seen from the crossing lines between these respective axes.

[]:

1.9 Examples

1.9.1 Output space exploration

Output space exploration is a form of optimization based on novelty search. In the workbench, it relies on the optimization functionality. You can use output space exploration by passing an instance of either `OutputSpaceExploration` or `AutoAdaptiveOutputSpaceExploration` as algorithm to `evaluator.optimize`. The fact that output space exploration uses the optimization functionality also implies that we can track convergence in a similar manner. Epsilon progress is defined identically, but evidently other metrics such as hypervolume might not be applicable in the context of output space exploration.

The difference between `OutputSpaceExploration` and `AutoAdaptiveOutputSpaceExploration` is in the evolutionary operators. `AutoAdaptiveOutputSpaceExploration` uses auto adaptive operator selection as implemented in the BORG MOEA, while `OutputSpaceExploration` by default uses Simulated Binary crossover with polynomial mutation. Injection of new solutions is handled through auto adaptive population sizing and periodically starting with a new population if search is stalling. Below, examples are given of how to use both algorithms, as well as a quick visualization of the convergence dynamics.

For this example, we are using a stylized case study frequently used to develop and test decision making under deep

uncertainty methods: the shallow lake problem. In this problem, a city has to decide on the amount of pollution they are going to put into a shallow lake per year. The city gets benefits from polluting the lake, but if an unknown threshold is crossed, the lake permanently shifts to an undesirable polluted state. For further details on this case study, see for example [Quinn et al, 2017](#) and [Bartholomew et al, 2021](#).

```
[1]: from lake_model_outputspace_exploration import lake_problem
```

```
from ema_workbench import (
    Model,
    RealParameter,
    ScalarOutcome,
    Constant,
    ema_logging,
    MultiprocessingEvaluator,
    Policy,
    SequentialEvaluator,
    OutputSpaceExploration,
)
```

```
[2]: ema_logging.log_to_stderr(ema_logging.INFO)
```

```
# instantiate the model
lake_model = Model("lakeproblem", function=lake_problem)
lake_model.time_horizon = 100

# specify uncertainties
lake_model.uncertainties = [
    RealParameter("b", 0.1, 0.45),
    RealParameter("q", 2.0, 4.5),
    RealParameter("mean", 0.01, 0.05),
    RealParameter("stdev", 0.001, 0.005),
    RealParameter("delta", 0.93, 0.99),
]

# set levers, one for each time step
lake_model.levers = [RealParameter(str(i), 0, 0.1) for i in range(lake_model.time_
↪ horizon)]

# specify outcomes
# output space exploration

lake_model.outcomes = [
    ScalarOutcome("max_P", kind=ScalarOutcome.MAXIMIZE),
    ScalarOutcome("utility", kind=ScalarOutcome.MAXIMIZE),
    ScalarOutcome("inertia", kind=ScalarOutcome.MAXIMIZE),
    ScalarOutcome("reliability", kind=ScalarOutcome.MAXIMIZE),
]

# override some of the defaults of the model
lake_model.constants = [Constant("alpha", 0.41), Constant("nsamples", 150)]
```

Above, we have setup the lake problem, specified the uncertainties, policy levers, outcomes of interest and (optionally) some constants. We are now ready to run output space exploration on this model. For this, we use the default optimization functionality of the world bank, but pass the `OutputSpaceExploration` class. Below, we are running

output space exploration over the uncertainties, which implies we need to pass a reference policy. We also rerun the algorithm for 5 different seeds. We can track convergence of the outputspace exploration by tracking ϵ -progress.

The `grid_spec` keyword argument, which is specific to output space exploration specifies the grid structure we are imposing on the output space. There must be a tuple for each outcome of interest. Each tuple specifies the minimum value, the maximum value and the size of the grid cell on this outcome dimension.

```
[3]: from ema_workbench.em_framework.optimization import EpsilonProgress

# generate some default policy
reference = Policy("nopolicy", **{l.name: 0.02 for l in lake_model.levers})
n_seeds = 5

convergences = []

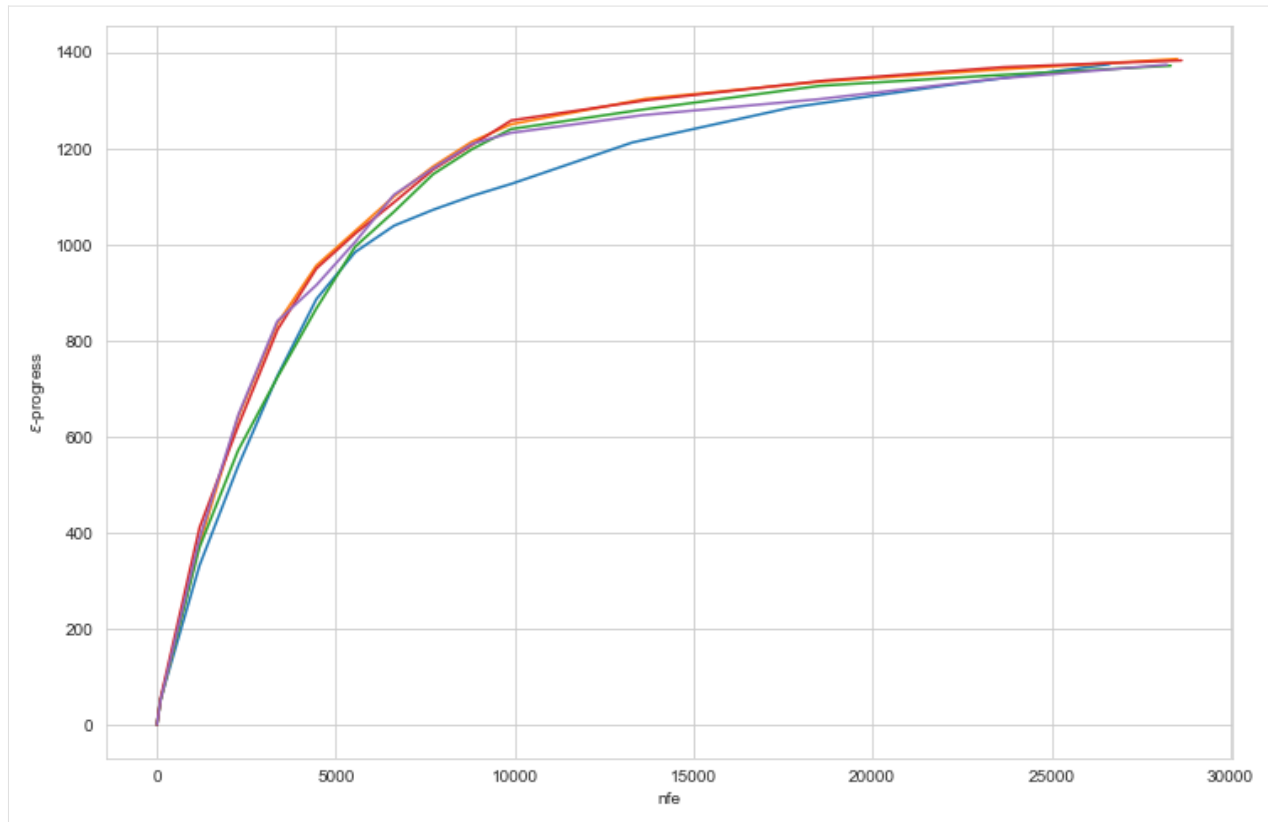
with MultiprocessingEvaluator(lake_model) as evaluator:
    for _ in range(n_seeds):
        convergence_metrics = [
            EpsilonProgress(),
        ]
        res, convergence = evaluator.optimize(
            algorithm=OutputSpaceExploration,
            grid_spec=[(0, 12, 0.5), (0, 0.6, 0.05), (0, 1, 0.1), (0, 1, 0.1)],
            nfe=25000,
            searchover="uncertainties",
            reference=reference,
            convergence=convergence_metrics,
        )
        convergences.append(convergence)
```

```
[MainProcess/INFO] pool started with 10 workers
26625it [01:40, 265.19it/s]
[MainProcess/INFO] optimization completed, found 1375 solutions
28514it [01:47, 265.52it/s]
[MainProcess/INFO] optimization completed, found 1386 solutions
28336it [01:49, 259.12it/s]
[MainProcess/INFO] optimization completed, found 1372 solutions
28638it [01:49, 262.33it/s]
[MainProcess/INFO] optimization completed, found 1383 solutions
28227it [01:48, 259.60it/s]
[MainProcess/INFO] optimization completed, found 1374 solutions
[MainProcess/INFO] terminating pool
```

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

sns.set_style("whitegrid")
fig, ax = plt.subplots()
for convergence in convergences:
    ax.plot(convergence.nfe, convergence.epsilon_progress)

ax.set_xlabel("nfe")
ax.set_ylabel("$\epsilon$-progress")
plt.show()
```



The figure above shows the ϵ -progress per nfe for each of the five random seeds. As can be seen, the algorithm converges to essentially the same amount of epsilon progress across the seeds. This suggests that the algorithm has converged and thus found all possible output grid cells.

auto adaptive operator selection

The foregoing example used the default version of the output space exploration algorithm. This algorithm uses Simulated Binary Crossover (SBX) with Polynomial Mutation (PM) as its evolutionary operators. However, it is conceivable that for some problems other evolutionary operators might be more suitable. You can pass your own combination of operators if desired using [platypus](#). For general use, however, it can sometimes be easier to let the algorithm itself figure out which operators to use. To support this, we also provide an `AutoAdaptiveOutputSpaceExploration` algorithm. This algorithm uses Auto Adaptive Operator selection as also used in the [BORG](#) algorithm. Note that this algorithm is limited to `RealParameters` only.

```
[5]: from ema_workbench.em_framework.outputspace_exploration import \
    AutoAdaptiveOutputSpaceExploration
    from ema_workbench.em_framework.optimization import OperatorProbabilities

    convergences = []

    with MultiprocessingEvaluator(lake_model) as evaluator:
        for _ in range(5):
            convergence_metrics = [
                EpsilonProgress(),
```

(continues on next page)

(continued from previous page)

```

        OperatorProbabilities("SBX", 0),
        OperatorProbabilities("PCX", 1),
        OperatorProbabilities("DE", 2),
        OperatorProbabilities("UNDX", 3),
        OperatorProbabilities("SPX", 4),
        OperatorProbabilities("UM", 5),
    ]
    res, convergence = evaluator.optimize(
        algorithm=AutoAdaptiveOutputSpaceExploration,
        grid_spec=[(0, 12, 0.5), (0, 0.6, 0.05), (0, 1, 0.1), (0, 1, 0.1)],
        nfe=25000,
        searchover="uncertainties",
        reference=reference,
        variator=None,
        convergence=convergence_metrics,
    )
    convergences.append(convergence)

```

```

[MainProcess/INFO] pool started with 10 workers
29556it [01:56, 252.85it/s]
[MainProcess/INFO] optimization completed, found 1437 solutions
30067it [01:58, 253.31it/s]
[MainProcess/INFO] optimization completed, found 1443 solutions
29957it [01:55, 259.12it/s]
[MainProcess/INFO] optimization completed, found 1458 solutions
30091it [01:57, 256.60it/s]
[MainProcess/INFO] optimization completed, found 1451 solutions
30167it [01:58, 254.13it/s]
[MainProcess/INFO] optimization completed, found 1443 solutions
[MainProcess/INFO] terminating pool

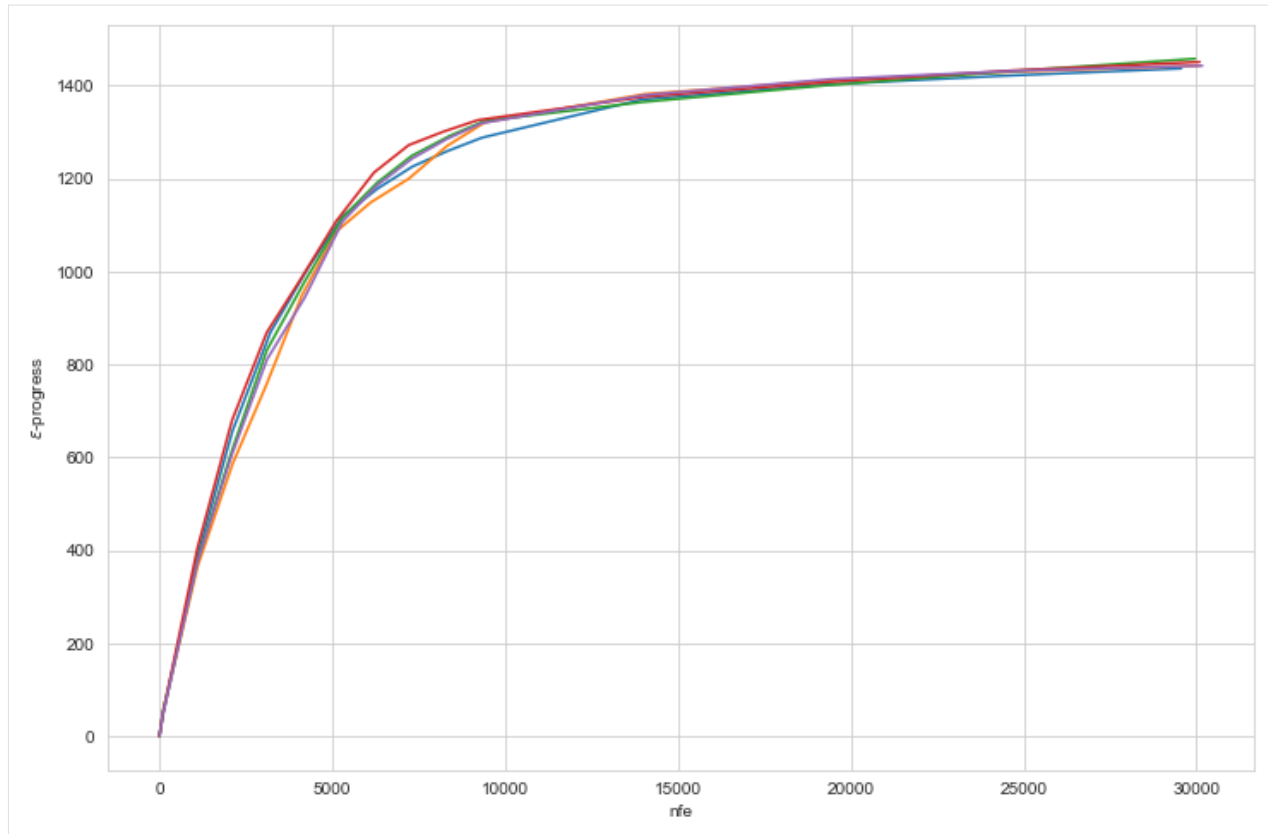
```

```

[7]: sns.set_style("whitegrid")
    fig, ax = plt.subplots()
    for convergence in convergences:
        ax.plot(convergence.nfe, convergence.epsilon_progress)

    ax.set_xlabel("nfe")
    ax.set_ylabel("$\epsilon$-progress")
    plt.show()

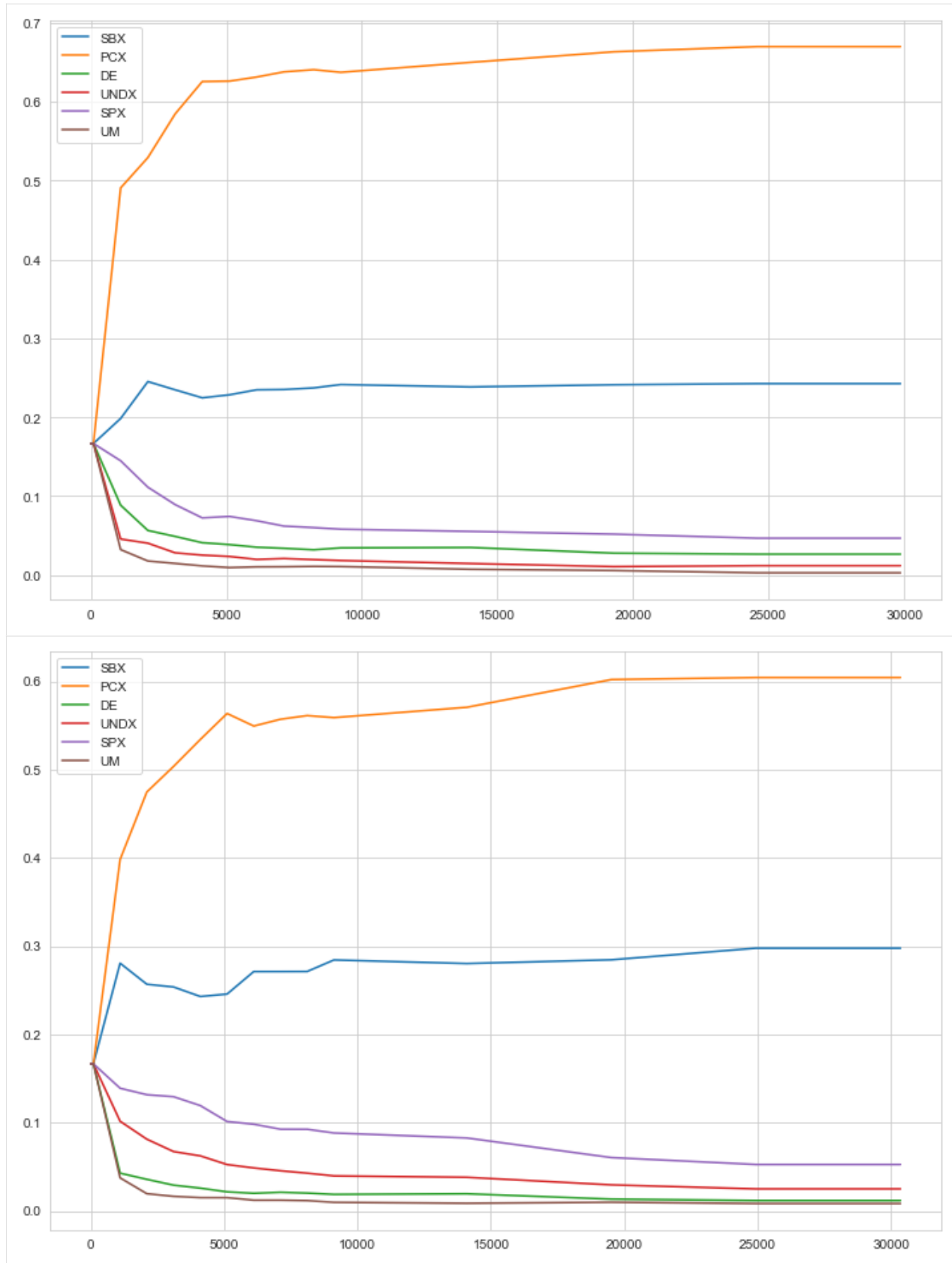
```

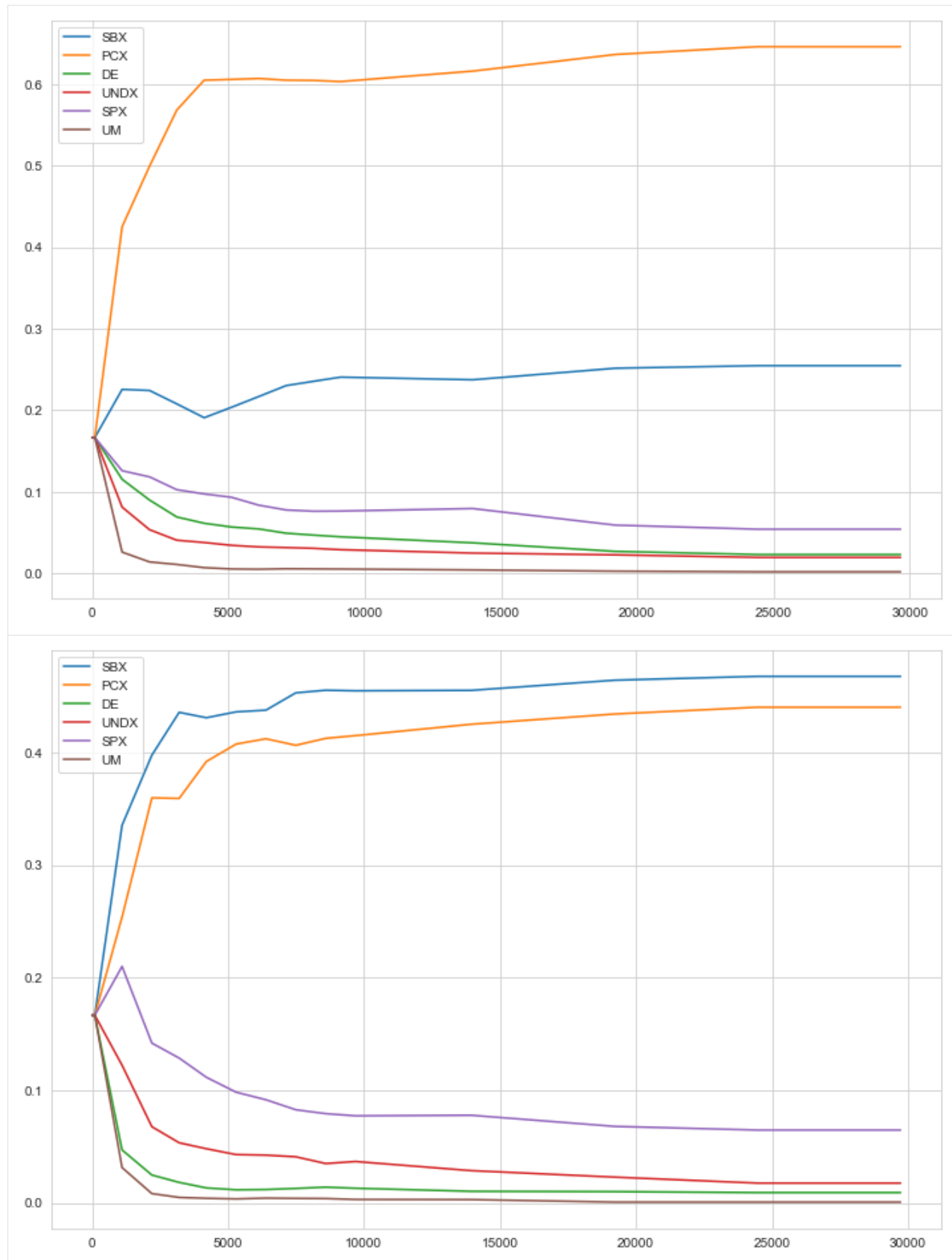


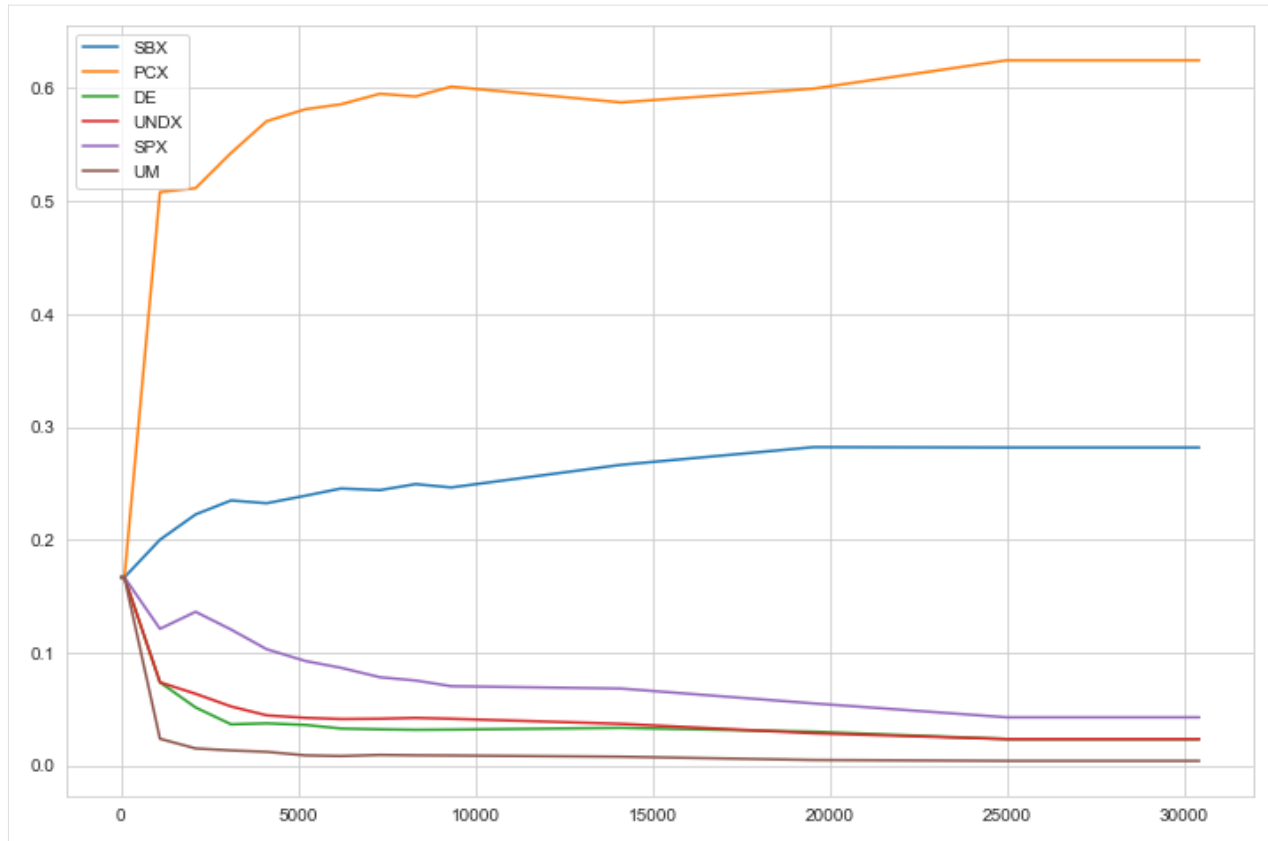
Above, you see the ϵ -convergence for the auto adaptive operator selection version of the outputspace algorithm. Like the normal version, it converges to essentially the same number across the different seeds. Note also that the ϵ -progress is slightly higher, and also the total number of identified solutions (see log messages) is higher. That suggests that for even for a relatively simple problem, there is value in using the auto adaptive operator selection.

In case of `AutoAdaptiveOutputSpaceExploration` it can sometimes also be revealing to check the dynamics of the operators over the evolution. This is shown below separately for each seed. For the meaning of the abbreviations, check the [BORG algorithm](#).

```
[8]: for convergence in convergences:
    fig, ax = plt.subplots()
    ax.plot(convergence.nfe, convergence.SBX, label="SBX")
    ax.plot(convergence.nfe, convergence.PCX, label="PCX")
    ax.plot(convergence.nfe, convergence.DE, label="DE")
    ax.plot(convergence.nfe, convergence.UNDX, label="UNDX")
    ax.plot(convergence.nfe, convergence.SPX, label="SPX")
    ax.plot(convergence.nfe, convergence.UM, label="UM")
    ax.legend()
plt.show()
```







LHS

for comparison, let's also generate a Latin Hypercube Sample and compare the results. Since the output space exploration return close to 1500 solutions, we use 1500 for LHS as well.

```
[8]: with MultiprocessingEvaluator(lake_model) as evaluator:
      experiments, outcomes = evaluator.perform_experiments(scenarios=1500,
      policies=reference)
```

```
[MainProcess/INFO] pool started with 10 workers
[MainProcess/INFO] performing 1500 scenarios * 1 policies * 1 model(s) = 1500 experiments
100%| 1500/1500 [00:05<00:00, 293.09it/s]
[MainProcess/INFO] experiments finished
[MainProcess/INFO] terminating pool
```

comparison

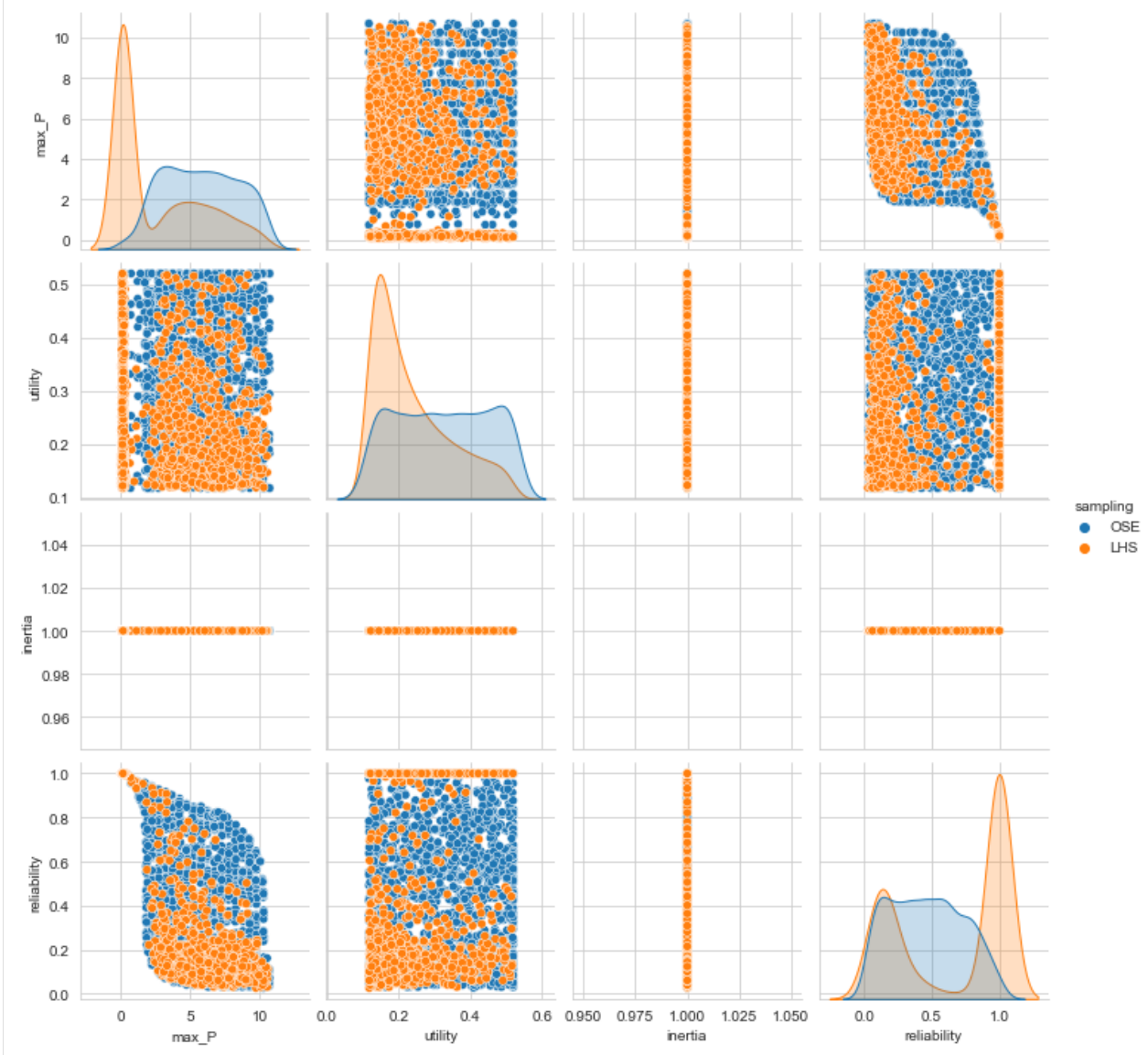
Below we compare the LHS with the last seed of the autoadaptive algorithm. Remember, the purpose of output space exploration is to identify the kinds of behavior that a model can show given a set of uncertain parameters. So, we are creating a pairwise scatter plot of the outcomes for both output space exploration (in blue) and the latin hypercube sampling (in orange).

```
[9]: outcomes = pd.DataFrame(outcomes)
      outcomes["sampling"] = "LHS"
```

```
[10]: ose = res.iloc[:, 5:].copy()
      ose["sampling"] = "OSE"
```

```
[11]: data = pd.concat([ose, outcomes], axis=0, ignore_index=True)
```

```
[12]: # sns.pairplot(data, hue='sampling')
      sns.pairplot(data, hue="sampling", vars=data.columns[0:4])
      plt.show()
```



As you can clearly see, the output space exploration algorithm has uncovered behaviors that were not seen in the 1500 sample LHS.

```
[ ]:
```

1.9.2 Performing Scenario Discovery in Python

The purpose of example is to demonstrate how one can do scenario discovery in python. I will demonstrate how we can perform both PRIM in an interactive way, as well as briefly show how to use CART, which is also available in the exploratory modeling workbench. There is ample literature on both CART and PRIM and their relative merits for use in scenario discovery. So I won't be discussing that here in any detail.

In order to demonstrate the use of the exploratory modeling workbench for scenario discovery, I am using a published example. I am using the data used in the original article by Ben Bryant and Rob Lempert where they first introduced 2010. Ben Bryant kindly made this data available and allowed me to share it. The data comes as a csv file. We can import the data easily using pandas. columns 2 up to and including 10 contain the experimental design, while the classification is presented in column 15

This example is a slightly updated version of a blog post on <https://waterprogramming.wordpress.com/2015/08/05/scenario-discovery-in-python/>

```
[1]: import pandas as pd

data = pd.read_csv("../data/bryant et al 2010 data.csv", index_col=False)
x = data.iloc[:, 2:11]
y = data.iloc[:, 15].values
```

The exploratory modeling workbench comes with a separate analysis package. This analysis package contains prim. So let's import prim. The workbench also has its own logging functionality. We can turn this on to get some more insight into prim while it is running.

```
[2]: from ema_workbench.analysis import prim
from ema_workbench.util import ema_logging

ema_logging.log_to_stderr(ema_logging.INFO);
```

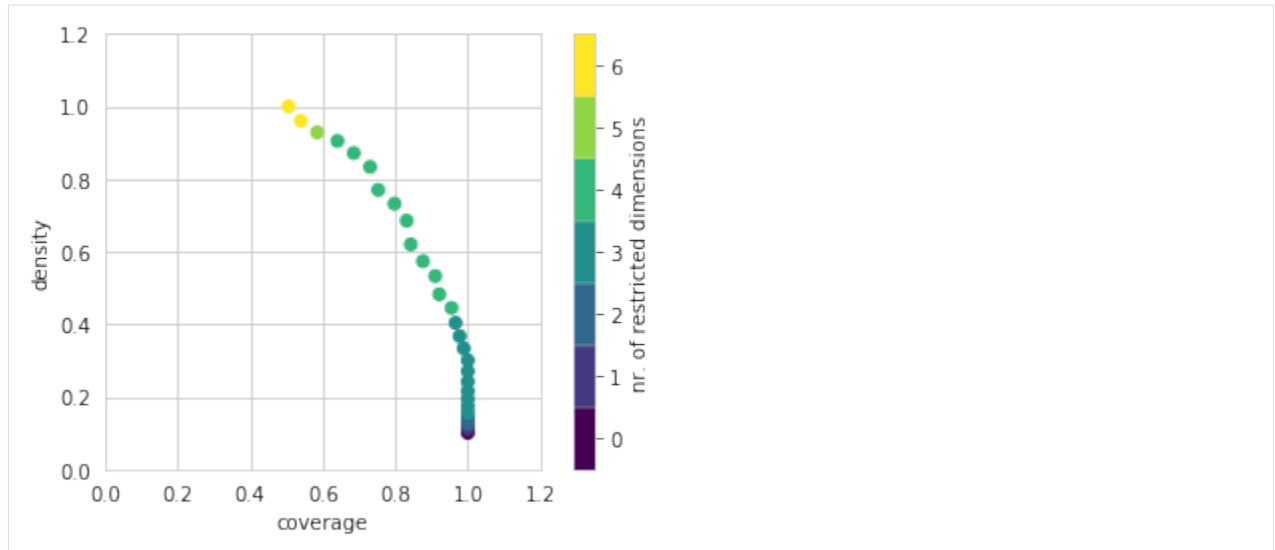
Next, we need to instantiate the prim algorithm. To mimic the original work of Ben Bryant and Rob Lempert, we set the peeling alpha to 0.1. The peeling alpha determines how much data is peeled off in each iteration of the algorithm. The lower the value, the less data is removed in each iteration. The minimum coverage threshold that a box should meet is set to 0.8. Next, we can use the instantiated algorithm to find a first box.

```
[3]: prim_alg = prim.Prim(x, y, threshold=0.8, peel_alpha=0.1)
box1 = prim_alg.find_box()

[MainProcess/INFO] 882 points remaining, containing 89 cases of interest
[MainProcess/INFO] mean: 1.0, mass: 0.05102040816326531, coverage: 0.5056179775280899,
density: 1.0 restricted_dimensions: 6
```

Let's investigate this first box in some detail. A first thing to look at is the trade off between coverage and density. The box has a convenience function for this called `show_tradeoff`.

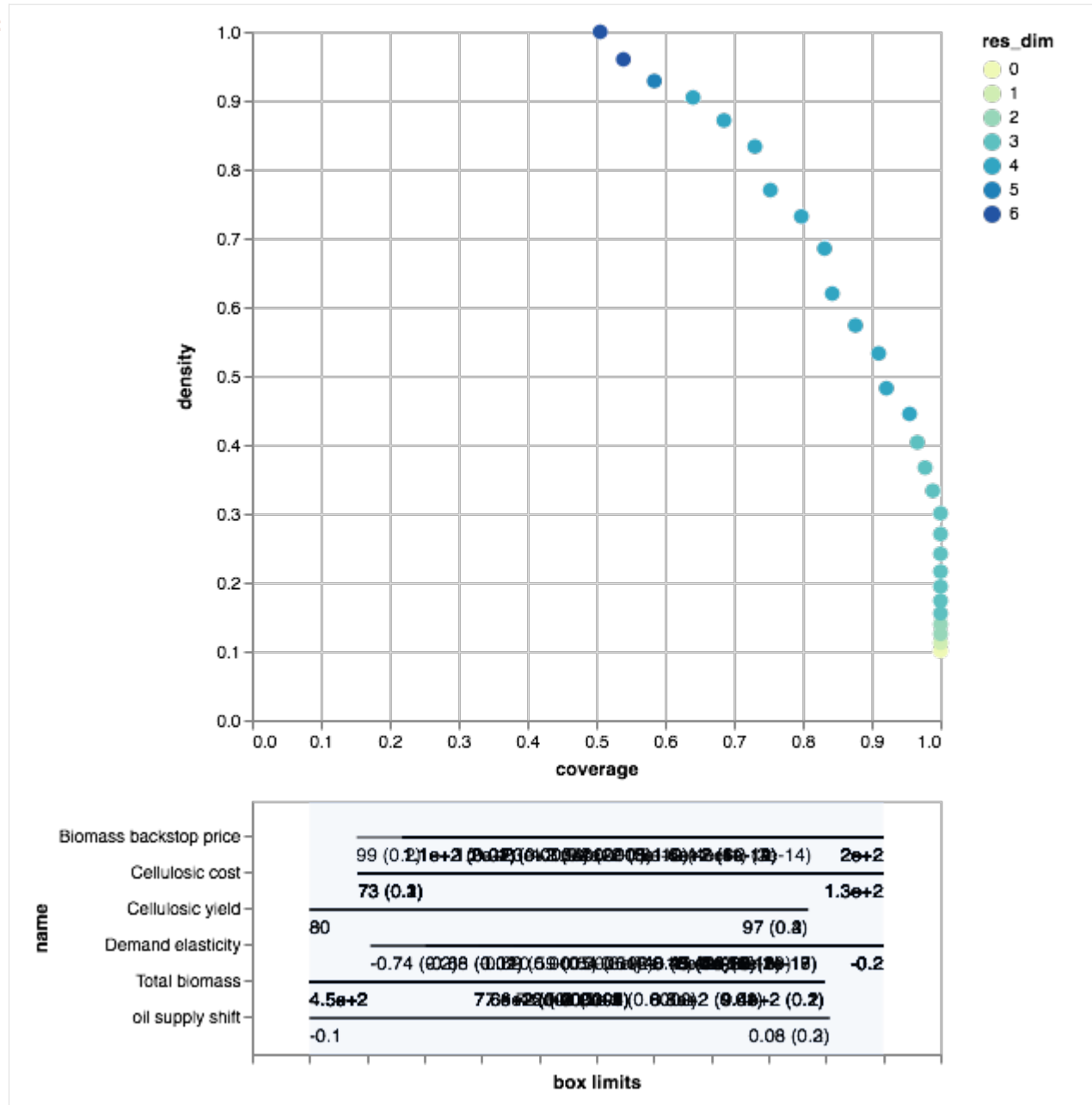
```
[4]: box1.show_tradeoff()
plt.show()
```



Since we are doing this analysis in a notebook, we can take advantage of the interactivity that the browser offers. A relatively recent addition to the python ecosystem is the library [altair](#). Altair can be used to create interactive plots for use in a browser. Altair is an optional dependency for the workbench. If available, we can create the following visual.

```
[5]: box1.inspect_tradeoff()
```

[5]:



Here we can interactively explore the boxes associated with each point in the density coverage trade-off. It also offers mouse overs for the various points on the trade off curve. Given the id of each point, we can also use the workbench to manually inspect the peeling trajectory. Following Bryant & Lempert, we inspect box 21.

[6]: `box1.resample(21)`

```
[MainProcess/INFO] resample 0
[MainProcess/INFO] resample 1
[MainProcess/INFO] resample 2
[MainProcess/INFO] resample 3
[MainProcess/INFO] resample 4
[MainProcess/INFO] resample 5
```

(continues on next page)

(continued from previous page)

```
[MainProcess/INFO] resample 6
[MainProcess/INFO] resample 7
[MainProcess/INFO] resample 8
[MainProcess/INFO] resample 9
```

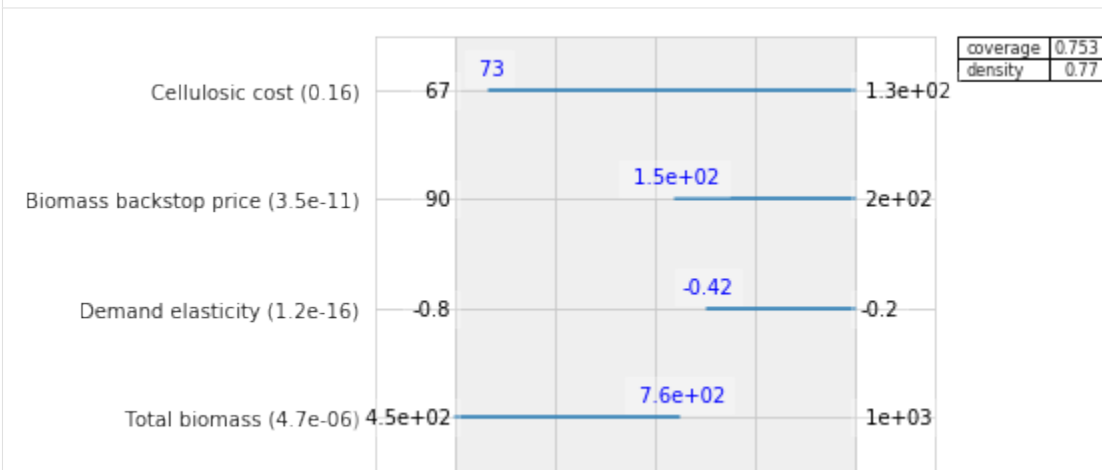
```
[6]:
```

	reproduce coverage	reproduce density
Total biomass	100.0	100.0
Demand elasticity	100.0	100.0
Biomass backstop price	100.0	100.0
Cellulosic cost	90.0	90.0
Cellulosic yield	20.0	20.0
Electricity coproduction	20.0	20.0
Feedstock distribution	0.0	0.0
Oil elasticity	0.0	0.0
oil supply shift	0.0	0.0

```
[7]: box1.inspect(21)
box1.inspect(21, style="graph")
plt.show()
```

```
coverage    0.752809
density     0.770115
id          21
mass        0.0986395
mean        0.770115
res_dim     4
Name: 21, dtype: object
```

	box 21		qp values
	min	max	
Total biomass	450.000000	755.799988	$[-1.0, 4.716968553178765e-06]$
Demand elasticity	-0.422000	-0.202000	$[1.1849299115762218e-16, -1.0]$
Biomass backstop price	150.049995	199.600006	$[3.515112530263049e-11, -1.0]$
Cellulosic cost	72.650002	133.699997	$[0.15741333528927348, -1.0]$

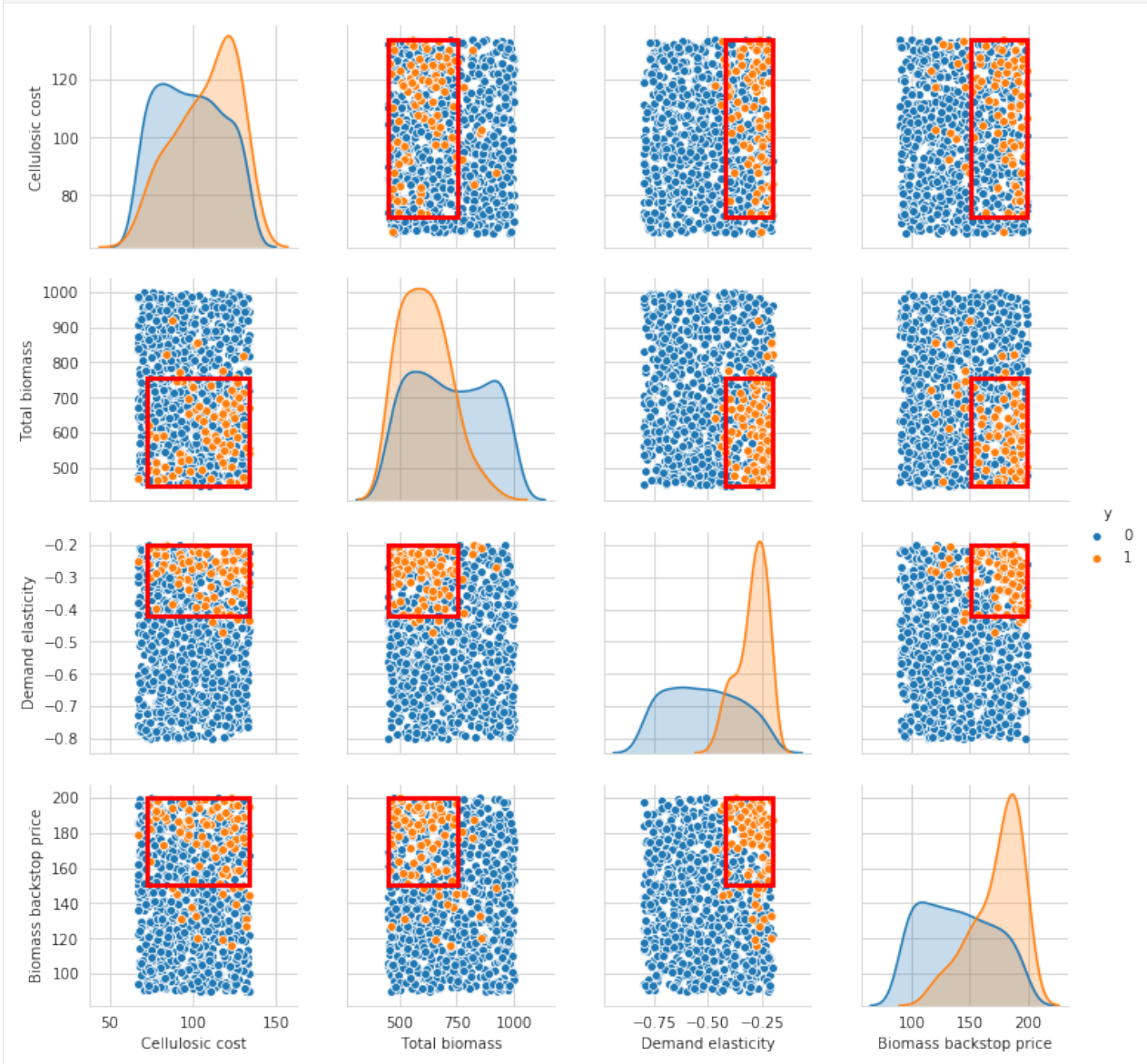


If one were to do a detailed comparison with the results reported in the original article, one would see small numerical differences. These differences arise out of subtle differences in implementation. The most important difference is that

the exploratory modeling workbench uses a custom objective function inside prim which is different from the one used in the scenario discovery toolkit. Other differences have to do with details about the hill climbing optimization that is used in prim, and in particular how ties are handled in selected the next step. The differences between the two implementations are only numerical, and don't affect the overarching conclusions drawn from the analysis.

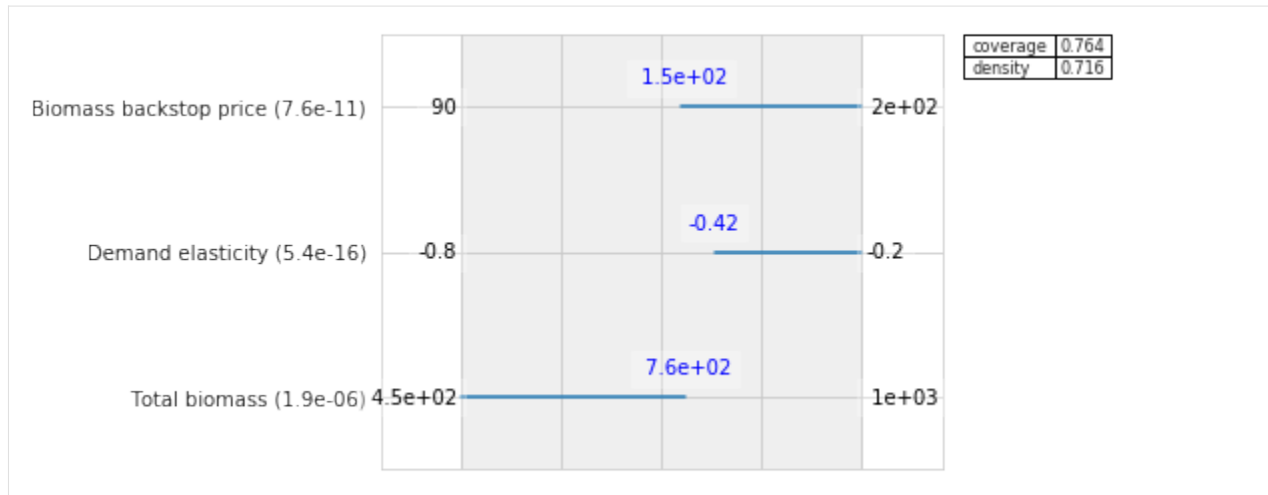
Let's select this 21 box, and get a more detailed view of what the box looks like. Following Bryant et al., we can use scatter plots for this.

```
[8]: box1.select(21)
fig = box1.show_pairs_scatter(21)
plt.show()
```



Because the last restriction is not significant, we can choose to drop this restriction from the box.

```
[9]: box1.drop_restriction("Cellulosic cost")
box1.inspect(style="graph")
plt.show()
```



We have now found a first box that explains over 75% of the cases of interest. Let's see if we can find a second box that explains the remainder of the cases.

```
[10]: box2 = prim_alg.find_box()
```

```
[MainProcess/INFO] 787 points remaining, containing 21 cases of interest
[MainProcess/INFO] box does not meet threshold criteria, value is 0.3541666666666667,
↳ returning dump box
```

As we can see, we are unable to find a second box. The best coverage we can achieve is 0.35, which is well below the specified 0.8 threshold. Let's look at the final overall results from interactively fitting PRIM to the data. For this, we can use to convenience functions that transform the stats and boxes to pandas data frames.

```
[11]: prim_alg.stats_to_dataframe()
```

```
[11]:
```

	coverage	density	mass	res_dim
box 1	0.764045	0.715789	0.10771	3
box 2	0.235955	0.026684	0.89229	0

```
[12]: prim_alg_boxes_to_dataframe()
```

```
[12]:
```

	box 1		box 2	
	min	max	min	max
Demand elasticity	-0.422000	-0.202000	-0.8	-0.202000
Biomass backstop price	150.049995	199.600006	90.0	199.600006
Total biomass	450.000000	755.799988	450.0	997.799988

CART

The way of interacting with CART is quite similar to how we setup the prim analysis. We import cart from the analysis package. We instantiate the algorithm, and next fit CART to the data. This is done via the build_tree method.

```
[13]: from ema_workbench.analysis import cart
```

```
cart_alg = cart.CART(x, y, 0.05)
cart_alg.build_tree()
```



```

/Users/jhkwakkel/miniconda3/lib/python3.6/importlib/_bootstrap.py:219: ImportWarning: can
↳ 't resolve package from __spec__ or __package__, falling back on __name__ and __path__
  return f(*args, **kwargs)
/Users/jhkwakkel/miniconda3/lib/python3.6/importlib/_bootstrap.py:219: ImportWarning: can
↳ 't resolve package from __spec__ or __package__, falling back on __name__ and __path__
  return f(*args, **kwargs)
/Users/jhkwakkel/miniconda3/lib/python3.6/importlib/_bootstrap.py:219: ImportWarning: can
↳ 't resolve package from __spec__ or __package__, falling back on __name__ and __path__
  return f(*args, **kwargs)

```

Now that we have trained CART on the data, we can investigate its results. Just like PRIM, we can use `stats_to_dataframe` and `boxes_to_dataframe` to get an overview.

```
[14]: cart_alg.stats_to_dataframe()
```

```
[14]:
```

	coverage	density	mass	res	dim
box 1	0.011236	0.021739	0.052154		2
box 2	0.000000	0.000000	0.546485		2
box 3	0.000000	0.000000	0.103175		2
box 4	0.044944	0.090909	0.049887		2
box 5	0.224719	0.434783	0.052154		2
box 6	0.112360	0.227273	0.049887		3
box 7	0.000000	0.000000	0.051020		3
box 8	0.606742	0.642857	0.095238		2

```
[15]: cart_alg.boxes_to_dataframe()
```

```
[15]:
```

	box 1		box 2		box 3 \
	min	max	min	max	min
Cellulosic yield	80.0	81.649998	81.649998	99.900002	80.000
Demand elasticity	-0.8	-0.439000	-0.800000	-0.439000	-0.439
Biomass backstop price	90.0	199.600006	90.000000	199.600006	90.000

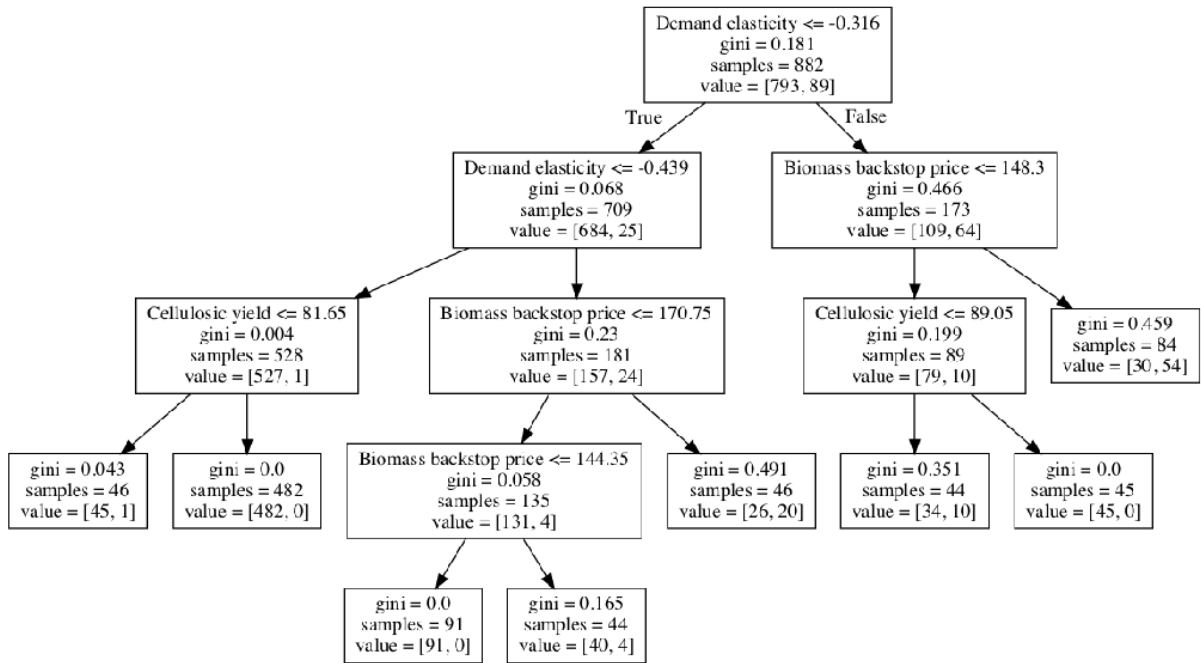
	box 4		box 5 \	
	max	min	max	min
Cellulosic yield	99.900002	80.000000	99.900002	80.000
Demand elasticity	-0.316500	-0.439000	-0.316500	-0.439
Biomass backstop price	144.350006	144.350006	170.750000	170.750

	box 6		box 7 \	
	max	min	max	min
Cellulosic yield	99.900002	80.0000	89.049999	89.049999
Demand elasticity	-0.316500	-0.3165	-0.202000	-0.316500
Biomass backstop price	199.600006	90.0000	148.300003	90.000000

	box 8	
	max	min
Cellulosic yield	99.900002	80.000000
Demand elasticity	-0.202000	-0.316500
Biomass backstop price	148.300003	148.300003

Alternatively, we might want to look at the classification tree directly. For this, we can use the `show_tree` method.

```
[16]: fig = cart_alg.show_tree()
fig.set_size_inches((18, 12))
plt.show()
```



```
[ ]:
```

1.9.3 Vadere EMA connector demo

This notebook demonstrates the setup of a Vadere model connection. It uses an example Vadere model, made with Vadere 2.1. Before running the code yourself, please acquire a copy of Vadere from [the official Vadere website](#). For more information on the use of the EMA Workbench, please refer to the [official EMA Workbench documentation](#). This demo is based on the code provided on the documentation pages.

Step 1: imports

The first step is to import the needed modules. This depends on the use and the type of analysis that is intended. The most important one here is the VadereModel from the model connectors. As said, please refer for more information on this to the [official EMA Workbench documentation](#).

```
[ ]: from ema_workbench import (
    perform_experiments,
    RealParameter,
    ema_logging,
    MultiprocessingEvaluator,
    Samplers,
    ScalarOutcome,
    IntegerParameter,
    RealParameter,
```

(continues on next page)

(continued from previous page)

```
)
from ema_workbench.connectors.vadere import VadereModel
import pandas as pd
import numpy as np
```

Step 2: Setting up the model

In this demonstration, we use a simple Vadere model example. The model includes two sources and one target, and spawns a number of pedestrians every second. The model uses the OSM locomotion implementation, and collects the following:

- The average evacuation time -> stored in evacuationTime.txt
- The average speed of all pedestrians each time step -> stored in speed.csv

Note that the EMA connector assumes .txt files for scalar outcomes and .csv files for time series outcomes. Please use these file types, and declare them explicitly in processor_files as shown below.

```
[2]: # This model saves scalar results to a density.txt and speed.txt file.
# Please acquire your own copy of Vadere, and place the vadere-console.jar in your model/
# scenarios directory
# Note that the vadere model files and the console.jar should always be placed in a
# separate wd as the python runfile
model = VadereModel(
    "demoModel",
    vadere_jar="vadere-console.jar",
    processor_files=["evacuationTime.txt", "speed.csv"],
    model_file="demo.scenario",
    wd="models/vadereModel/scenarios/",
)
```

```
[3]: # set the number of replications to handle model stochasticity
model.replications = 5
```

Note that for specifying model uncertainties (and potential levers), the Vadere model class can change any variable present in the model file (Vadere scenario). To realize this, an exact location to the variable of interest in the Vadere scenario file has to be specified. Vadere scenario files follow a nested dictionary structure. Therefore, the exact location of the variable should be passed in a list of argumentes, passed as one string.

See the example below, that variates the following:

- The number of spawned pedestrians from both sources
- The mean of the free flow speed distribution

```
[4]: model.uncertainties = [
    IntegerParameter(
        name="spawnNumberA",
        lower_bound=1,
        upper_bound=50,
        variable_name=['("scenario", "topography", "sources", 0, "spawnNumber")'],
    ),
    IntegerParameter(
        name="spawnNumberB",
```

(continues on next page)

(continued from previous page)

```

        lower_bound=1,
        upper_bound=50,
        variable_name=[('scenario', 'topography', 'sources', 1, 'spawnNumber')],
    ),
    RealParameter(
        name="FreeFlowSpeed",
        lower_bound=0.7,
        upper_bound=1.5,
        variable_name=[
            ('scenario', 'topography', 'attributesPedestrian', 'speedDistributionMean')
        ],
    ),
]

```

The model outcomes can be specified by passing the exact name as present in the output file (speed.txt here). The naming convention depends on the used Vadere data processors, but usually follows the name + id of the processor. When in doubt, it is advised to do a demo Vadere run using the Vadere software and to inspect the generated output files. Note that we take the mean of the outcomes here, since we specified multiple replications. Note that we now only focus on scalar outcomes. See the end of this demo for timeseries.

```

[5]: model.outcomes = [
    ScalarOutcome(name="evacuationTime", variable_name="meanEvacuationTime-PID8",
    ↪function=np.mean)
]

```

Step 3: Performing experiments

The last step is to perform experiment with the Vadere model. Both sequential runs as runs in parallel are supported. Note however that a Vadere run can use a lot of RAM, and using all available CPU cores can lead to performance issues in some cases. Additionally, one Vadere run is by default already multithreaded. It is therefore recommended to not use the full set of available processes.

```

[6]: # enable EMA logging
ema_logging.log_to_stderr(ema_logging.INFO)

```

```

[6]: <Logger EMA (DEBUG)>

```

```

[7]: # run in sequential 2 experiments
results_sequential = perform_experiments(model, scenarios=2, uncertainty_
    ↪sampling=Samplers.LHS)

[MainProcess/INFO] performing 2 scenarios * 1 policies * 1 model(s) = 2 experiments
0%|                                     | 0/2 [00:00<?, ?it/
    ↪s][MainProcess/INFO] performing experiments sequentially
100%|| 2/2 [01:03<00:00, 31.75s/it]
[MainProcess/INFO] experiments finished

```

```

[8]: # run 6 experiments in parallel
with MultiprocessingEvaluator(model, n_processes=4) as evaluator:
    experiments, outcomes = evaluator.perform_experiments(
        scenarios=4, uncertainty_sampling=Samplers.LHS
    )

```

```
[MainProcess/INFO] pool started with 4 workers
[MainProcess/INFO] performing 4 scenarios * 1 policies * 1 model(s) = 4 experiments
100%| 4/4 [00:37<00:00, 9.39s/it]
[MainProcess/INFO] experiments finished
[MainProcess/INFO] terminating pool
[ForkPoolWorker-1/INFO] finalizing
[ForkPoolWorker-4/INFO] finalizing
[ForkPoolWorker-3/INFO] finalizing
[ForkPoolWorker-2/INFO] finalizing
```

Inspect the results

we can now look at the results directly. For more extensive exploratory analysis methods, please refer to the [official EMA Workbench documentation](#).

```
[9]: experiments.head()
```

```
[9]:   spawnNumberA  spawnNumberB  FreeFlowSpeed  scenario  policy    model
0           25.0           33.0         0.838960         2    None  demoModel
1           46.0           19.0         1.413804         3    None  demoModel
2           28.0           11.0         0.942684         4    None  demoModel
3            2.0           50.0         1.111305         5    None  demoModel
```

```
[10]: pd.DataFrame(outcomes).head()
```

```
[10]:   evacuationTime
0         14.268966
1          9.990154
2         11.673846
3         11.155385
```

Using time series data

It is additionally possible to specify time series output data. See below for an example on how to do this.

Note that this example now uses a different model class, since we do not want to average multiple scalar outcomes but are rather interested in a time series outcome. Hence, the `SingleReplicationVadereModel` is used. Now, the `timeStep` and average speeds are collected every step.

```
[11]: from ema_workbench import TimeSeriesOutcome
      from ema_workbench.connectors.vadere import SingleReplicationVadereModel
```

```
[12]: # This model saves scalar results to a density.txt and speed.txt file.
      # Please acquire your own copy of Vadere, and place the vadere-console.jar in your model/
      ↪scenarios directory
      # Note that the vadere model files should always be placed in a separate wd as the ↪
      ↪python runfile
      model = SingleReplicationVadereModel(
          "demoModel",
          vadere_jar="vadere-console.jar",
          processor_files=["evacuationTime.txt", "speed.csv"],
```

(continues on next page)

(continued from previous page)

```

model_file="demo.scenario",
wd="models/vadereModel/scenarios/",
)

```

```

[13]: model.uncertainties = [
    IntegerParameter(
        name="spawnNumberA",
        lower_bound=1,
        upper_bound=50,
        variable_name=['("scenario", "topography", "sources", 0, "spawnNumber")'],
    ),
    IntegerParameter(
        name="spawnNumberB",
        lower_bound=1,
        upper_bound=50,
        variable_name=['("scenario", "topography", "sources", 1, "spawnNumber")'],
    ),
    RealParameter(
        name="FreeFlowSpeed",
        lower_bound=0.7,
        upper_bound=1.5,
        variable_name=[
            ('("scenario", "topography", "attributesPedestrian", "speedDistributionMean")'
            ],
    ),
]

```

```

[14]: model.outcomes = [TimeSeriesOutcome(name="speedTime", variable_name="areaSpeed-PID5")]

```

```

[15]: # run 4 experiments in parallel
with MultiprocessingEvaluator(model, n_processes=4) as evaluator:
    experiments, outcomes = evaluator.perform_experiments(
        scenarios=4, uncertainty_sampling=Samplers.LHS
    )

```

```

[MainProcess/INFO] pool started with 4 workers
[MainProcess/INFO] performing 4 scenarios * 1 policies * 1 model(s) = 4 experiments
100%| 4/4 [00:07<00:00, 1.90s/it]
[MainProcess/INFO] experiments finished
[MainProcess/INFO] terminating pool
[ForkPoolWorker-8/INFO] finalizing
[ForkPoolWorker-5/INFO] finalizing
[ForkPoolWorker-6/INFO] finalizing
[ForkPoolWorker-7/INFO] finalizing

```

```

[16]: from ema_workbench.analysis.plotting import lines

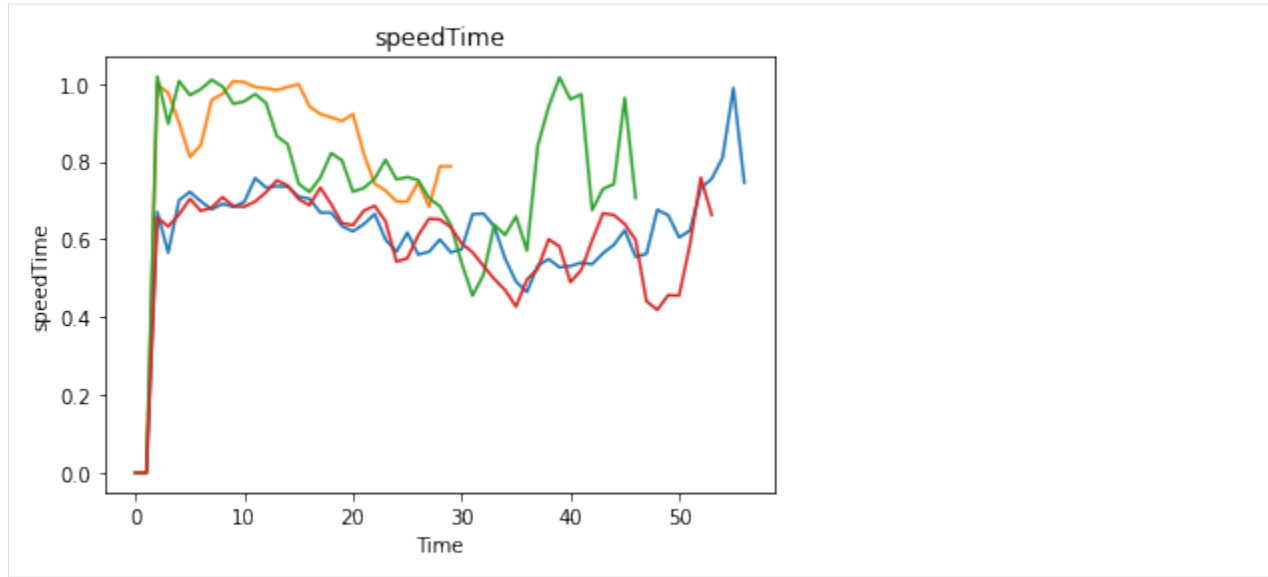
lines(experiments, outcomes, outcomes_to_show="speedTime")

```

```

[16]: (<Figure size 432x288 with 1 Axes>,
      {'speedTime': <AxesSubplot:title={'center': 'speedTime'}, xlabel='Time', ylabel=
      ↪ 'speedTime'>})

```



1.9.4 example_eijgenraam.py

```

1  """
2
3  """
4
5  # Created on 12 Mar 2020
6  #
7  # .. codeauthor:: jhkwakkel
8
9  import bisect
10 import functools
11 import math
12 import operator
13
14 import numpy as np
15 import scipy as sp
16
17 from ema_workbench import Model, RealParameter, ScalarOutcome, ema_logging,
18     MultiprocessingEvaluator
19
20 ##=====
21 ## Implement the model described by Eijgenraam et al. (2012)
22 ## code taken from Rhodium Eijgenraam example
23 ##-----
24
25 # Parameters pulled from the paper describing each dike ring
26 params = ("c", "b", "lam", "alpha", "eta", "zeta", "V0", "P0", "max_Pf")
27 raw_data = {
28     10: (16.6939, 0.6258, 0.0014, 0.033027, 0.320, 0.003774, 1564.9, 0.00044, 1 / 2000),
29     11: (42.6200, 1.7068, 0.0000, 0.032000, 0.320, 0.003469, 1700.1, 0.00117, 1 / 2000),
30     15: (125.6422, 1.1268, 0.0098, 0.050200, 0.760, 0.003764, 11810.4, 0.00137, 1 /

```

(continues on next page)

(continued from previous page)

```

↪2000),
16: (324.6287, 2.1304, 0.0100, 0.057400, 0.760, 0.002032, 22656.5, 0.00110, 1 / ↪
↪2000),
22: (154.4388, 0.9325, 0.0066, 0.070000, 0.620, 0.002893, 9641.1, 0.00055, 1 / 2000),
23: (26.4653, 0.5250, 0.0034, 0.053400, 0.800, 0.002031, 61.6, 0.00137, 1 / 2000),
24: (71.6923, 1.0750, 0.0059, 0.043900, 1.060, 0.003733, 2706.4, 0.00188, 1 / 2000),
35: (49.7384, 0.6888, 0.0088, 0.036000, 1.060, 0.004105, 4534.7, 0.00196, 1 / 2000),
38: (24.3404, 0.7000, 0.0040, 0.025321, 0.412, 0.004153, 3062.6, 0.00171, 1 / 1250),
41: (58.8110, 0.9250, 0.0033, 0.025321, 0.422, 0.002749, 10013.1, 0.00171, 1 / 1250),
42: (21.8254, 0.4625, 0.0019, 0.026194, 0.442, 0.001241, 1090.8, 0.00171, 1 / 1250),
43: (340.5081, 4.2975, 0.0043, 0.025321, 0.448, 0.002043, 19767.6, 0.00171, 1 / ↪
↪1250),
44: (24.0977, 0.7300, 0.0054, 0.031651, 0.316, 0.003485, 37596.3, 0.00033, 1 / 1250),
45: (3.4375, 0.1375, 0.0069, 0.033027, 0.320, 0.002397, 10421.2, 0.00016, 1 / 1250),
47: (8.7813, 0.3513, 0.0026, 0.029000, 0.358, 0.003257, 1369.0, 0.00171, 1 / 1250),
48: (35.6250, 1.4250, 0.0063, 0.023019, 0.496, 0.003076, 7046.4, 0.00171, 1 / 1250),
49: (20.0000, 0.8000, 0.0046, 0.034529, 0.304, 0.003744, 823.3, 0.00171, 1 / 1250),
50: (8.1250, 0.3250, 0.0000, 0.033027, 0.320, 0.004033, 2118.5, 0.00171, 1 / 1250),
51: (15.0000, 0.6000, 0.0071, 0.036173, 0.294, 0.004315, 570.4, 0.00171, 1 / 1250),
52: (49.2200, 1.6075, 0.0047, 0.036173, 0.304, 0.001716, 4025.6, 0.00171, 1 / 1250),
53: (69.4565, 1.1625, 0.0028, 0.031651, 0.336, 0.002700, 9819.5, 0.00171, 1 / 1250),
}
data = {i: dict(zip(params, raw_data[i])) for i in raw_data}

# Set the ring we are analyzing
ring = 15
max_failure_probability = data[ring]["max_Pf"]

# Compute the investment cost to increase the dike height
def exponential_investment_cost(
    u, # increase in dike height
    h0, # original height of the dike
    c, # constant from Table 1
    b, # constant from Table 1
    lam,
): # constant from Table 1
    if u == 0:
        return 0
    else:
        return (c + b * u) * math.exp(lam * (h0 + u))

def eijsenraam_model(
    X1,
    X2,
    X3,
    X4,
    X5,
    X6,
    T1,
    T2,

```

(continues on next page)

(continued from previous page)

```

79     T3,
80     T4,
81     T5,
82     T6,
83     T=300,
84     P0=data[ring]["P0"],
85     V0=data[ring]["V0"],
86     alpha=data[ring]["alpha"],
87     delta=0.04,
88     eta=data[ring]["eta"],
89     gamma=0.035,
90     rho=0.015,
91     zeta=data[ring]["zeta"],
92     c=data[ring]["c"],
93     b=data[ring]["b"],
94     lam=data[ring]["lam"],
95 ):
96     """Python implementation of the Eijgenraam model
97
98     Params
99     -----
100     Xs : list
101         list of dike heightenings
102     Ts : list
103         time of dike heightenings
104     T : int, optional
105         planning horizon
106     P0 : <>, optional
107         constant from Table 1
108     V0 : <>, optional
109         constant from Table 1
110     alpha : <>, optional
111         constant from Table 1
112     delta : float, optional
113         discount rate, mentioned in Section 2.2
114     eta : <>, optional
115         constant from Table 1
116     gamma : float, optional
117         paper says this is taken from government report, but no indication
118         of actual value
119     rho : float, optional
120         risk-free rate, mentioned in Section 2.2
121     zeta : <>, optional
122         constant from Table 1
123     c : <>, optional
124         constant from Table 1
125     b : <>, optional
126         constant from Table 1
127     lam : <>, optional
128         constant from Table 1
129
130     """

```

(continues on next page)

(continued from previous page)

```

131 Ts = [T1, T2, T3, T4, T5, T6]
132 Xs = [X1, X2, X3, X4, X5, X6]
133
134 Ts = [int(Ts[i] + sum(Ts[:i])) for i in range(len(Ts)) if Ts[i] + sum(Ts[:i]) < T]
135 Xs = Xs[: len(Ts)]
136
137 if len(Ts) == 0:
138     Ts = [0]
139     Xs = [0]
140
141 if Ts[0] > 0:
142     Ts.insert(0, 0)
143     Xs.insert(0, 0)
144
145 S0 = P0 * V0
146 beta = alpha * eta + gamma - rho
147 theta = alpha - zeta
148
149 # calculate investment
150 investment = 0
151
152 for i in range(len(Xs)):
153     step_cost = exponential_investment_cost(Xs[i], 0 if i == 0 else sum(Xs[:i]), c,
154 ↪ b, lam)
155     step_discount = math.exp(-delta * Ts[i])
156     investment += step_cost * step_discount
157
158 # calculate expected losses
159 losses = 0
160
161 for i in range(len(Xs) - 1):
162     losses += math.exp(-theta * sum(Xs[: (i + 1)])) * (
163         math.exp((beta - delta) * Ts[i + 1]) - math.exp((beta - delta) * Ts[i])
164     )
165
166 if Ts[-1] < T:
167     losses += math.exp(-theta * sum(Xs)) * (
168         math.exp((beta - delta) * T) - math.exp((beta - delta) * Ts[-1])
169     )
170
171 losses = losses * S0 / (beta - delta)
172
173 # salvage term
174 losses += S0 * math.exp(beta * T) * math.exp(-theta * sum(Xs)) * math.exp(-delta *
175 ↪ T) / delta
176
177 def find_height(t):
178     if t < Ts[0]:
179         return 0
180     elif t > Ts[-1]:
181         return sum(Xs)
182     else:

```

(continues on next page)

(continued from previous page)

```

181         return sum(Xs[: bisect.bisect_right(Ts, t)])
182
183     failure_probability = [
184         P0 * np.exp(alpha * eta * t) * np.exp(-alpha * find_height(t)) for t in range(T_
185 ↪ + 1)
186     ]
187     total_failure = 1 - functools.reduce(operator.mul, [1 - p for p in failure_
188 ↪ probability], 1)
189     mean_failure = sum(failure_probability) / (T + 1)
190     max_failure = max(failure_probability)
191
192     return (investment, losses, investment + losses, total_failure, mean_failure, max_
193 ↪ failure)
194
195 if __name__ == "__main__":
196     model = Model("eigenraam", eigenraam_model)
197
198     model.responses = [
199         ScalarOutcome("TotalInvestment", ScalarOutcome.INFO),
200         ScalarOutcome("TotalLoss", ScalarOutcome.INFO),
201         ScalarOutcome("TotalCost", ScalarOutcome.MINIMIZE),
202         ScalarOutcome("TotalFailureProb", ScalarOutcome.INFO),
203         ScalarOutcome("AvgFailureProb", ScalarOutcome.MINIMIZE),
204         ScalarOutcome("MaxFailureProb", ScalarOutcome.MINIMIZE),
205     ]
206
207     # Set uncertainties
208     model.uncertainties = [
209         RealParameter.from_dist("P0", sp.stats.lognorm(scale=0.00137, s=0.25)),
210         # @UndefinedVariable
211         RealParameter.from_dist("alpha", sp.stats.norm(loc=0.0502, scale=0.01)),
212         # @UndefinedVariable
213         RealParameter.from_dist("eta", sp.stats.lognorm(scale=0.76, s=0.1)),
214     ] # @UndefinedVariable
215
216     # having a list like parameter were values are automagically wrappen
217     # into a list can be quite useful....
218     model.levers = [RealParameter(f"X{i}", 0, 500) for i in range(1, 7)] + [
219         RealParameter(f"T{i}", 0, 300) for i in range(1, 7)
220     ]
221
222     ema_logging.log_to_stderr(ema_logging.INFO)
223
224     with MultiprocessingEvaluator(model, n_processes=4) as evaluator:
225         results = evaluator.perform_experiments(1000, 4)

```

1.9.5 example_excel.py

```
1  """
2  Created on 27 Jul. 2011
3
4  This file illustrates the use the EMA classes for a model in Excel.
5
6  It used the excel file provided by
7  `A. Sharov <https://home.comcast.net/~sharov/PopEcol/lec10/fullmod.html>`
8
9  This excel file implements a simple predator prey model.
10
11  .. codeauthor:: jhkwakkel <j.h.kwakkel (at) tudelft (dot) nl>
12  """
13  from ema_workbench import RealParameter, TimeSeriesOutcome, ema_logging, perform_
14  ↪experiments
15
16  from ema_workbench.connectors.excel import ExcelModel
17  from ema_workbench.em_framework.evaluators import MultiprocessingEvaluator
18
19  if __name__ == "__main__":
20      ema_logging.log_to_stderr(level=ema_logging.INFO)
21
22      model = ExcelModel("predatorPrey", wd="./models/excelModel", model_file="excel_
23  ↪example.xlsx")
24      model.uncertainties = [
25          RealParameter("K2", 0.01, 0.2),
26          # we can refer to a cell in the normal way
27          # we can also use named cells
28          RealParameter("KKK", 450, 550),
29          RealParameter("rP", 0.05, 0.15),
30          RealParameter("aaa", 0.00001, 0.25),
31          RealParameter("tH", 0.45, 0.55),
32          RealParameter("kk", 0.1, 0.3),
33      ]
34
35      # specification of the outcomes
36      model.outcomes = [
37          TimeSeriesOutcome("B4:B1076"),
38          # we can refer to a range in the normal way
39          TimeSeriesOutcome("P_t"),
40      ] # we can also use named range
41
42      # name of the sheet
43      model.default_sheet = "Sheet1"
44
45      with MultiprocessingEvaluator(model) as evaluator:
46          results = perform_experiments(model, 100, reporting_interval=1,
47  ↪evaluator=evaluator)
```

1.9.6 example_flu.py

```

1  """
2  Created on 20 dec. 2010
3
4  This file illustrated the use of the workbench for a model
5  specified in Python itself. The example is based on `Pruyt & Hamarat <https://www.
6  ↪systemdynamics.org/conferences/2010/proceed/papers/P1253.pdf>`.
7  For comparison, run both this model and the flu_vensim_no_policy_example.py and
8  compare the results.
9
10 .. codeauthor:: jhkwakkel <j.h.kwakkel (at) tudelft (dot) nl>
11                  chamarat <c.hamarat (at) tudelft (dot) nl>
12
13 """
14 import matplotlib.pyplot as plt
15 import numpy as np
16 from numpy import sin, min, exp
17
18 from ema_workbench import Model, RealParameter, TimeSeriesOutcome, perform_experiments, ↵
19 ↪ema_logging
20 from ema_workbench import MultiprocessingEvaluator, SequentialEvaluator
21 from ema_workbench.analysis import lines, Density
22
23 # =====
24 #
25 #   the model itself
26 #
27 # =====
28
29 FINAL_TIME = 48
30 INITIAL_TIME = 0
31 TIME_STEP = 0.0078125
32
33 switch_regions = 1.0
34 switch_immunity = 1.0
35 switch_deaths = 1.0
36 switch_immunity_cap = 1.0
37
38 def LookupFunctionX(variable, start, end, step, skew, growth, v=0.5):
39     return start + ((end - start) / ((1 + skew * exp(-growth * (variable - step))) ** (1 ↵
40     ↪/ v)))
41
42 def flu_model(
43     x11=0,
44     x12=0,
45     x21=0,
46     x22=0,
47     x31=0,
48     x32=0,

```

(continues on next page)

(continued from previous page)

```

49     x41=0,
50     x51=0,
51     x52=0,
52     x61=0,
53     x62=0,
54     x81=0,
55     x82=0,
56     x91=0,
57     x92=0,
58     x101=0,
59     x102=0,
60 ):
61     # Assigning initial values
62     additional_seasonal_immune_population_fraction_R1 = float(x11)
63     additional_seasonal_immune_population_fraction_R2 = float(x12)
64
65     fatality_rate_region_1 = float(x21)
66     fatality_rate_region_2 = float(x22)
67
68     initial_immune_fraction_of_the_population_of_region_1 = float(x31)
69     initial_immune_fraction_of_the_population_of_region_2 = float(x32)
70
71     normal_interregional_contact_rate = float(x41)
72     interregional_contact_rate = switch_regions * normal_interregional_contact_rate
73
74     permanent_immune_population_fraction_R1 = float(x51)
75     permanent_immune_population_fraction_R2 = float(x52)
76
77     recovery_time_region_1 = float(x61)
78     recovery_time_region_2 = float(x62)
79
80     susceptible_to_immune_population_delay_time_region_1 = 1
81     susceptible_to_immune_population_delay_time_region_2 = 1
82
83     root_contact_rate_region_1 = float(x81)
84     root_contact_rate_region_2 = float(x82)
85
86     infection_rate_region_1 = float(x91)
87     infection_rate_region_2 = float(x92)
88
89     normal_contact_rate_region_1 = float(x101)
90     normal_contact_rate_region_2 = float(x102)
91
92     #####
93     susceptible_to_immune_population_flow_region_1 = 0.0
94     susceptible_to_immune_population_flow_region_2 = 0.0
95     #####
96
97     initial_value_population_region_1 = 6.0 * 10**8
98     initial_value_population_region_2 = 3.0 * 10**9
99
100    initial_value_infected_population_region_1 = 10.0

```

(continues on next page)

(continued from previous page)

```

101 initial_value_infected_population_region_2 = 10.0
102
103 initial_value_immune_population_region_1 = (
104     switch_immunity
105     * initial_immune_fraction_of_the_population_of_region_1
106     * initial_value_population_region_1
107 )
108 initial_value_immune_population_region_2 = (
109     switch_immunity
110     * initial_immune_fraction_of_the_population_of_region_2
111     * initial_value_population_region_2
112 )
113
114 initial_value_susceptible_population_region_1 = (
115     initial_value_population_region_1 - initial_value_immune_population_region_1
116 )
117 initial_value_susceptible_population_region_2 = (
118     initial_value_population_region_2 - initial_value_immune_population_region_2
119 )
120
121 recovered_population_region_1 = 0.0
122 recovered_population_region_2 = 0.0
123
124 infected_population_region_1 = initial_value_infected_population_region_1
125 infected_population_region_2 = initial_value_infected_population_region_2
126
127 susceptible_population_region_1 = initial_value_susceptible_population_region_1
128 susceptible_population_region_2 = initial_value_susceptible_population_region_2
129
130 immune_population_region_1 = initial_value_immune_population_region_1
131 immune_population_region_2 = initial_value_immune_population_region_2
132
133 deceased_population_region_1 = [0.0]
134 deceased_population_region_2 = [0.0]
135 runTime = [INITIAL_TIME]
136
137 # --End of Initialization--
138
139 Max_infected = 0.0
140
141 for time in range(int(INITIAL_TIME / TIME_STEP), int(FINAL_TIME / TIME_STEP)):
142     runTime.append(runTime[-1] + TIME_STEP)
143     total_population_region_1 = (
144         infected_population_region_1
145         + recovered_population_region_1
146         + susceptible_population_region_1
147         + immune_population_region_1
148     )
149     total_population_region_2 = (
150         infected_population_region_2
151         + recovered_population_region_2
152         + susceptible_population_region_2

```

(continues on next page)

(continued from previous page)

```

153         + immune_population_region_2
154     )
155
156     infected_population_region_1 = max(0, infected_population_region_1)
157     infected_population_region_2 = max(0, infected_population_region_2)
158
159     infected_fraction_region_1 = infected_population_region_1 / total_population_
↪region_1
160     infected_fraction_region_2 = infected_population_region_2 / total_population_
↪region_2
161
162     impact_infected_population_on_contact_rate_region_1 = 1 - (
163         infected_fraction_region_1 ** (1 / root_contact_rate_region_1)
164     )
165     impact_infected_population_on_contact_rate_region_2 = 1 - (
166         infected_fraction_region_2 ** (1 / root_contact_rate_region_2)
167     )
168
169     #         if ((time*TIME_STEP) >= 4) and ((time*TIME_STEP)<=10):
170     #             normal_contact_rate_region_1 = float(x101)*(1 - 0.5)
171     #         else:normal_contact_rate_region_1 = float(x101)
172
173     normal_contact_rate_region_1 = float(x101) * (
174         1 - LookupFunctionX(infected_fraction_region_1, 0, 1, 0.15, 0.75, 15)
175     )
176
177     contact_rate_region_1 = (
178         normal_contact_rate_region_1 * impact_infected_population_on_contact_rate_
↪region_1
179     )
180     contact_rate_region_2 = (
181         normal_contact_rate_region_2 * impact_infected_population_on_contact_rate_
↪region_2
182     )
183
184     recoveries_region_1 = (
185         (1 - (fatality_rate_region_1 * switch_deaths))
186         * infected_population_region_1
187         / recovery_time_region_1
188     )
189     recoveries_region_2 = (
190         (1 - (fatality_rate_region_2 * switch_deaths))
191         * infected_population_region_2
192         / recovery_time_region_2
193     )
194
195     flu_deaths_region_1 = (
196         fatality_rate_region_1
197         * switch_deaths
198         * infected_population_region_1
199         / recovery_time_region_1
200     )

```

(continues on next page)

(continued from previous page)

```

201     flu_deaths_region_2 = (
202         fatality_rate_region_2
203         * switch_deaths
204         * infected_population_region_2
205         / recovery_time_region_2
206     )
207
208     infections_region_1 = (
209         susceptible_population_region_1
210         * contact_rate_region_1
211         * infection_rate_region_1
212         * infected_fraction_region_1
213     ) + (
214         susceptible_population_region_1
215         * interregional_contact_rate
216         * infection_rate_region_1
217         * infected_fraction_region_2
218     )
219     infections_region_2 = (
220         susceptible_population_region_2
221         * contact_rate_region_2
222         * infection_rate_region_2
223         * infected_fraction_region_2
224     ) + (
225         susceptible_population_region_2
226         * interregional_contact_rate
227         * infection_rate_region_2
228         * infected_fraction_region_1
229     )
230
231     infected_population_region_1_NEXT = infected_population_region_1 + (
232         TIME_STEP * (infections_region_1 - flu_deaths_region_1 - recoveries_region_1)
233     )
234     infected_population_region_2_NEXT = infected_population_region_2 + (
235         TIME_STEP * (infections_region_2 - flu_deaths_region_2 - recoveries_region_2)
236     )
237
238     if infected_population_region_1_NEXT < 0 or infected_population_region_2_NEXT < 0:
239         pass
240
241     recovered_population_region_1_NEXT = recovered_population_region_1 + (
242         TIME_STEP * recoveries_region_1
243     )
244     recovered_population_region_2_NEXT = recovered_population_region_2 + (
245         TIME_STEP * recoveries_region_2
246     )
247
248     if fatality_rate_region_1 >= 0.025:
249         qw = 1.0
250     elif fatality_rate_region_1 >= 0.01:
251         qw = 0.8

```

(continues on next page)

(continued from previous page)

```

252 elif fatality_rate_region_1 >= 0.001:
253     qw = 0.6
254 elif fatality_rate_region_1 >= 0.0001:
255     qw = 0.4
256 else:
257     qw = 0.2
258
259 if (time * TIME_STEP) <= 10:
260     normal_immune_population_fraction_region_1 = (
261         additional_seasonal_immune_population_fraction_R1 / 2
262     ) * sin(4.5 + (time * TIME_STEP / 2)) + (
263         (
264             (2 * permanent_immune_population_fraction_R1)
265             + additional_seasonal_immune_population_fraction_R1
266         )
267         / 2
268     )
269 else:
270     normal_immune_population_fraction_region_1 = max(
271         (
272             float(qw),
273             (additional_seasonal_immune_population_fraction_R1 / 2)
274             * sin(4.5 + (time * TIME_STEP / 2))
275             + (
276                 (
277                     (2 * permanent_immune_population_fraction_R1)
278                     + additional_seasonal_immune_population_fraction_R1
279                 )
280                 / 2
281             ),
282         )
283     )
284
285 normal_immune_population_fraction_region_2 = switch_immunity_cap * min(
286     (
287         (
288             sin((time * TIME_STEP / 2) + 1.5)
289             * additional_seasonal_immune_population_fraction_R2
290             / 2
291         )
292         + (
293             (
294                 (2 * permanent_immune_population_fraction_R2)
295                 + additional_seasonal_immune_population_fraction_R2
296             )
297             / 2
298         ),
299         (
300             permanent_immune_population_fraction_R1
301             + additional_seasonal_immune_population_fraction_R1
302         ),
303     ),

```

(continues on next page)

(continued from previous page)

```

304     ) + (
305         (1 - switch_immunity_cap)
306         * (
307             (
308                 sin((time * TIME_STEP / 2) + 1.5)
309                 * additional_seasonal_immune_population_fraction_R2
310                 / 2
311             )
312             + (
313                 (
314                     (2 * permanent_immune_population_fraction_R2)
315                     + additional_seasonal_immune_population_fraction_R2
316                 )
317                 / 2
318             )
319         )
320     )
321
322     normal_immune_population_region_1 = (
323         normal_immune_population_fraction_region_1 * total_population_region_1
324     )
325     normal_immune_population_region_2 = (
326         normal_immune_population_fraction_region_2 * total_population_region_2
327     )
328
329     if switch_immunity == 1:
330         susminreg1_1 = (
331             normal_immune_population_region_1 - immune_population_region_1
332         ) / susceptible_to_immune_population_delay_time_region_1
333         susminreg1_2 = (
334             susceptible_population_region_1
335             / susceptible_to_immune_population_delay_time_region_1
336         )
337         susmaxreg1 = -(
338             immune_population_region_1 / susceptible_to_immune_population_delay_time_
339             ↪ region_1
340         )
341         if (susmaxreg1 >= susminreg1_1) or (susmaxreg1 >= susminreg1_2):
342             susceptible_to_immune_population_flow_region_1 = susmaxreg1
343         elif (susminreg1_1 < susminreg1_2) and (susminreg1_1 > susmaxreg1):
344             susceptible_to_immune_population_flow_region_1 = susminreg1_1
345         elif (susminreg1_2 < susminreg1_1) and (susminreg1_2 > susmaxreg1):
346             susceptible_to_immune_population_flow_region_1 = susminreg1_2
347         else:
348             susceptible_to_immune_population_flow_region_1 = 0
349
350     if switch_immunity == 1:
351         susminreg2_1 = (
352             normal_immune_population_region_2 - immune_population_region_2
353         ) / susceptible_to_immune_population_delay_time_region_2
354         susminreg2_2 = (
355             susceptible_population_region_2

```

(continues on next page)

(continued from previous page)

```

355         / susceptible_to_immune_population_delay_time_region_2
356     )
357     susmaxreg2 = -(
358         immune_population_region_2 / susceptible_to_immune_population_delay_time_
↪region_2
359     )
360     if (susmaxreg2 >= susminreg2_1) or (susmaxreg2 >= susminreg2_2):
361         susceptible_to_immune_population_flow_region_2 = susmaxreg2
362     elif (susminreg2_1 < susminreg2_2) and (susminreg2_1 > susmaxreg2):
363         susceptible_to_immune_population_flow_region_2 = susminreg2_1
364     elif (susminreg2_2 < susminreg2_1) and (susminreg2_2 > susmaxreg2):
365         susceptible_to_immune_population_flow_region_2 = susminreg2_2
366     else:
367         susceptible_to_immune_population_flow_region_2 = 0
368
369     susceptible_population_region_1_NEXT = susceptible_population_region_1 - (
370         TIME_STEP * (infections_region_1 + susceptible_to_immune_population_flow_
↪region_1)
371     )
372     susceptible_population_region_2_NEXT = susceptible_population_region_2 - (
373         TIME_STEP * (infections_region_2 + susceptible_to_immune_population_flow_
↪region_2)
374     )
375
376     immune_population_region_1_NEXT = immune_population_region_1 + (
377         TIME_STEP * susceptible_to_immune_population_flow_region_1
378     )
379     immune_population_region_2_NEXT = immune_population_region_2 + (
380         TIME_STEP * susceptible_to_immune_population_flow_region_2
381     )
382
383     deceased_population_region_1_NEXT = deceased_population_region_1[-1] + (
384         TIME_STEP * flu_deaths_region_1
385     )
386     deceased_population_region_2_NEXT = deceased_population_region_2[-1] + (
387         TIME_STEP * flu_deaths_region_2
388     )
389
390     # Updating integral values
391     if Max_infected < (
392         infected_population_region_1_NEXT
393         / (
394             infected_population_region_1_NEXT
395             + recovered_population_region_1_NEXT
396             + susceptible_population_region_1_NEXT
397             + immune_population_region_1_NEXT
398         )
399     ):
400         Max_infected = infected_population_region_1_NEXT / (
401             infected_population_region_1_NEXT
402             + recovered_population_region_1_NEXT
403             + susceptible_population_region_1_NEXT

```

(continues on next page)

(continued from previous page)

```

404         + immune_population_region_1_NEXT
405     )
406
407     recovered_population_region_1 = recovered_population_region_1_NEXT
408     recovered_population_region_2 = recovered_population_region_2_NEXT
409
410     infected_population_region_1 = infected_population_region_1_NEXT
411     infected_population_region_2 = infected_population_region_2_NEXT
412
413     susceptible_population_region_1 = susceptible_population_region_1_NEXT
414     susceptible_population_region_2 = susceptible_population_region_2_NEXT
415
416     immune_population_region_1 = immune_population_region_1_NEXT
417     immune_population_region_2 = immune_population_region_2_NEXT
418
419     deceased_population_region_1.append(deceased_population_region_1_NEXT)
420     deceased_population_region_2.append(deceased_population_region_2_NEXT)
421
422     # End of main code
423
424     return {"TIME": runTime, "deceased_population_region_1": deceased_population_region_
↪1}
425
426
427 if __name__ == "__main__":
428     ema_logging.log_to_stderr(ema_logging.INFO)
429
430     model = Model("mexicanFlu", function=flu_model)
431     model.uncertainties = [
432         RealParameter("x11", 0, 0.5),
433         RealParameter("x12", 0, 0.5),
434         RealParameter("x21", 0.0001, 0.1),
435         RealParameter("x22", 0.0001, 0.1),
436         RealParameter("x31", 0, 0.5),
437         RealParameter("x32", 0, 0.5),
438         RealParameter("x41", 0, 0.9),
439         RealParameter("x51", 0, 0.5),
440         RealParameter("x52", 0, 0.5),
441         RealParameter("x61", 0, 0.8),
442         RealParameter("x62", 0, 0.8),
443         RealParameter("x81", 1, 10),
444         RealParameter("x82", 1, 10),
445         RealParameter("x91", 0, 0.1),
446         RealParameter("x92", 0, 0.1),
447         RealParameter("x101", 0, 200),
448         RealParameter("x102", 0, 200),
449     ]
450
451     model.outcomes = [TimeSeriesOutcome("TIME"), TimeSeriesOutcome("deceased_population_
↪region_1")]
452
453     nr_experiments = 500

```

(continues on next page)

(continued from previous page)

```

454
455     with SequentialEvaluator(model) as evaluator:
456         results = perform_experiments(model, nr_experiments, evaluator=evaluator)
457
458     print("laat")
459
460     lines(
461         results,
462         outcomes_to_show="deceased_population_region_1",
463         show_envelope=True,
464         density=Density.KDE,
465         titles=None,
466         experiments_to_show=np.arange(0, nr_experiments, 10),
467     )
468     plt.show()

```

1.9.7 example_lake_model.py

```

1  """
2  An example of the lake problem using the ema workbench.
3
4  The model itself is adapted from the Rhodium example by Dave Hadka,
5  see https://gist.github.com/dhadka/a8d7095c98130d8f73bc
6
7  """
8  import math
9
10 import numpy as np
11 from scipy.optimize import brentq
12
13 from ema_workbench import (
14     Model,
15     RealParameter,
16     ScalarOutcome,
17     Constant,
18     ema_logging,
19     MultiprocessingEvaluator,
20 )
21 from ema_workbench.em_framework.evaluators import Samplers
22
23
24 def lake_problem(
25     b=0.42, # decay rate for P in lake (0.42 = irreversible)
26     q=2.0, # recycling exponent
27     mean=0.02, # mean of natural inflows
28     stdev=0.001, # future utility discount rate
29     delta=0.98, # standard deviation of natural inflows
30     alpha=0.4, # utility from pollution
31     nsamples=100, # Monte Carlo sampling of natural inflows
32     **kwargs,

```

(continues on next page)

(continued from previous page)

```

33 ):
34     try:
35         decisions = [kwargs[str(i)] for i in range(100)]
36     except KeyError:
37         decisions = [0] * 100
38
39     Pcrit = brentq(lambda x: x**q / (1 + x**q) - b * x, 0.01, 1.5)
40     nvars = len(decisions)
41     X = np.zeros((nvars,))
42     average_daily_P = np.zeros((nvars,))
43     decisions = np.array(decisions)
44     reliability = 0.0
45
46     for _ in range(nsamples):
47         X[0] = 0.0
48
49         natural_inflows = np.random.lognormal(
50             math.log(mean**2 / math.sqrt(stdev**2 + mean**2)),
51             math.sqrt(math.log(1.0 + stdev**2 / mean**2)),
52             size=nvars,
53         )
54
55         for t in range(1, nvars):
56             X[t] = (
57                 (1 - b) * X[t - 1]
58                 + X[t - 1] ** q / (1 + X[t - 1] ** q)
59                 + decisions[t - 1]
60                 + natural_inflows[t - 1]
61             )
62             average_daily_P[t] += X[t] / float(nsamples)
63
64         reliability += np.sum(X < Pcrit) / float(nsamples * nvars)
65
66     max_P = np.max(average_daily_P)
67     utility = np.sum(alpha * decisions * np.power(delta, np.arange(nvars)))
68     inertia = np.sum(np.absolute(np.diff(decisions)) < 0.02) / float(nvars - 1)
69
70     return max_P, utility, inertia, reliability
71
72
73 if __name__ == "__main__":
74     ema_logging.log_to_stderr(ema_logging.INFO)
75
76     # instantiate the model
77     lake_model = Model("lakeproblem", function=lake_problem)
78     lake_model.time_horizon = 100
79
80     # specify uncertainties
81     lake_model.uncertainties = [
82         RealParameter("b", 0.1, 0.45),
83         RealParameter("q", 2.0, 4.5),
84         RealParameter("mean", 0.01, 0.05),

```

(continues on next page)

(continued from previous page)

```

85     RealParameter("stdev", 0.001, 0.005),
86     RealParameter("delta", 0.93, 0.99),
87 ]
88
89 # set levers, one for each time step
90 lake_model.levers = [RealParameter(str(i), 0, 0.1) for i in range(lake_model.time_
↪ horizon)]
91
92 # specify outcomes
93 lake_model.outcomes = [
94     ScalarOutcome("max_P"),
95     ScalarOutcome("utility"),
96     ScalarOutcome("inertia"),
97     ScalarOutcome("reliability"),
98 ]
99
100 # override some of the defaults of the model
101 lake_model.constants = [Constant("alpha", 0.41), Constant("nsamples", 150)]
102
103 # generate some random policies by sampling over levers
104 n_scenarios = 1000
105 n_policies = 4
106
107 with MultiprocessingEvaluator(lake_model) as evaluator:
108     res = evaluator.perform_experiments(n_scenarios, n_policies, lever_
↪ sampling=Samplers.MC)

```

1.9.8 example_netlogo.py

```

1  """
2
3  This example is a proof of principle for how NetLogo models can be
4  controlled using pyNetLogo and the ema_workbench. Note that this
5  example uses the NetLogo 6 version of the predator prey model that
6  comes with NetLogo. If you are using NetLogo 5, replace the model file
7  with the one that comes with NetLogo.
8
9  """
10 import numpy as np
11
12 from ema_workbench import (
13     RealParameter,
14     ema_logging,
15     ScalarOutcome,
16     TimeSeriesOutcome,
17     MultiprocessingEvaluator,
18 )
19 from ema_workbench.connectors.netlogo import NetLogoModel
20
21 # Created on 20 mrt. 2013

```

(continues on next page)

(continued from previous page)

```

22 #
23 # .. codeauthor:: jhkwakkel
24
25
26 if __name__ == "__main__":
27     # turn on logging
28     ema_logging.log_to_stderr(ema_logging.INFO)
29
30     model = NetLogoModel(
31         "predprey", wd="./models/predatorPreyNetlogo", model_file="Wolf Sheep Predation.
↪nlogo"
32     )
33     model.run_length = 100
34     model.replications = 10
35
36     model.uncertainties = [
37         RealParameter("grass-regrowth-time", 1, 99),
38         RealParameter("initial-number-sheep", 50, 100),
39         RealParameter("initial-number-wolves", 50, 100),
40         RealParameter("sheep-reproduce", 5, 10),
41         RealParameter("wolf-reproduce", 5, 10),
42     ]
43
44     model.outcomes = [
45         ScalarOutcome("sheep", variable_name="count sheep", function=np.mean),
46         TimeSeriesOutcome("wolves"),
47         TimeSeriesOutcome("grass"),
48     ]
49
50     # perform experiments
51     n = 10
52
53     with MultiprocessingEvaluator(model, n_processes=2, maxtasksperchild=4) as evaluator:
54         results = evaluator.perform_experiments(n)
55
56     print()

```

1.9.9 example_pysd_tecup.py

```

1  """
2
3
4  """
5
6  # Created on Jul 23, 2016
7  #
8  # .. codeauthor:: jhkwakkel <j.h.kwakkel (at) tudelft (dot) nl>
9
10 from ema_workbench import RealParameter, TimeSeriesOutcome, ema_logging, perform_
↪experiments

```

(continues on next page)

(continued from previous page)

```

11
12 from ema_workbench.connectors.pysd_connector import PysdModel
13
14 if __name__ == "__main__":
15     ema_logging.log_to_stderr(ema_logging.INFO)
16
17     mdl_file = "./models/pysd/Teacup.mdl"
18
19     model = PysdModel(mdl_file=mdl_file)
20
21     model.uncertainties = [RealParameter("Room Temperature", 33, 120)]
22     model.outcomes = [TimeSeriesOutcome("Teacup Temperature")]
23
24     perform_experiments(model, 100)

```

1.9.10 example_python.py

```

1  """
2  Created on 20 dec. 2010
3
4  This file illustrated the use the EMA classes for a contrived example
5  It's main purpose has been to test the parallel processing functionality
6
7  .. codeauthor:: jhkwakkel <j.h.kwakkel (at) tudelft (dot) nl>
8  """
9  from ema_workbench import Model, RealParameter, ScalarOutcome, ema_logging, perform_
10 ↪ experiments
11
12 def some_model(x1=None, x2=None, x3=None):
13     return {"y": x1 * x2 + x3}
14
15
16 if __name__ == "__main__":
17     ema_logging.LOG_FORMAT = "[% (name)s/% (levelname)s/% (processName)s] %(message)s"
18     ema_logging.log_to_stderr(ema_logging.INFO)
19
20     model = Model("simpleModel", function=some_model) # instantiate the model
21
22     # specify uncertainties
23     model.uncertainties = [
24         RealParameter("x1", 0.1, 10),
25         RealParameter("x2", -0.01, 0.01),
26         RealParameter("x3", -0.01, 0.01),
27     ]
28     # specify outcomes
29     model.outcomes = [ScalarOutcome("y")]
30
31     results = perform_experiments(model, 100)

```

1.9.11 example_simio.py

```

1  """
2  Created on 27 Jun 2019
3
4  @author: jhkwakkel
5  """
6  from ema_workbench import ema_logging, CategoricalParameter, MultiprocessingEvaluator,
   ↪ ScalarOutcome
7
8  from ema_workbench.connectors.simio_connector import SimioModel
9
10 if __name__ == "__main__":
11     ema_logging.log_to_stderr(ema_logging.INFO)
12
13     model = SimioModel(
14         "simioDemo", wd="./model_bahareh", model_file="SupplyChainV3.spfx", main_model=
   ↪ "Model"
15     )
16
17     model.uncertainties = [
18         CategoricalParameter("DemandDistributionParameter", (20, 30, 40, 50, 60)),
19         CategoricalParameter("DemandInterarrivalTime", (0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.
   ↪ 75, 2)),
20     ]
21
22     model.levers = [
23         CategoricalParameter("InitialInventory", (500, 600, 700, 800, 900)),
24         CategoricalParameter("ReorderPoint", (100, 200, 300, 400, 500)),
25         CategoricalParameter("OrderUpToQuantity", (500, 600, 700, 800, 900)),
26         CategoricalParameter("ReviewPeriod", (3, 4, 5, 6, 7)),
27     ]
28
29     model.outcomes = [ScalarOutcome("AverageInventory"), ScalarOutcome(
   ↪ "AverageServiceLevel")]
30
31     n_scenarios = 10
32     n_policies = 2
33
34     with MultiprocessingEvaluator(model) as evaluator:
35         results = evaluator.perform_experiments(n_scenarios, n_policies)

```

1.9.12 example_vensim.py

```

1  """
2  Created on 3 Jan. 2011
3
4  This file illustrated the use the EMA classes for a contrived vensim
5  example
6
7

```

(continues on next page)

(continued from previous page)

```

8  .. codeauthor:: jhkwakkel <j.h.kwakkel (at) tudelft (dot) nl>
9                  chamarat <c.hamarat (at) tudelft (dot) nl>
10  """
11  from ema_workbench import TimeSeriesOutcome, perform_experiments, RealParameter, ema_
12  ↪ logging
13
14  from ema_workbench.connectors.vensim import VensimModel
15
16  if __name__ == "__main__":
17      # turn on logging
18      ema_logging.log_to_stderr(ema_logging.INFO)
19
20      # instantiate a model
21      wd = "./models/vensim example"
22      vensimModel = VensimModel("simpleModel", wd=wd, model_file="model.vpm")
23      vensimModel.uncertainties = [RealParameter("x11", 0, 2.5), RealParameter("x12", -2.5,
24  ↪ 2.5)]
25
26      vensimModel.outcomes = [TimeSeriesOutcome("a")]
27
28      results = perform_experiments(vensimModel, 1000)

```

1.9.13 example_vensim_advanced_flu.py

```

1  """
2  Created on 20 May, 2011
3
4  This module shows how you can use vensim models directly
5  instead of coding the model in Python. The underlying case
6  is the same as used in fluExample
7
8  .. codeauthor:: jhkwakkel <j.h.kwakkel (at) tudelft (dot) nl>
9                  epruyt <e.pruyt (at) tudelft (dot) nl>
10  """
11  import numpy as np
12
13  from ema_workbench import (
14      TimeSeriesOutcome,
15      ScalarOutcome,
16      ema_logging,
17      Policy,
18      MultiprocessingEvaluator,
19      save_results,
20  )
21  from ema_workbench.connectors.vensim import VensimModel
22  from ema_workbench.em_framework.parameters import parameters_from_csv
23
24
25  def time_of_max(infected_fraction, time):
26      index = np.where(infected_fraction == np.max(infected_fraction))

```

(continues on next page)

(continued from previous page)

```

27     timing = time[index][0]
28     return timing
29
30
31 if __name__ == "__main__":
32     ema_logging.log_to_stderr(ema_logging.INFO)
33
34     model = VensimModel("fluCase", wd="./models/flu", model_file="FLUvensimV1basecase.vpm
↪")
35
36     # outcomes
37     model.outcomes = [
38         TimeSeriesOutcome("deceased population region 1"),
39         TimeSeriesOutcome("infected fraction R1"),
40         ScalarOutcome(
41             "max infection fraction", variable_name="infected fraction R1", function=np.
↪max
42         ),
43         ScalarOutcome(
44             "time of max", variable_name=["infected fraction R1", "TIME"], function=time_
↪of_max
45         ),
46     ]
47
48     # create uncertainties based on csv
49     model.uncertainties = parameters_from_csv("./models/flu/flu_uncertainties.csv")
50
51     # add policies
52     policies = [
53         Policy("no policy", model_file="FLUvensimV1basecase.vpm"),
54         Policy("static policy", model_file="FLUvensimV1static.vpm"),
55         Policy("adaptive policy", model_file="FLUvensimV1dynamic.vpm"),
56     ]
57
58     with MultiprocessingEvaluator(model, n_processes=4) as evaluator:
59         results = evaluator.perform_experiments(1000, policies=policies)
60
61     save_results(results, "./data/1000 flu cases with policies.tar.gz")

```

1.9.14 example_vensim_flu.py

```

1  """
2  Created on 20 May, 2011
3
4  This module shows how you can use vensim models directly
5  instead of coding the model in Python. The underlying case
6  is the same as used in fluExample
7
8  .. codeauthor:: jhkwakkel <j.h.kwakkel (at) tudelft (dot) nl>
9                  epruyt <e.pruyt (at) tudelft (dot) nl>

```

(continues on next page)

(continued from previous page)

```

10  """
11  import numpy as np
12
13  from ema_workbench import (
14      RealParameter,
15      TimeSeriesOutcome,
16      ema_logging,
17      ScalarOutcome,
18      perform_experiments,
19  )
20  from ema_workbench.connectors.vensim import VensimModel
21  from ema_workbench.em_framework.parameters import Policy
22
23  if __name__ == "__main__":
24      ema_logging.log_to_stderr(ema_logging.INFO)
25
26      model = VensimModel("fluCase", wd=r"./models/flu", model_file=r"FLUvensimV1basecase.
↳vpm")
27
28      # outcomes
29      model.outcomes = [
30          TimeSeriesOutcome("deceased population region 1"),
31          TimeSeriesOutcome("infected fraction R1"),
32          ScalarOutcome(
33              "max infection fraction", variable_name="infected fraction R1", function=np.
↳max
34          ),
35      ]
36
37      # Plain Parametric Uncertainties
38      model.uncertainties = [
39          RealParameter("additional seasonal immune population fraction R1", 0, 0.5),
40          RealParameter("additional seasonal immune population fraction R2", 0, 0.5),
41          RealParameter("fatality ratio region 1", 0.0001, 0.1),
42          RealParameter("fatality rate region 2", 0.0001, 0.1),
43          RealParameter("initial immune fraction of the population of region 1", 0, 0.5),
44          RealParameter("initial immune fraction of the population of region 2", 0, 0.5),
45          RealParameter("normal interregional contact rate", 0, 0.9),
46          RealParameter("permanent immune population fraction R1", 0, 0.5),
47          RealParameter("permanent immune population fraction R2", 0, 0.5),
48          RealParameter("recovery time region 1", 0.1, 0.75),
49          RealParameter("recovery time region 2", 0.1, 0.75),
50          RealParameter("susceptible to immune population delay time region 1", 0.5, 2),
51          RealParameter("susceptible to immune population delay time region 2", 0.5, 2),
52          RealParameter("root contact rate region 1", 0.01, 5),
53          RealParameter("root contact ratio region 2", 0.01, 5),
54          RealParameter("infection ratio region 1", 0, 0.15),
55          RealParameter("infection rate region 2", 0, 0.15),
56          RealParameter("normal contact rate region 1", 10, 100),
57          RealParameter("normal contact rate region 2", 10, 200),
58      ]
59

```

(continues on next page)

(continued from previous page)

```

60 # add policies
61 policies = [
62     Policy("no policy", model_file=r"FLUvensimV1basecase.vpm"),
63     Policy("static policy", model_file=r"FLUvensimV1static.vpm"),
64     Policy("adaptive policy", model_file=r"FLUvensimV1dynamic.vpm"),
65 ]
66
67 results = perform_experiments(model, 1000, policies=policies)

```

1.9.15 example_vensim_lookup.py

```

1  """
2  Created on Oct 1, 2012
3
4  This is a simple example of the lookup uncertainty provided for
5  use in conjunction with vensim models. This example is largely based on
6  `Eker et al. (2014) <https://onlinelibrary.wiley.com/doi/10.1002/sdr.1518/supinfo>`_
7
8  @author: sibleker
9  @author: jhkwakkel
10 """
11 import matplotlib.pyplot as plt
12
13 from ema_workbench import TimeSeriesOutcome, perform_experiments, ema_logging
14 from ema_workbench.analysis import lines, Density
15 from ema_workbench.connectors.vensim import LookupUncertainty, VensimModel
16
17
18 class Burnout(VensimModel):
19     model_file = r"BURNOUT.vpm"
20     outcomes = [
21         TimeSeriesOutcome("Accomplishments to Date"),
22         TimeSeriesOutcome("Energy Level"),
23         TimeSeriesOutcome("Hours Worked Per Week"),
24         TimeSeriesOutcome("accomplishments per hour"),
25     ]
26
27     def __init__(self, working_directory, name):
28         super().__init__(working_directory, name)
29
30         self.uncertainties = [
31             LookupUncertainty(
32                 "hearne2",
33                 [(-1, 3), (-2, 1), (0, 0.9), (0.1, 1), (0.99, 1.01), (0.99, 1.01)],
34                 "accomplishments per hour lookup",
35                 self,
36                 0,
37                 1,
38             ),
39             LookupUncertainty(

```

(continues on next page)

(continued from previous page)

```

40         "hearne2",
41         [(-0.75, 0.75), (-0.75, 0.75), (0, 1.5), (0.1, 1.6), (-0.3, 1.5), (0.25, 2.5)],
42         "fractional change in expectations from perceived adequacy lookup",
43         self,
44         -1,
45         1,
46     ),
47     LookupUncertainty(
48         "hearne2",
49         [(-2, 2), (-1, 2), (0, 1.5), (0.1, 1.6), (0.5, 2), (0.5, 2)],
50         "effect of perceived adequacy on energy drain lookup",
51         self,
52         0,
53         10,
54     ),
55     LookupUncertainty(
56         "hearne2",
57         [(-2, 2), (-1, 2), (0, 1.5), (0.1, 1.6), (0.5, 1.5), (0.1, 2)],
58         "effect of perceived adequacy of hours worked lookup",
59         self,
60         0,
61         2.5,
62     ),
63     LookupUncertainty(
64         "hearne2",
65         [(-1, 1), (-1, 1), (0, 0.9), (0.1, 1), (0.5, 1.5), (1, 1.5)],
66         "effect of energy levels on hours worked lookup",
67         self,
68         0,
69         1.5,
70     ),
71     LookupUncertainty(
72         "hearne2",
73         [(-1, 1), (-1, 1), (0, 0.9), (0.1, 1), (0.5, 1.5), (1, 1.5)],
74         "effect of high energy on further recovery lookup",
75         self,
76         0,
77         1.25,
78     ),
79     LookupUncertainty(
80         "hearne2",
81         [(-2, 2), (-1, 1), (0, 100), (20, 120), (0.5, 1.5), (0.5, 2)],
82         "effect of hours worked on energy recovery lookup",
83         self,
84         0,
85         1.5,
86     ),
87     LookupUncertainty(
88         "approximation",
89         [(-0.5, 0.35), (3, 5), (1, 10), (0.2, 0.4), (0, 120)],
90         "effect of hours worked on energy drain lookup",

```

(continues on next page)

(continued from previous page)

```

91         self,
92         0,
93         3,
94     ),
95     LookupUncertainty(
96         "hearnel",
97         [(0, 1), (0, 0.15), (1, 1.5), (0.75, 1.25)],
98         "effect of low energy on further depletion lookup",
99         self,
100        0,
101        1,
102    ),
103 ]
104
105 self._delete_lookup_uncertainties()
106
107
108 if __name__ == "__main__":
109     ema_logging.log_to_stderr(ema_logging.INFO)
110     model = Burnout(r"./models/burnout", "burnout")
111
112     # run policy with old cases
113     results = perform_experiments(model, 100)
114     lines(results, "Energy Level", density=Density.BOXPLOT)
115     plt.show()

```

1.9.16 example_vensim_no_policy_flu.py

```

1  """
2  Created on 20 May, 2011
3
4  This module shows how you can use vensim models directly
5  instead of coding the model in Python. The underlying case
6  is the same as used in fluExample
7
8  .. codeauthor:: jhkwakkel <j.h.kwakkel (at) tudelft (dot) nl>
9                 epruyt <e.pruyt (at) tudelft (dot) nl>
10
11  """
12  from ema_workbench import (
13      RealParameter,
14      TimeSeriesOutcome,
15      ema_logging,
16      perform_experiments,
17      MultiprocessingEvaluator,
18  )
19
20  from ema_workbench.connectors.vensim import VensimModel
21
22  if __name__ == "__main__":
23      ema_logging.log_to_stderr(ema_logging.INFO)

```

(continues on next page)

(continued from previous page)

```

23
24 model = VensimModel("fluCase", wd="./models/flu", model_file="FLUvensimV1basecase.vpm
↪")
25
26 # outcomes
27 model.outcomes = [
28     TimeSeriesOutcome("deceased population region 1"),
29     TimeSeriesOutcome("infected fraction R1"),
30 ]
31
32 # Plain Parametric Uncertainties
33 model.uncertainties = [
34     RealParameter("additional seasonal immune population fraction R1", 0, 0.5),
35     RealParameter("additional seasonal immune population fraction R2", 0, 0.5),
36     RealParameter("fatality ratio region 1", 0.0001, 0.1),
37     RealParameter("fatality rate region 2", 0.0001, 0.1),
38     RealParameter("initial immune fraction of the population of region 1", 0, 0.5),
39     RealParameter("initial immune fraction of the population of region 2", 0, 0.5),
40     RealParameter("normal interregional contact rate", 0, 0.9),
41     RealParameter("permanent immune population fraction R1", 0, 0.5),
42     RealParameter("permanent immune population fraction R2", 0, 0.5),
43     RealParameter("recovery time region 1", 0.1, 0.75),
44     RealParameter("recovery time region 2", 0.1, 0.75),
45     RealParameter("susceptible to immune population delay time region 1", 0.5, 2),
46     RealParameter("susceptible to immune population delay time region 2", 0.5, 2),
47     RealParameter("root contact rate region 1", 0.01, 5),
48     RealParameter("root contact ratio region 2", 0.01, 5),
49     RealParameter("infection ratio region 1", 0, 0.15),
50     RealParameter("infection rate region 2", 0, 0.15),
51     RealParameter("normal contact rate region 1", 10, 100),
52     RealParameter("normal contact rate region 2", 10, 200),
53 ]
54
55 nr_experiments = 10
56 with MultiprocessingEvaluator(model) as evaluator:
57     results = perform_experiments(model, nr_experiments, evaluator=evaluator)

```

1.9.17 example_vensim_scarcity.py

```

1  """
2  Created on 8 mrt. 2011
3
4  .. codeauthor:: jhkwakkel <j.h.kwakkel (at) tudelft (dot) nl>
5                  epruyt <e.pruyt (at) tudelft (dot) nl>
6  """
7  from math import exp
8
9  from ema_workbench.connectors.vensim import VensimModel
10 from ema_workbench.em_framework import (
11     RealParameter,

```

(continues on next page)

(continued from previous page)

```

12     CategoricalParameter,
13     TimeSeriesOutcome,
14     perform_experiments,
15 )
16 from ema_workbench.util import ema_logging
17
18
19 class ScarcityModel(VensimModel):
20     def returnsToScale(self, x, speed, scale):
21         return (x * 1000, scale * 1 / (1 + exp(-1 * speed * (x - 50))))
22
23     def approxLearning(self, x, speed, scale, start):
24         x = x - start
25         loc = 1 - scale
26         a = (x * 10000, scale * 1 / (1 + exp(speed * x)) + loc)
27         return a
28
29     def f(self, x, speed, loc):
30         return (x / 10, loc * 1 / (1 + exp(speed * x)))
31
32     def priceSubstite(self, x, speed, begin, end):
33         scale = 2 * end
34         start = begin - scale / 2
35
36         return (x + 2000, scale * 1 / (1 + exp(-1 * speed * x)) + start)
37
38     def run_model(self, scenario, policy):
39         """Method for running an instantiated model structure"""
40         kwargs = scenario
41         loc = kwargs.pop("lookup shortage loc")
42         speed = kwargs.pop("lookup shortage speed")
43         lookup = [self.f(x / 10, speed, loc) for x in range(0, 100)]
44         kwargs["shortage price effect lookup"] = lookup
45
46         speed = kwargs.pop("lookup price substitute speed")
47         begin = kwargs.pop("lookup price substitute begin")
48         end = kwargs.pop("lookup price substitute end")
49         lookup = [self.priceSubstite(x, speed, begin, end) for x in range(0, 100, 10)]
50         kwargs["relative price substitute lookup"] = lookup
51
52         scale = kwargs.pop("lookup returns to scale speed")
53         speed = kwargs.pop("lookup returns to scale scale")
54         lookup = [self.returnsToScale(x, speed, scale) for x in range(0, 101, 10)]
55         kwargs["returns to scale lookup"] = lookup
56
57         scale = kwargs.pop("lookup approximated learning speed")
58         speed = kwargs.pop("lookup approximated learning scale")
59         start = kwargs.pop("lookup approximated learning start")
60         lookup = [self.approxLearning(x, speed, scale, start) for x in range(0, 101, 10)]
61         kwargs["approximated learning effect lookup"] = lookup
62
63         super().run_model(kwargs, policy)

```

(continues on next page)

(continued from previous page)

```

64
65
66 if __name__ == "__main__":
67     ema_logging.log_to_stderr(ema_logging.DEBUG)
68
69     model = ScarcityModel("scarcity", wd=r"./models/scarcity", model_file=r"\MetalsEMA.
    ↪vpm")
70
71     model.outcomes = [
72         TimeSeriesOutcome("relative market price"),
73         TimeSeriesOutcome("supply demand ratio"),
74         TimeSeriesOutcome("real annual demand"),
75         TimeSeriesOutcome("produced of intrinsically demanded"),
76         TimeSeriesOutcome("supply"),
77         TimeSeriesOutcome("Installed Recycling Capacity"),
78         TimeSeriesOutcome("Installed Extraction Capacity"),
79     ]
80
81     model.uncertainties = [
82         RealParameter("price elasticity of demand", 0, 0.5),
83         RealParameter("fraction of maximum extraction capacity used", 0.6, 1.2),
84         RealParameter("initial average recycling cost", 1, 4),
85         RealParameter("exogenously planned extraction capacity", 0, 15000),
86         RealParameter("absolute recycling loss fraction", 0.1, 0.5),
87         RealParameter("normal profit margin", 0, 0.4),
88         RealParameter("initial annual supply", 100000, 120000),
89         RealParameter("initial in goods", 1500000, 2500000),
90         RealParameter("average construction time extraction capacity", 1, 10),
91         RealParameter("average lifetime extraction capacity", 20, 40),
92         RealParameter("average lifetime recycling capacity", 20, 40),
93         RealParameter("initial extraction capacity under construction", 5000, 20000),
94         RealParameter("initial recycling capacity under construction", 5000, 20000),
95         RealParameter("initial recycling infrastructure", 5000, 20000),
96         # order of delay
97         CategoricalParameter("order in goods delay", (1, 4, 10, 1000)),
98         CategoricalParameter("order recycling capacity delay", (1, 4, 10)),
99         CategoricalParameter("order extraction capacity delay", (1, 4, 10)),
100        # uncertainties associated with lookups
101        RealParameter("lookup shortage loc", 20, 50),
102        RealParameter("lookup shortage speed", 1, 5),
103        RealParameter("lookup price substitute speed", 0.1, 0.5),
104        RealParameter("lookup price substitute begin", 3, 7),
105        RealParameter("lookup price substitute end", 15, 25),
106        RealParameter("lookup returns to scale speed", 0.01, 0.2),
107        RealParameter("lookup returns to scale scale", 0.3, 0.7),
108        RealParameter("lookup approximated learning speed", 0.01, 0.2),
109        RealParameter("lookup approximated learning scale", 0.3, 0.6),
110        RealParameter("lookup approximated learning start", 30, 60),
111    ]
112
113    results = perform_experiments(model, 50)

```

1.9.18 feature_scoring_flu.py

```

1  """
2  Created on 30 Oct 2018
3
4  @author: jhkwakkel
5  """
6  import matplotlib.pyplot as plt
7  import numpy as np
8  import seaborn as sns
9
10 from ema_workbench import ema_logging, load_results
11 from ema_workbench.analysis import feature_scoring
12
13 ema_logging.log_to_stderr(level=ema_logging.INFO)
14
15 # load data
16 fn = r"./data/1000 flu cases with policies.tar.gz"
17 x, outcomes = load_results(fn)
18
19 # we have timeseries so we need scalars
20 y = {
21     "deceased population": outcomes["deceased population region 1"][:, -1],
22     "max. infected fraction": np.max(outcomes["infected fraction R1"], axis=1),
23 }
24
25 scores = feature_scoring.get_feature_scores_all(x, y)
26 sns.heatmap(scores, annot=True, cmap="viridis")
27 plt.show()

```

1.9.19 feature_scoring_flu_confidence.py

```

1  """
2  Created on 30 Oct 2018
3
4  @author: jhkwakkel
5  """
6  import matplotlib.pyplot as plt
7  import numpy as np
8  import pandas as pd
9  import seaborn as sns
10
11 from ema_workbench import ema_logging, load_results
12 from ema_workbench.analysis.feature_scoring import get_ex_feature_scores, RuleInductionType
13
14 ema_logging.log_to_stderr(level=ema_logging.INFO)
15
16 # load data
17 fn = r"./data/1000 flu cases no policy.tar.gz"
18 x, outcomes = load_results(fn)

```

(continues on next page)

(continued from previous page)

```

19
20 x = x.drop(["model", "policy"], axis=1)
21 y = np.max(outcomes["infected fraction R1"], axis=1)
22
23 all_scores = []
24 for i in range(100):
25     indices = np.random.choice(np.arange(0, x.shape[0]), size=x.shape[0])
26     selected_x = x.iloc[indices, :]
27     selected_y = y[indices]
28
29     scores = get_ex_feature_scores(selected_x, selected_y, mode=RuleInductionType.
↳ REGRESSION)[0]
30     all_scores.append(scores)
31 all_scores = pd.concat(all_scores, axis=1, sort=False)
32
33 sns.boxplot(data=all_scores.T)
34 plt.show()

```

1.9.20 feature_scoring_flu_overtime.py

```

1  """
2  Created on 30 Oct 2018
3
4  @author: jhkwakkel
5  """
6  import matplotlib.pyplot as plt
7  import pandas as pd
8  import seaborn as sns
9
10 from ema_workbench import ema_logging, load_results
11 from ema_workbench.analysis import get_ex_feature_scores, RuleInductionType
12
13 ema_logging.log_to_stderr(level=ema_logging.INFO)
14
15 # load data
16 fn = r"./data/1000 flu cases no policy.tar.gz"
17
18 x, outcomes = load_results(fn)
19 x = x.drop(["model", "policy"], axis=1)
20
21 y = outcomes["deceased population region 1"]
22
23 all_scores = []
24
25 # we only want to show those uncertainties that are in the top 5
26 # most sensitive parameters at any time step
27 top_5 = set()
28 for i in range(2, y.shape[1], 8):
29     data = y[:, i]
30     scores = get_ex_feature_scores(x, data, mode=RuleInductionType.REGRESSION)[0]

```

(continues on next page)

(continued from previous page)

```

31     # add the top five for this time step to the set of top5s
32     top_5 |= set(scores.nlargest(5, 1).index.values)
33     scores = scores.rename(columns={1: outcomes["TIME"][0, i]})
34     all_scores.append(scores)
35
36 all_scores = pd.concat(all_scores, axis=1, sort=False)
37 all_scores = all_scores.loc[top_5, :]
38
39 fig, ax = plt.subplots()
40 sns.heatmap(all_scores, ax=ax, cmap="viridis")
41 plt.show()

```

1.9.21 optimization_lake_model_dps.py

```

1  """
2  This example replicates Quinn, J.D., Reed, P.M., Keller, K. (2017)
3  Direct policy search for robust multi-objective management of deeply
4  uncertain socio-ecological tipping points. Environmental Modelling &
5  Software 92, 125-141.
6
7  It also show cases how the workbench can be used to apply the MORDM extension
8  suggested by Watson, A.A., Kasprzyk, J.R. (2017) Incorporating deeply uncertain
9  factors into the many objective search process. Environmental Modelling &
10 Software 89, 159-171.
11
12 """
13 import math
14
15 import numpy as np
16 from scipy.optimize import brentq
17
18 from ema_workbench import (
19     Model,
20     RealParameter,
21     ScalarOutcome,
22     Constant,
23     ema_logging,
24     MultiprocessingEvaluator,
25     CategoricalParameter,
26     Scenario,
27 )
28
29 from ema_workbench.em_framework.optimization import ArchiveLogger, EpsilonProgress
30
31 # Created on 1 Jun 2017
32 #
33 # .. codeauthor::jhkwakkel <j.h.kwakkel (at) tudelft (dot) nl>
34
35
36

```

(continues on next page)

(continued from previous page)

```

37 def get_antropogenic_release(xt, c1, c2, r1, r2, w1):
38     """
39
40     Parameters
41     -----
42     xt : float
43         pollution in lake at time t
44     c1 : float
45         center rbf 1
46     c2 : float
47         center rbf 2
48     r1 : float
49         radius rbf 1
50     r2 : float
51         radius rbf 2
52     w1 : float
53         weight of rbf 1
54
55     note:: w2 = 1 - w1
56
57     """
58
59     rule = w1 * (abs(xt - c1) / r1) ** 3 + (1 - w1) * (abs(xt - c2) / r2) ** 3
60     at1 = max(rule, 0.01)
61     at = min(at1, 0.1)
62
63     return at
64
65
66 def lake_problem(
67     b=0.42, # decay rate for P in lake (0.42 = irreversible)
68     q=2.0, # recycling exponent
69     mean=0.02, # mean of natural inflows
70     stdev=0.001, # future utility discount rate
71     delta=0.98, # standard deviation of natural inflows
72     alpha=0.4, # utility from pollution
73     nsamples=100, # Monte Carlo sampling of natural inflows
74     myears=1, # the runtime of the simulation model
75     c1=0.25,
76     c2=0.25,
77     r1=0.5,
78     r2=0.5,
79     w1=0.5,
80 ):
81     Pcrit = brentq(lambda x: x**q / (1 + x**q) - b * x, 0.01, 1.5)
82
83     X = np.zeros(myears)
84     average_daily_P = np.zeros(myears)
85     reliability = 0.0
86     inertia = 0
87     utility = 0
88

```

(continues on next page)

(continued from previous page)

```

89     for _ in range(nsamples):
90         X[0] = 0.0
91         decision = 0.1
92
93         decisions = np.zeros(myears)
94         decisions[0] = decision
95
96         natural_inflows = np.random.lognormal(
97             math.log(mean**2 / math.sqrt(stdev**2 + mean**2)),
98             math.sqrt(math.log(1.0 + stdev**2 / mean**2)),
99             size=myears,
100         )
101
102         for t in range(1, myears):
103             # here we use the decision rule
104             decision = get_antropogenic_release(X[t - 1], c1, c2, r1, r2, w1)
105             decisions[t] = decision
106
107             X[t] = (
108                 (1 - b) * X[t - 1]
109                 + X[t - 1] ** q / (1 + X[t - 1] ** q)
110                 + decision
111                 + natural_inflows[t - 1]
112             )
113             average_daily_P[t] += X[t] / nsamples
114
115             reliability += np.sum(X < Pcrit) / (nsamples * myears)
116             inertia += np.sum(np.absolute(np.diff(decisions) < 0.02)) / (nsamples * myears)
117             utility += np.sum(alpha * decisions * np.power(delta, np.arange(myears))) /
↪ nsamples
118         max_P = np.max(average_daily_P)
119
120     return max_P, utility, inertia, reliability
121
122
123 if __name__ == "__main__":
124     ema_logging.log_to_stderr(ema_logging.INFO)
125
126     # instantiate the model
127     lake_model = Model("lakeproblem", function=lake_problem)
128     # specify uncertainties
129     lake_model.uncertainties = [
130         RealParameter("b", 0.1, 0.45),
131         RealParameter("q", 2.0, 4.5),
132         RealParameter("mean", 0.01, 0.05),
133         RealParameter("stdev", 0.001, 0.005),
134         RealParameter("delta", 0.93, 0.99),
135     ]
136
137     # set levers
138     lake_model.levers = [
139         RealParameter("c1", -2, 2),

```

(continues on next page)

(continued from previous page)

```

140     RealParameter("c2", -2, 2),
141     RealParameter("r1", 0, 2),
142     RealParameter("r2", 0, 2),
143     CategoricalParameter("w1", np.linspace(0, 1, 10)),
144 ]
145 # specify outcomes
146 lake_model.outcomes = [
147     ScalarOutcome("max_P", kind=ScalarOutcome.MINIMIZE),
148     ScalarOutcome("utility", kind=ScalarOutcome.MAXIMIZE),
149     ScalarOutcome("inertia", kind=ScalarOutcome.MAXIMIZE),
150     ScalarOutcome("reliability", kind=ScalarOutcome.MAXIMIZE),
151 ]
152
153 # override some of the defaults of the model
154 lake_model.constants = [
155     Constant("alpha", 0.41),
156     Constant("nsamples", 100),
157     Constant("myears", 100),
158 ]
159
160 # reference is optional, but can be used to implement search for
161 # various user specified scenarios along the lines suggested by
162 # Watson and Kasprzyk (2017)
163 reference = Scenario("reference", b=0.4, q=2, mean=0.02, stdev=0.01)
164
165 convergence_metrics = [
166     ArchiveLogger(
167         "./data",
168         [l.name for l in lake_model.levers],
169         [o.name for o in lake_model.outcomes],
170         base_filename="lake_model_dps_archive.tar.gz",
171     ),
172     EpsilonProgress(),
173 ]
174
175 with MultiprocessingEvaluator(lake_model) as evaluator:
176     results, convergence = evaluator.optimize(
177         searchover="levers",
178         nfe=1000000,
179         epsilons=[0.1] * len(lake_model.outcomes),
180         reference=reference,
181         convergence=convergence_metrics,
182     )

```

1.9.22 optimization_lake_model_intertemporal.py

```

1  """
2  An example of the lake problem using the ema workbench.
3
4  The model itself is adapted from the Rhodium example by Dave Hadka,
5  see https://gist.github.com/dhadka/a8d7095c98130d8f73bc
6
7  """
8  import math
9
10 import numpy as np
11 from scipy.optimize import brentq
12
13 from ema_workbench import (
14     Model,
15     RealParameter,
16     ScalarOutcome,
17     ema_logging,
18     MultiprocessingEvaluator,
19     Constraint,
20 )
21 from ema_workbench.em_framework.optimization import HyperVolume, EpsilonProgress
22
23
24 def lake_problem(
25     b=0.42,
26     q=2.0,
27     mean=0.02,
28     stdev=0.0017,
29     delta=0.98,
30     alpha=0.4,
31     nsamples=100,
32     l0=0,
33     l1=0,
34     l2=0,
35     l3=0,
36     l4=0,
37     l5=0,
38     l6=0,
39     l7=0,
40     l8=0,
41     l9=0,
42     l10=0,
43     l11=0,
44     l12=0,
45     l13=0,
46     l14=0,
47     l15=0,
48     l16=0,
49     l17=0,
50     l18=0,
51     l19=0,

```

(continues on next page)

(continued from previous page)

```
52 120=0,  
53 121=0,  
54 122=0,  
55 123=0,  
56 124=0,  
57 125=0,  
58 126=0,  
59 127=0,  
60 128=0,  
61 129=0,  
62 130=0,  
63 131=0,  
64 132=0,  
65 133=0,  
66 134=0,  
67 135=0,  
68 136=0,  
69 137=0,  
70 138=0,  
71 139=0,  
72 140=0,  
73 141=0,  
74 142=0,  
75 143=0,  
76 144=0,  
77 145=0,  
78 146=0,  
79 147=0,  
80 148=0,  
81 149=0,  
82 150=0,  
83 151=0,  
84 152=0,  
85 153=0,  
86 154=0,  
87 155=0,  
88 156=0,  
89 157=0,  
90 158=0,  
91 159=0,  
92 160=0,  
93 161=0,  
94 162=0,  
95 163=0,  
96 164=0,  
97 165=0,  
98 166=0,  
99 167=0,  
100 168=0,  
101 169=0,  
102 170=0,  
103 171=0,
```

(continues on next page)

(continued from previous page)

```
104     172=0,
105     173=0,
106     174=0,
107     175=0,
108     176=0,
109     177=0,
110     178=0,
111     179=0,
112     180=0,
113     181=0,
114     182=0,
115     183=0,
116     184=0,
117     185=0,
118     186=0,
119     187=0,
120     188=0,
121     189=0,
122     190=0,
123     191=0,
124     192=0,
125     193=0,
126     194=0,
127     195=0,
128     196=0,
129     197=0,
130     198=0,
131     199=0,
132 ):
133     decisions = np.array(
134         [
135             10,
136             11,
137             12,
138             13,
139             14,
140             15,
141             16,
142             17,
143             18,
144             19,
145             110,
146             111,
147             112,
148             113,
149             114,
150             115,
151             116,
152             117,
153             118,
154             119,
155             120,
```

(continues on next page)

(continued from previous page)

156	121,
157	122,
158	123,
159	124,
160	125,
161	126,
162	127,
163	128,
164	129,
165	130,
166	131,
167	132,
168	133,
169	134,
170	135,
171	136,
172	137,
173	138,
174	139,
175	140,
176	141,
177	142,
178	143,
179	144,
180	145,
181	146,
182	147,
183	148,
184	149,
185	150,
186	151,
187	152,
188	153,
189	154,
190	155,
191	156,
192	157,
193	158,
194	159,
195	160,
196	161,
197	162,
198	163,
199	164,
200	165,
201	166,
202	167,
203	168,
204	169,
205	170,
206	171,
207	172,

(continues on next page)

(continued from previous page)

```

208         173,
209         174,
210         175,
211         176,
212         177,
213         178,
214         179,
215         180,
216         181,
217         182,
218         183,
219         184,
220         185,
221         186,
222         187,
223         188,
224         189,
225         190,
226         191,
227         192,
228         193,
229         194,
230         195,
231         196,
232         197,
233         198,
234         199,
235     ]
236 )
237
238 Pcrit = brentq(lambda x: x**q / (1 + x**q) - b * x, 0.01, 1.5)
239 nvars = len(decisions)
240 X = np.zeros((nvars,))
241 average_daily_P = np.zeros((nvars,))
242 decisions = np.array(decisions)
243 reliability = 0.0
244
245 for _ in range(nsamples):
246     X[0] = 0.0
247
248     natural_inflows = np.random.lognormal(
249         math.log(mean**2 / math.sqrt(stdev**2 + mean**2)),
250         math.sqrt(math.log(1.0 + stdev**2 / mean**2)),
251         size=nvars,
252     )
253
254     for t in range(1, nvars):
255         X[t] = (
256             (1 - b) * X[t - 1]
257             + X[t - 1] ** q / (1 + X[t - 1] ** q)
258             + decisions[t - 1]
259             + natural_inflows[t - 1]

```

(continues on next page)

(continued from previous page)

```

260         )
261         average_daily_P[t] += X[t] / float(nsamples)
262
263         reliability += np.sum(X < Pcrit) / float(nsamples * nvars)
264
265     max_P = np.max(average_daily_P)
266     utility = np.sum(alpha * decisions * np.power(delta, np.arange(nvars)))
267     inertia = np.sum(np.abs(np.diff(decisions)) > 0.02) / float(nvars - 1)
268
269     return max_P, utility, inertia, reliability
270
271
272 if __name__ == "__main__":
273     ema_logging.log_to_stderr(ema_logging.INFO)
274
275     # instantiate the model
276     lake_model = Model("lakeproblem", function=lake_problem)
277     lake_model.time_horizon = 100 # used to specify the number of timesteps
278
279     # specify uncertainties
280     lake_model.uncertainties = [
281         RealParameter("mean", 0.01, 0.05),
282         RealParameter("stdev", 0.001, 0.005),
283         RealParameter("b", 0.1, 0.45),
284         RealParameter("q", 2.0, 4.5),
285         RealParameter("delta", 0.93, 0.99),
286     ]
287
288     # set levers, one for each time step
289     lake_model.levers = [RealParameter(f"l{i}", 0, 0.1) for i in range(lake_model.time_
↪ horizon)]
290
291     # specify outcomes
292     # specify outcomes
293     lake_model.outcomes = [
294         ScalarOutcome("max_P", kind=ScalarOutcome.MINIMIZE, expected_range=(0, 5)),
295         ScalarOutcome("utility", kind=ScalarOutcome.MAXIMIZE, expected_range=(0, 2)),
296         ScalarOutcome("inertia", kind=ScalarOutcome.MAXIMIZE, expected_range=(0, 1)),
297         ScalarOutcome("reliability", kind=ScalarOutcome.MAXIMIZE, expected_range=(0, 1)),
298     ]
299
300     convergence_metrics = [EpsilonProgress()]
301
302     constraints = [
303         Constraint("max pollution", outcome_names="max_P", function=lambda x: max(0, x -
↪ 5))
304     ]
305
306     with MultiprocessingEvaluator(lake_model) as evaluator:
307         results, convergence = evaluator.optimize(
308             nfe=5000,
309             searchover="levers",

```

(continues on next page)

(continued from previous page)

```

310         epsilons=[0.125, 0.05, 0.01, 0.01],
311         convergence=convergence_metrics,
312         constraints=constraints,
313     )

```

1.9.23 outputspace_exploration_lake_model.py

```

1  """
2  An example of using output space exploration on the lake problem
3
4  The lake problem itself is adapted from the Rhodium example by Dave Hadka,
5  see https://gist.github.com/dhadka/a8d7095c98130d8f73bc
6
7  """
8  import math
9
10 import numpy as np
11 from scipy.optimize import brentq
12
13 from ema_workbench import (
14     Model,
15     RealParameter,
16     ScalarOutcome,
17     Constant,
18     ema_logging,
19     MultiprocessingEvaluator,
20     Policy,
21 )
22
23 from ema_workbench.em_framework.outputspace_exploration import OutputSpaceExploration
24
25
26 def lake_problem(
27     b=0.42, # decay rate for P in lake (0.42 = irreversible)
28     q=2.0, # recycling exponent
29     mean=0.02, # mean of natural inflows
30     stdev=0.001, # future utility discount rate
31     delta=0.98, # standard deviation of natural inflows
32     alpha=0.4, # utility from pollution
33     nsamples=100, # Monte Carlo sampling of natural inflows
34     **kwargs,
35 ):
36     try:
37         decisions = [kwargs[str(i)] for i in range(100)]
38     except KeyError:
39         decisions = [
40             0,
41             ] * 100
42
43     Pcrit = brentq(lambda x: x**q / (1 + x**q) - b * x, 0.01, 1.5)

```

(continues on next page)

(continued from previous page)

```

44     nvars = len(decisions)
45     X = np.zeros((nvars,))
46     average_daily_P = np.zeros((nvars,))
47     decisions = np.array(decisions)
48     reliability = 0.0
49
50     for _ in range(nsamples):
51         X[0] = 0.0
52
53         natural_inflows = np.random.lognormal(
54             math.log(mean**2 / math.sqrt(stdev**2 + mean**2)),
55             math.sqrt(math.log(1.0 + stdev**2 / mean**2)),
56             size=nvars,
57         )
58
59         for t in range(1, nvars):
60             X[t] = (
61                 (1 - b) * X[t - 1]
62                 + X[t - 1] ** q / (1 + X[t - 1] ** q)
63                 + decisions[t - 1]
64                 + natural_inflows[t - 1]
65             )
66             average_daily_P[t] += X[t] / float(nsamples)
67
68             reliability += np.sum(X < Pcrit) / float(nsamples * nvars)
69
70     max_P = np.max(average_daily_P)
71     utility = np.sum(alpha * decisions * np.power(delta, np.arange(nvars)))
72     inertia = np.sum(np.absolute(np.diff(decisions)) < 0.02) / float(nvars - 1)
73
74     return max_P, utility, inertia, reliability
75
76
77 if __name__ == "__main__":
78     ema_logging.log_to_stderr(ema_logging.INFO)
79
80     # instantiate the model
81     lake_model = Model("lakeproblem", function=lake_problem)
82     lake_model.time_horizon = 100
83
84     # specify uncertainties
85     lake_model.uncertainties = [
86         RealParameter("b", 0.1, 0.45),
87         RealParameter("q", 2.0, 4.5),
88         RealParameter("mean", 0.01, 0.05),
89         RealParameter("stdev", 0.001, 0.005),
90         RealParameter("delta", 0.93, 0.99),
91     ]
92
93     # set levers, one for each time step
94     lake_model.levers = [RealParameter(str(i), 0, 0.1) for i in range(lake_model.time_
↪ horizon)]

```

(continues on next page)

(continued from previous page)

```

95
96 # specify outcomes
97 # note that outcomes of kind INFO will be ignored
98 lake_model.outcomes = [
99     ScalarOutcome("max_P", kind=ScalarOutcome.MAXIMIZE),
100     ScalarOutcome("utility", kind=ScalarOutcome.MAXIMIZE),
101     ScalarOutcome("inertia", kind=ScalarOutcome.MAXIMIZE),
102     ScalarOutcome("reliability", kind=ScalarOutcome.MAXIMIZE),
103 ]
104
105 # override some of the defaults of the model
106 lake_model.constants = [Constant("alpha", 0.41), Constant("nsamples", 150)]
107
108 # generate a reference policy
109 n_scenarios = 1000
110 reference = Policy("nopolicy", **{l.name: 0.02 for l in lake_model.levers})
111
112 # we are doing output space exploration given a reference
113 # policy, so we are exploring the output space over the uncertainties
114 # grid spec specifies the grid structure imposed on the output space
115 # each tuple is associated with an outcome. It gives the minimum
116 # maximum, and epsilon value.
117 with MultiprocessingEvaluator(lake_model) as evaluator:
118     res = evaluator.optimize(
119         algorithm=OutputSpaceExploration,
120         grid_spec=[
121             (0, 12, 0.5),
122             (0, 1, 0.05),
123             (0, 1, 0.1),
124             (0, 1, 0.1),
125         ],
126         nfe=1000,
127         searchover="uncertainties",
128         reference=reference,
129     )

```

1.9.24 plotting_envelopes_flu.py

```

1 """
2 Created on Jul 8, 2014
3
4 @author: jhkwakkel@tudelft.net
5 """
6 import matplotlib.pyplot as plt
7
8 from ema_workbench import ema_logging, load_results
9 from ema_workbench.analysis import envelopes, Density
10
11 ema_logging.log_to_stderr(ema_logging.INFO)
12

```

(continues on next page)

(continued from previous page)

```

13 file_name = r"./data/1000 flu cases with policies.tar.gz"
14 experiments, outcomes = load_results(file_name)
15
16 # the plotting functions return the figure and a dict of axes
17 fig, axes = envelopes(experiments, outcomes, group_by="policy", density=Density.KDE,
18                       ↪fill=True)
19
20 # we can access each of the axes and make changes
21 for key, value in axes.items():
22     # the key is the name of the outcome for the normal plot
23     # and the name plus '_density' for the endstate distribution
24     if key.endswith("_density"):
25         value.set_xscale("log")
26
27 plt.show()

```

1.9.25 plotting_envelopes_with_lines_flu.py

```

1  """
2  Created on Jul 8, 2014
3
4  @author: jhkwakkel@tudelft.net
5  """
6  import matplotlib.pyplot as plt
7  import numpy as np
8
9  from ema_workbench import ema_logging, load_results
10 from ema_workbench.analysis import lines, Density
11
12 ema_logging.log_to_stderr(ema_logging.INFO)
13
14 file_name = r"./data/1000 flu cases with policies.tar.gz"
15 experiments, outcomes = load_results(file_name)
16
17 # let's specify a few scenarios that we want to show for
18 # each of the policies
19 scenario_ids = np.arange(0, 1000, 100)
20 experiments_to_show = experiments["scenario_id"].isin(scenario_ids)
21
22 # the plotting functions return the figure and a dict of axes
23 fig, axes = lines(
24     experiments,
25     outcomes,
26     group_by="policy",
27     density=Density.KDE,
28     show_envelope=True,
29     experiments_to_show=experiments_to_show,
30 )
31
32 # we can access each of the axes and make changes

```

(continues on next page)

(continued from previous page)

```

33 for key, value in axes.items():
34     # the key is the name of the outcome for the normal plot
35     # and the name plus '_density' for the endstate distribution
36     if key.endswith("_density"):
37         value.set_xscale("log")
38
39 plt.show()

```

1.9.26 plotting_kdeovertime_flu.py

```

1  """
2  Created on Jul 8, 2014
3
4  @author: jhkwakkel@tudelft.net
5  """
6  import matplotlib.pyplot as plt
7
8  from ema_workbench import ema_logging, load_results
9  from ema_workbench.analysis.plotting import kde_over_time
10
11 ema_logging.log_to_stderr(ema_logging.INFO)
12
13 # file_name = r'./data/1000 runs scarcity.tar.gz'
14 file_name = "./data/1000 flu cases no policy.tar.gz"
15 experiments, outcomes = load_results(file_name)
16
17 # the plotting functions return the figure and a dict of axes
18 fig, axes = kde_over_time(experiments, outcomes, log=True)
19
20 plt.show()

```

1.9.27 plotting_lines_flu.py

```

1  """
2  Created on Jul 8, 2014
3
4  @author: jhkwakkel@tudelft.net
5  """
6  import matplotlib.pyplot as plt
7
8  from ema_workbench import ema_logging, load_results
9  from ema_workbench.analysis import lines, Density
10
11 ema_logging.log_to_stderr(ema_logging.INFO)
12
13 file_name = r"./data/1000 flu cases no policy.tar.gz"
14 experiments, outcomes = load_results(file_name)
15
16 # the plotting functions return the figure and a dict of axes

```

(continues on next page)

(continued from previous page)

```

17 fig, axes = lines(experiments, outcomes, density=Density.VIOLIN)
18
19 # we can access each of the axes and make changes
20 for key, value in axes.items():
21     # the key is the name of the outcome for the normal plot
22     # and the name plus '_density' for the endstate distribution
23     if key.endswith("_density"):
24         value.set_xscale("log")
25
26 plt.show()

```

1.9.28 plotting_multiple_densities_flu.py

```

1  """
2  Created on Jul 8, 2014
3
4  @author: jhkwakkel@tudelft.net
5  """
6  import math
7
8  import matplotlib.pyplot as plt
9
10 from ema_workbench import ema_logging, load_results
11 from ema_workbench.analysis import multiple_densities, Density
12
13 ema_logging.log_to_stderr(ema_logging.INFO)
14
15 file_name = "./data/1000 flu cases with policies.tar.gz"
16 experiments, outcomes = load_results(file_name)
17
18 # pick points in time for which we want to see a
19 # density subplot
20 time = outcomes["TIME"][0, :]
21 times = time[1 :: math.ceil(time.shape[0] / 6)].tolist()
22
23 multiple_densities(
24     experiments,
25     outcomes,
26     log=True,
27     points_in_time=times,
28     group_by="policy",
29     density=Density.KDE,
30     fill=True,
31 )
32
33 plt.show()

```

1.9.29 plotting_pairsplot_flu.py

```

1  """
2  Created on 20 sep. 2011
3
4  .. codeauthor:: jhkwakkel <j.h.kwakkel (at) tudelft (dot) nl>
5  """
6  import matplotlib.pyplot as plt
7  import numpy as np
8
9  from ema_workbench import load_results, ema_logging
10 from ema_workbench.analysis.pairs_plotting import pairs_lines, pairs_scatter, pairs_
    ↪ density
11
12 ema_logging.log_to_stderr(level=ema_logging.DEFAULT_LEVEL)
13
14 # load the data
15 fh = "./data/1000 flu cases no policy.tar.gz"
16 experiments, outcomes = load_results(fh)
17
18 # transform the results to the required format
19 # that is, we want to know the max peak and the casualties at the end of the
20 # run
21 tr = {}
22
23 # get time and remove it from the dict
24 time = outcomes.pop("TIME")
25
26 for key, value in outcomes.items():
27     if key == "deceased population region 1":
28         tr[key] = value[:, -1] # we want the end value
29     else:
30         # we want the maximum value of the peak
31         max_peak = np.max(value, axis=1)
32         tr["max peak"] = max_peak
33
34         # we want the time at which the maximum occurred
35         # the code here is a bit obscure, I don't know why the transpose
36         # of value is needed. This however does produce the appropriate results
37         logical = value.T == np.max(value, axis=1)
38         tr["time of max"] = time[logical.T]
39
40 pairs_scatter(experiments, tr, filter_scalar=False)
41 pairs_lines(experiments, outcomes)
42 pairs_density(experiments, tr, filter_scalar=False)
43 plt.show()

```

1.9.30 robust_optimization_lake_model_dps.py

```

1  """
2  This example takes the direct policy search formulation of the lake problem as
3  found in Quinn et al (2017), but embeds in in a robust optimization.
4
5  Quinn, J.D., Reed, P.M., Keller, K. (2017)
6  Direct policy search for robust multi-objective management of deeply
7  uncertain socio-ecological tipping points. Environmental Modelling &
8  Software 92, 125-141.
9
10 """
11
12 import math
13
14 import numpy as np
15 from scipy.optimize import brentq
16
17 from ema_workbench import (
18     Model,
19     RealParameter,
20     ScalarOutcome,
21     Constant,
22     ema_logging,
23     MultiprocessingEvaluator,
24 )
25 from ema_workbench.em_framework.samplers import sample_uncertainties
26
27 # Created on 1 Jun 2017
28 #
29 # .. codeauthor::jhwakkel <j.h.kwakkel (at) tudelft (dot) nl>
30
31 __all__ = []
32
33
34 def get_antropogenic_release(xt, c1, c2, r1, r2, w1):
35     """
36
37     Parameters
38     -----
39     xt : float
40         pollution in lake at time t
41     c1 : float
42         center rbf 1
43     c2 : float
44         center rbf 2
45     r1 : float
46         radius rbf 1
47     r2 : float
48         radius rbf 2
49     w1 : float
50         weight of rbf 1
51

```

(continues on next page)

(continued from previous page)

```

52     note:: w2 = 1 - w1
53
54     """
55
56     rule = w1 * (abs(xt - c1 / r1)) ** 3 + (1 - w1) * (abs(xt - c2 / r2) ** 3)
57     at1 = max(rule, 0.01)
58     at = min(at1, 0.1)
59
60     return at
61
62
63 def lake_problem(
64     b=0.42, # decay rate for P in lake (0.42 = irreversible)
65     q=2.0, # recycling exponent
66     mean=0.02, # mean of natural inflows
67     stdev=0.001, # future utility discount rate
68     delta=0.98, # standard deviation of natural inflows
69     alpha=0.4, # utility from pollution
70     nsamples=100, # Monte Carlo sampling of natural inflows
71     myears=1, # the runtime of the simulation model
72     c1=0.25,
73     c2=0.25,
74     r1=0.5,
75     r2=0.5,
76     w1=0.5,
77 ):
78     Pcrit = brentq(lambda x: x**q / (1 + x**q) - b * x, 0.01, 1.5)
79
80     X = np.zeros(myears)
81     average_daily_P = np.zeros(myears)
82     reliability = 0.0
83     inertia = 0
84     utility = 0
85
86     for _ in range(nsamples):
87         X[0] = 0.0
88         decision = 0.1
89
90         decisions = np.zeros(myears)
91         decisions[0] = decision
92
93         natural_inflows = np.random.lognormal(
94             math.log(mean**2 / math.sqrt(stdev**2 + mean**2)),
95             math.sqrt(math.log(1.0 + stdev**2 / mean**2)),
96             size=myears,
97         )
98
99         for t in range(1, myears):
100             # here we use the decision rule
101             decision = get_antropogenic_release(X[t - 1], c1, c2, r1, r2, w1)
102             decisions[t] = decision
103

```

(continues on next page)

(continued from previous page)

```

104         X[t] = (
105             (1 - b) * X[t - 1]
106             + X[t - 1] ** q / (1 + X[t - 1] ** q)
107             + decision
108             + natural_inflows[t - 1]
109         )
110         average_daily_P[t] += X[t] / nsamples
111
112         reliability += np.sum(X < Pcrit) / (nsamples * myears)
113         inertia += np.sum(np.absolute(np.diff(decisions) < 0.02)) / (nsamples * myears)
114         utility += np.sum(alpha * decisions * np.power(delta, np.arange(myears))) /
↪ nsamples
115         max_P = np.max(average_daily_P)
116
117     return max_P, utility, inertia, reliability
118
119
120 if __name__ == "__main__":
121     ema_logging.log_to_stderr(ema_logging.INFO)
122
123     # instantiate the model
124     lake_model = Model("lakeproblem", function=lake_problem)
125     # specify uncertainties
126     lake_model.uncertainties = [
127         RealParameter("b", 0.1, 0.45),
128         RealParameter("q", 2.0, 4.5),
129         RealParameter("mean", 0.01, 0.05),
130         RealParameter("stdev", 0.001, 0.005),
131         RealParameter("delta", 0.93, 0.99),
132     ]
133
134     # set levers
135     lake_model.levers = [
136         RealParameter("c1", -2, 2),
137         RealParameter("c2", -2, 2),
138         RealParameter("r1", 0, 2),
139         RealParameter("r2", 0, 2),
140         RealParameter("w1", 0, 1),
141     ]
142
143     # specify outcomes
144     lake_model.outcomes = [
145         ScalarOutcome("max_P"),
146         ScalarOutcome("utility"),
147         ScalarOutcome("inertia"),
148         ScalarOutcome("reliability"),
149     ]
150
151     # override some of the defaults of the model
152     lake_model.constants = [
153         Constant("alpha", 0.41),
154         Constant("nsamples", 100),

```

(continues on next page)

(continued from previous page)

```

155     Constant("myyears", 100),
156 ]
157
158 # setup and execute the robust optimization
159 def signal_to_noise(data):
160     mean = np.mean(data)
161     std = np.std(data)
162     sn = mean / std
163     return sn
164
165 MAXIMIZE = ScalarOutcome.MAXIMIZE # @UndefinedVariable
166 MINIMIZE = ScalarOutcome.MINIMIZE # @UndefinedVariable
167 robustnes_functions = [
168     ScalarOutcome("mean p", kind=MINIMIZE, variable_name="max_P", function=np.mean),
169     ScalarOutcome("std p", kind=MINIMIZE, variable_name="max_P", function=np.std),
170     ScalarOutcome(
171         "sn reliability", kind=MAXIMIZE, variable_name="reliability",
172         ↪function=signal_to_noise
173     ),
174 ]
175 n_scenarios = 10
176 scenarios = sample_uncertainties(lake_model, n_scenarios)
177 nfe = 10000
178
179 with MultiprocessingEvaluator(lake_model) as evaluator:
180     evaluator.robust_optimize(
181         robustnes_functions,
182         scenarios,
183         nfe=nfe,
184         epsilons=[0.1] * len(robustnes_functions),
185         population_size=5,
186     )

```

1.9.31 sample_jointly_lake_model.py

```

1  """
2  An example of the lake problem using the ema workbench. This example
3  illustrated how you can control more finely how samples are being generated.
4  In this particular case, we want to apply Sobol analysis over both the
5  uncertainties and levers at the same time.
6
7  """
8  import math
9
10 import numpy as np
11 import pandas as pd
12 from SALib.analyze import sobol
13 from scipy.optimize import brentq
14
15 from ema_workbench import (

```

(continues on next page)

(continued from previous page)

```

16     Model,
17     RealParameter,
18     ScalarOutcome,
19     Constant,
20     ema_logging,
21     MultiprocessingEvaluator,
22     Scenario,
23 )
24 from ema_workbench.em_framework import get_SALib_problem, sample_parameters
25 from ema_workbench.em_framework import SobolSampler
26
27 # from ema_workbench.em_framework.evaluators import Samplers
28
29
30 def get_antropogenic_release(xt, c1, c2, r1, r2, w1):
31     """
32
33     Parameters
34     -----
35     xt : float
36         pollution in lake at time t
37     c1 : float
38         center rbf 1
39     c2 : float
40         center rbf 2
41     r1 : float
42         radius rbf 1
43     r2 : float
44         radius rbf 2
45     w1 : float
46         weight of rbf 1
47
48     note:: w2 = 1 - w1
49
50     """
51
52     rule = w1 * (abs(xt - c1) / r1) ** 3 + (1 - w1) * (abs(xt - c2) / r2) ** 3
53     at1 = max(rule, 0.01)
54     at = min(at1, 0.1)
55
56     return at
57
58
59 def lake_problem(
60     b=0.42, # decay rate for P in lake (0.42 = irreversible)
61     q=2.0, # recycling exponent
62     mean=0.02, # mean of natural inflows
63     stdev=0.001, # future utility discount rate
64     delta=0.98, # standard deviation of natural inflows
65     alpha=0.4, # utility from pollution
66     nsamples=100, # Monte Carlo sampling of natural inflows
67     myears=1, # the runtime of the simulation model

```

(continues on next page)

(continued from previous page)

```

68     c1=0.25,
69     c2=0.25,
70     r1=0.5,
71     r2=0.5,
72     w1=0.5,
73 ):
74     Pcrit = brentq(lambda x: x**q / (1 + x**q) - b * x, 0.01, 1.5)
75
76     X = np.zeros((myyears,))
77     average_daily_P = np.zeros((myyears,))
78     reliability = 0.0
79     inertia = 0
80     utility = 0
81
82     for _ in range(nsamples):
83         X[0] = 0.0
84         decision = 0.1
85
86         decisions = np.zeros(myyears)
87         decisions[0] = decision
88
89         natural_inflows = np.random.lognormal(
90             math.log(mean**2 / math.sqrt(stdev**2 + mean**2)),
91             math.sqrt(math.log(1.0 + stdev**2 / mean**2)),
92             size=myyears,
93         )
94
95         for t in range(1, myyears):
96             # here we use the decision rule
97             decision = get_antropogenic_release(X[t - 1], c1, c2, r1, r2, w1)
98             decisions[t] = decision
99
100            X[t] = (
101                (1 - b) * X[t - 1]
102                + X[t - 1] ** q / (1 + X[t - 1] ** q)
103                + decision
104                + natural_inflows[t - 1]
105            )
106            average_daily_P[t] += X[t] / nsamples
107
108            reliability += np.sum(X < Pcrit) / (nsamples * myyears)
109            inertia += np.sum(np.absolute(np.diff(decisions) < 0.02)) / (nsamples * myyears)
110            utility += np.sum(alpha * decisions * np.power(delta, np.arange(myyears))) /
↪ nsamples
111            max_P = np.max(average_daily_P)
112
113            return max_P, utility, inertia, reliability
114
115
116 def analyze(results, ooi):
117     """analyze results using SALib sobol, returns a dataframe"""
118

```

(continues on next page)

(continued from previous page)

```

119     _, outcomes = results
120
121     parameters = lake_model.uncertainties.copy() + lake_model.levers.copy()
122
123     problem = get_SALib_problem(parameters)
124     y = outcomes[ooi]
125     sobol_indices = sobol.analyze(problem, y)
126     sobol_stats = {key: sobol_indices[key] for key in ["ST", "ST_conf", "S1", "S1_conf"]}
127     sobol_stats = pd.DataFrame(sobol_stats, index=problem["names"])
128     sobol_stats.sort_values(by="ST", ascending=False)
129     s2 = pd.DataFrame(sobol_indices["S2"], index=problem["names"], columns=problem["names
130     ↪"])
131     s2_conf = pd.DataFrame(
132         sobol_indices["S2_conf"], index=problem["names"], columns=problem["names"]
133     )
134
135     return sobol_stats, s2, s2_conf
136
137 if __name__ == "__main__":
138     ema_logging.log_to_stderr(ema_logging.INFO)
139
140     # instantiate the model
141     lake_model = Model("lakeproblem", function=lake_problem)
142     # specify uncertainties
143     lake_model.uncertainties = [
144         RealParameter("b", 0.1, 0.45),
145         RealParameter("q", 2.0, 4.5),
146         RealParameter("mean", 0.01, 0.05),
147         RealParameter("stdev", 0.001, 0.005),
148         RealParameter("delta", 0.93, 0.99),
149     ]
150
151     # set levers
152     lake_model.levers = [
153         RealParameter("c1", -2, 2),
154         RealParameter("c2", -2, 2),
155         RealParameter("r1", 0, 2),
156         RealParameter("r2", 0, 2),
157         RealParameter("w1", 0, 1),
158     ]
159     # specify outcomes
160     lake_model.outcomes = [
161         ScalarOutcome("max_P", kind=ScalarOutcome.MINIMIZE),
162         # @UndefinedVariable
163         ScalarOutcome("utility", kind=ScalarOutcome.MAXIMIZE),
164         # @UndefinedVariable
165         ScalarOutcome("inertia", kind=ScalarOutcome.MAXIMIZE),
166         # @UndefinedVariable
167         ScalarOutcome("reliability", kind=ScalarOutcome.MAXIMIZE),
168     ] # @UndefinedVariable
169

```

(continues on next page)

(continued from previous page)

```

170     # override some of the defaults of the model
171     lake_model.constants = [
172         Constant("alpha", 0.41),
173         Constant("nsamples", 100),
174         Constant("myears", 100),
175     ]
176
177     # combine parameters and uncertainties prior to sampling
178     n_scenarios = 1000
179     parameters = lake_model.uncertainties + lake_model.levers
180     scenarios = sample_parameters(parameters, n_scenarios, SobolSampler(), Scenario)
181
182     with MultiprocessingEvaluator(lake_model) as evaluator:
183         results = evaluator.perform_experiments(scenarios)
184
185     sobol_stats, s2, s2_conf = analyze(results, "max_P")
186     print(sobol_stats)
187     print(s2)
188     print(s2_conf)

```

1.9.32 sample_sobol_lake_model.py

```

1  """
2  An example of the lake problem using the ema workbench.
3
4  The model itself is adapted from the Rhodium example by Dave Hadka,
5  see https://gist.github.com/dhadka/a8d7095c98130d8f73bc
6
7  """
8  import math
9
10 import numpy as np
11 import pandas as pd
12 from SALib.analyze import sobol
13 from scipy.optimize import brentq
14
15 from ema_workbench import (
16     Model,
17     RealParameter,
18     ScalarOutcome,
19     Constant,
20     ema_logging,
21     MultiprocessingEvaluator,
22     Policy,
23 )
24 from ema_workbench.em_framework import get_SALib_problem
25 from ema_workbench.em_framework.evaluators import Samplers
26
27
28 def lake_problem(

```

(continues on next page)

(continued from previous page)

```

29     b=0.42, # decay rate for P in lake (0.42 = irreversible)
30     q=2.0, # recycling exponent
31     mean=0.02, # mean of natural inflows
32     stdev=0.001, # future utility discount rate
33     delta=0.98, # standard deviation of natural inflows
34     alpha=0.4, # utility from pollution
35     nsamples=100, # Monte Carlo sampling of natural inflows
36     **kwargs,
37 ):
38     try:
39         decisions = [kwargs[str(i)] for i in range(100)]
40     except KeyError:
41         decisions = [0] * 100
42
43     Pcrit = brentq(lambda x: x**q / (1 + x**q) - b * x, 0.01, 1.5)
44     nvars = len(decisions)
45     X = np.zeros((nvars,))
46     average_daily_P = np.zeros((nvars,))
47     decisions = np.array(decisions)
48     reliability = 0.0
49
50     for _ in range(nsamples):
51         X[0] = 0.0
52
53         natural_inflows = np.random.lognormal(
54             math.log(mean**2 / math.sqrt(stdev**2 + mean**2)),
55             math.sqrt(math.log(1.0 + stdev**2 / mean**2)),
56             size=nvars,
57         )
58
59         for t in range(1, nvars):
60             X[t] = (
61                 (1 - b) * X[t - 1]
62                 + X[t - 1] ** q / (1 + X[t - 1] ** q)
63                 + decisions[t - 1]
64                 + natural_inflows[t - 1]
65             )
66             average_daily_P[t] += X[t] / float(nsamples)
67
68             reliability += np.sum(X < Pcrit) / float(nsamples * nvars)
69
70     max_P = np.max(average_daily_P)
71     utility = np.sum(alpha * decisions * np.power(delta, np.arange(nvars)))
72     inertia = np.sum(np.absolute(np.diff(decisions)) < 0.02) / float(nvars - 1)
73
74     return max_P, utility, inertia, reliability
75
76
77 def analyze(results, ooi):
78     """analyze results using SALib sobol, returns a dataframe"""
79
80     _, outcomes = results

```

(continues on next page)

(continued from previous page)

```

81
82     problem = get_SALib_problem(lake_model.uncertainties)
83     y = outcomes[ooi]
84     sobol_indices = sobol.analyze(problem, y)
85     sobol_stats = {key: sobol_indices[key] for key in ["ST", "ST_conf", "S1", "S1_conf"]}
86     sobol_stats = pd.DataFrame(sobol_stats, index=problem["names"])
87     sobol_stats.sort_values(by="ST", ascending=False)
88     s2 = pd.DataFrame(sobol_indices["S2"], index=problem["names"], columns=problem["names
89 ↪"])
90     s2_conf = pd.DataFrame(
91         sobol_indices["S2_conf"], index=problem["names"], columns=problem["names"]
92     )
93     return sobol_stats, s2, s2_conf
94
95
96 if __name__ == "__main__":
97     ema_logging.log_to_stderr(ema_logging.INFO)
98
99     # instantiate the model
100    lake_model = Model("lakeproblem", function=lake_problem)
101    lake_model.time_horizon = 100
102
103    # specify uncertainties
104    lake_model.uncertainties = [
105        RealParameter("b", 0.1, 0.45),
106        RealParameter("q", 2.0, 4.5),
107        RealParameter("mean", 0.01, 0.05),
108        RealParameter("stdev", 0.001, 0.005),
109        RealParameter("delta", 0.93, 0.99),
110    ]
111
112    # set levers, one for each time step
113    lake_model.levers = [RealParameter(str(i), 0, 0.1) for i in range(lake_model.time_
114 ↪horizon)]
115
116    # specify outcomes
117    lake_model.outcomes = [
118        ScalarOutcome("max_P"),
119        ScalarOutcome("utility"),
120        ScalarOutcome("inertia"),
121        ScalarOutcome("reliability"),
122    ]
123
124    # override some of the defaults of the model
125    lake_model.constants = [Constant("alpha", 0.41), Constant("nsamples", 150)]
126
127    # generate a single default no release policy
128    policy = Policy("no release", **{str(i): 0.1 for i in range(100)})
129
130    n_scenarios = 1000

```

(continues on next page)

(continued from previous page)

```

131     with MultiprocessingEvaluator(lake_model) as evaluator:
132         results = evaluator.perform_experiments(
133             n_scenarios, policy, uncertainty_sampling=Samplers.SOBOL
134         )
135
136     sobol_stats, s2, s2_conf = analyze(results, "max_P")
137     print(sobol_stats)
138     print(s2)
139     print(s2_conf)

```

1.9.33 sd_boostedtrees_flu.py

```

1  """
2
3  """
4
5  import matplotlib.pyplot as plt
6  import numpy as np
7  import seaborn as sns
8  from matplotlib.collections import CircleCollection
9  from sklearn.ensemble import AdaBoostClassifier
10 from sklearn.tree import DecisionTreeClassifier
11
12 from ema_workbench import load_results, ema_logging
13 from ema_workbench.analysis import feature_scoring
14
15 ema_logging.log_to_stderr(ema_logging.INFO)
16
17
18 def plot_factormap(x1, x2, ax, bdt, nominal):
19     """helper function for plotting a 2d factor map"""
20     x_min, x_max = x[:, x1].min(), x[:, x1].max()
21     y_min, y_max = x[:, x2].min(), x[:, x2].max()
22     xx, yy = np.meshgrid(np.linspace(x_min, x_max, 500), np.linspace(y_min, y_max, 500))
23
24     grid = np.ones((xx.ravel().shape[0], x.shape[1])) * nominal
25     grid[:, x1] = xx.ravel()
26     grid[:, x2] = yy.ravel()
27
28     Z = bdt.predict(grid)
29     Z = Z.reshape(xx.shape)
30
31     ax.contourf(xx, yy, Z, cmap=plt.cm.Paired, alpha=0.5) # @UndefinedVariable
32
33     for i in (0, 1):
34         idx = y == i
35         ax.scatter(x[idx, x1], x[idx, x2], s=5)
36     ax.set_xlabel(columns[x1])
37     ax.set_ylabel(columns[x2])
38

```

(continues on next page)

(continued from previous page)

```

39
40 def plot_diag(x1, ax):
41     x_min, x_max = x[:, x1].min(), x[:, x1].max()
42     for i in (0, 1):
43         idx = y == i
44         ax.hist(x[idx, x1], range=(x_min, x_max), alpha=0.5)
45
46
47 # load data
48 experiments, outcomes = load_results("./data/1000 flu cases with policies.tar.gz")
49
50 # transform to numpy array with proper recoding of categorical variables
51 x, columns = feature_scoring._prepare_experiments(experiments)
52 y = outcomes["deceased population region 1"][:, -1] > 1000000
53
54 # establish mean case for factor maps
55 # this is questionable in particular in case of categorical dimensions
56 minima = x.min(axis=0)
57 maxima = x.max(axis=0)
58 nominal = minima + (maxima - minima) / 2
59
60 # fit the boosted tree
61 bdt = AdaBoostClassifier(DecisionTreeClassifier(max_depth=3), algorithm="SAMME", n_
62     ↪ estimators=200)
63 bdt.fit(x, y)
64
65 # determine which dimensions are most important
66 sorted_indices = np.argsort(bdt.feature_importances_)[::-1]
67
68 # do the actual plotting
69 # this is a quick hack, tying it to seaborn Pairgrid is probably
70 # the more elegant solution, but is tricky with what arguments
71 # can be passed to the plotting function
72 fig, axes = plt.subplots(ncols=5, nrows=5, figsize=(15, 15))
73
74 for i, row in enumerate(axes):
75     for j, ax in enumerate(row):
76         if i > j:
77             plot_factormap(sorted_indices[j], sorted_indices[i], ax, bdt, nominal)
78         elif i == j:
79             plot_diag(sorted_indices[j], ax)
80         else:
81             ax.set_xticks([])
82             ax.set_yticks([])
83             ax.axis("off")
84
85         if j > 0:
86             ax.set_yticklabels([])
87             ax.set_ylabel("")
88         if i < len(axes) - 1:
89             ax.set_xticklabels([])
90             ax.set_xlabel("")

```

(continues on next page)

(continued from previous page)

```

90
91 # add the legend
92 # Draw a full-figure legend outside the grid
93 handles = [
94     CircleCollection([10], color=sns.color_palette()[0]),
95     CircleCollection([10], color=sns.color_palette()[1]),
96 ]
97
98 legend = fig.legend(handles, ["False", "True"], scatterpoints=1)
99
100 plt.tight_layout()
101 plt.show()

```

1.9.34 sd_cart_flu.py

```

1  """
2  Created on May 26, 2015
3
4  @author: jhkwakkel
5  """
6  import matplotlib.pyplot as plt
7
8  import ema_workbench.analysis.cart as cart
9  from ema_workbench import ema_logging, load_results
10
11 ema_logging.log_to_stderr(level=ema_logging.INFO)
12
13
14 def classify(data):
15     # get the output for deceased population
16     result = data["deceased population region 1"]
17
18     # if deceased population is higher then 1.000.000 people,
19     # classify as 1
20     classes = result[:, -1] > 1000000
21
22     return classes
23
24
25 # load data
26 fn = "./data/1000 flu cases with policies.tar.gz"
27 results = load_results(fn)
28 experiments, outcomes = results
29
30 # extract results for 1 policy
31 logical = experiments["policy"] == "no policy"
32 new_experiments = experiments[logical]
33 new_outcomes = {}
34 for key, value in outcomes.items():
35     new_outcomes[key] = value[logical]

```

(continues on next page)

(continued from previous page)

```

36
37 results = (new_experiments, new_outcomes)
38
39 # perform cart on modified results tuple
40
41 cart_alg = cart.setup_cart(results, classify, mass_min=0.05)
42 cart_alg.build_tree()
43
44 # print cart to std_out
45 print(cart_alg.stats_to_dataframe())
46 print(cart_alg.bboxes_to_dataframe())
47
48 # visualize
49 cart_alg.show_bboxes(together=False)
50 cart_alg.show_tree()
51 plt.show()

```

1.9.35 sd_cart_wcm.py

```

1  """
2  Created on May 26, 2015
3
4  @author: jhkwakkel
5  """
6  import matplotlib.pyplot as plt
7
8  import ema_workbench.analysis.cart as cart
9  from ema_workbench import ema_logging, load_results
10
11 ema_logging.log_to_stderr(level=ema_logging.INFO)
12
13 default_flow = 2.178849944502783e7
14
15 # load data
16 fn = "./data/5000 runs WCM.tar.gz"
17 results = load_results(fn)
18 x, outcomes = results
19
20 ooi = "throughput Rotterdam"
21 outcome = outcomes[ooi] / default_flow
22 y = outcome < 1
23
24 cart_alg = cart.CART(x, y)
25 cart_alg.build_tree()
26
27 # print cart to std_out
28 print(cart_alg.stats_to_dataframe())
29 print(cart_alg.bboxes_to_dataframe())
30
31 # visualize

```

(continues on next page)

(continued from previous page)

```
32 cart_alg.show_boxes(together=False)
33 cart_alg.show_tree()
34 plt.show()
```

1.9.36 sd_dimensional_stacking_flu.py

```
1  """
2
3  This file illustrated the use of the workbench for using dimensional
4  stacking for scenario discovery
5
6
7  .. codeauthor:: jhkwakkel <j.h.kwakkel (at) tudelft (dot) nl>
8
9  """
10 import matplotlib.pyplot as plt
11
12 from ema_workbench import ema_logging, load_results
13 from ema_workbench.analysis import dimensional_stacking
14
15 ema_logging.log_to_stderr(level=ema_logging.INFO)
16
17 # load data
18 fn = "./data/1000 flu cases no policy.tar.gz"
19 x, outcomes = load_results(fn)
20
21 y = outcomes["deceased population region 1"][:, -1] > 1000000
22
23 fig = dimensional_stacking.create_pivot_plot(x, y, 2, bin_labels=True)
24
25 plt.show()
```

1.9.37 sd_logit_flu_example.py

```
1  """
2
3  """
4  import matplotlib.pyplot as plt
5  import seaborn as sns
6
7  import ema_workbench.analysis.logistic_regression as logistic_regression
8  from ema_workbench import load_results
9
10 # Created on 14 Mar 2019
11 #
12 # .. codeauthor:: jhkwakkel <j.h.kwakkel (at) tudelft (dot) nl>
13
14
15 experiments, outcomes = load_results("./data/1000 flu cases no policy.tar.gz")
```

(continues on next page)

(continued from previous page)

```

16 x = experiments.drop(["model", "policy"], axis=1)
17 y = outcomes["deceased population region 1"][:, -1] > 1000000
18
19
20 logit = logistic_regression.Logit(x, y)
21 logit.run()
22
23 logit.show_tradeoff()
24
25 # when we change the default threshold, the tradeoff curve is
26 # recalculated
27 logit.threshold = 0.8
28 logit.show_tradeoff()
29
30 # we can also look at the tradeoff across threshold values
31 # for a given model
32 logit.show_threshold_tradeoff(3)
33
34 # inspect shows the threshold tradeoff for the model
35 # as well as the statistics of the model
36 logit.inspect(3)
37
38 # we can also visualize the performance of the model
39 # using a pairwise scatter plot
40 sns.set_style("white")
41 logit.plot_pairwise_scatter(3)
42
43 plt.show()

```

1.9.38 sd_prim_PCA_flu.py

```

1  """
2
3  This file illustrated the use of the workbench for doing
4  a PRIM analysis with PCA preprocessing
5
6  The data was generated using a system dynamics models implemented in Vensim.
7  See flu_example.py for the code.
8
9
10 """
11 import matplotlib.pyplot as plt
12
13 import ema_workbench.analysis.prim as prim
14 from ema_workbench import ema_logging, load_results
15
16 #
17 # .. codeauthor:: jhkwakkel <j.h.kwakkel (at) tudelft (dot) nl>
18
19 ema_logging.log_to_stderr(level=ema_logging.INFO)

```

(continues on next page)

(continued from previous page)

```

20
21 # load data
22 fn = r"./data/1000 flu cases no policy.tar.gz"
23 x, outcomes = load_results(fn)
24
25 # specify y
26 y = outcomes["deceased population region 1"][:, -1] > 1000000
27
28 rotated_experiments, rotation_matrix = prim.pca_preprocess(x, y, exclude=["model",
29 ↪ "policy"])
30
31 # perform prim on modified results tuple
32 prim_obj = prim.Prim(rotated_experiments, y, threshold=0.8)
33 box = prim_obj.find_box()
34
35 box.show_tradeoff()
36 box.inspect(22)
37 plt.show()

```

1.9.39 sd_prim_byrant_and_lempert.py

```

1  """
2  Created on 12 Nov 2018
3
4  @author: jhkwakkel
5  """
6  import matplotlib.pyplot as plt
7  import pandas as pd
8
9  from ema_workbench.analysis import prim
10 from ema_workbench.util import ema_logging
11
12 ema_logging.log_to_stderr(ema_logging.INFO)
13
14 data = pd.read_csv("./data/bryant et al 2010 data.csv", index_col=False)
15 x = data.iloc[:, 2:11]
16 y = data.iloc[:, 15].values
17
18 prim_alg = prim.Prim(x, y, threshold=0.8, peel_alpha=0.1)
19 box1 = prim_alg.find_box()
20
21 box1.show_tradeoff()
22 print(box1.resample(21))
23 box1.inspect(21)
24 box1.inspect(21, style="graph")
25 box1.show_pairs_scatter(21)
26
27 plt.show()

```


1.9.40 sd_prim_constrained.py

```

1  """
2  a short example on how to use the constrained prim function.
3
4  for more details see Kwakkel (2019) A generalized many-objective optimization
5  approach for scenario discovery, doi: https://doi.org/10.1002/ffo2.8
6
7  """
8  import matplotlib.pyplot as plt
9  import pandas as pd
10
11  from ema_workbench.analysis import prim
12  from ema_workbench.util import ema_logging
13
14  ema_logging.log_to_stderr(ema_logging.INFO)
15
16  data = pd.read_csv("./data/bryant et al 2010 data.csv", index_col=False)
17  x = data.iloc[:, 2:11]
18  y = data.iloc[:, 15].values
19
20  box = prim.run_constrained_prim(x, y, peel_alpha=0.1)
21
22  box.show_tradeoff()
23  box.inspect(35)
24  box.inspect(35, style="graph")
25
26  plt.show()

```

1.9.41 sd_prim_flu.py

```

1  """
2
3  This file illustrated the use of the workbench for doing
4  a PRIM analysis.
5
6  The data was generated using a system dynamics models implemented in Vensim.
7  See flu_example.py for the code.
8
9
10  .. codeauthor:: jhkwakkel <j.h.kwakkel (at) tudelft (dot) nl>
11                  chamarat <c.hamarat (at) tudelft (dot) nl>
12
13  """
14  import matplotlib.pyplot as plt
15
16  import ema_workbench.analysis.prim as prim
17  from ema_workbench import ema_logging, load_results
18
19  ema_logging.log_to_stderr(level=ema_logging.INFO)
20

```

(continues on next page)

(continued from previous page)

```

21
22 def classify(data):
23     # get the output for deceased population
24     ooi = data["deceased population region 1"]
25     return ooi[:, -1] > 1000000
26
27
28 # load data
29 fn = r"./data/1000 flu cases no policy.tar.gz"
30 results = load_results(fn)
31
32 # perform prim on modified results tuple
33 prim_obj = prim.setup_prim(results, classify, threshold=0.8, threshold_type=1)
34
35 box_1 = prim_obj.find_box()
36 box_1.show_ppt()
37 box_1.show_tradeoff()
38 box_1.inspect(5, style="graph", boxlim_formatter="{: .2f}")
39 box_1.inspect(5)
40 box_1.select(5)
41 box_1.write_ppt_to_stdout()
42 box_1.show_pairs_scatter(5)
43
44 # print prim to std_out
45 print(prim_obj.stats_to_dataframe())
46 print(prim_obj.bboxes_to_dataframe())
47
48 # visualize
49 prim_obj.show_boxes()
50 plt.show()

```

1.9.42 sd_prim_wcm.py

```

1  """
2  Created on Feb 13, 2014
3
4  This example demonstrates the use of PRIM. The dataset was generated
5  using the world container model
6
7  (Tavasszy et al 2011; https://dx.doi.org/10.1016/j.jtrangeo.2011.05.005)
8
9
10 """
11 import matplotlib.pyplot as plt
12
13 from ema_workbench import ema_logging, load_results
14 from ema_workbench.analysis import prim
15
16 ema_logging.log_to_stderr(ema_logging.INFO)
17

```

(continues on next page)

(continued from previous page)

```

18 default_flow = 2.178849944502783e7
19
20
21 def classify(outcomes):
22     ooi = "throughput Rotterdam"
23     outcome = outcomes[ooi]
24     outcome = outcome / default_flow
25
26     classes = outcome < 1
27     return classes
28
29
30 fn = r"./data/5000 runs WCM.tar.gz"
31 results = load_results(fn)
32
33 prim_obj = prim.setup_prim(results, classify, mass_min=0.05, threshold=0.75)
34
35 # let's find a first box
36 box1 = prim_obj.find_box()
37
38 # let's analyze the peeling trajectory
39 box1.show_ppt()
40 box1.show_tradeoff()
41 box1.inspect_tradeoff()
42
43 box1.write_ppt_to_stdout()
44
45 # based on the peeling trajectory, we pick entry number 44
46 box1.select(44)
47
48 # show the resulting box
49 prim_obj.show_boxes()
50 prim_obj.bboxes_to_dataframe()
51
52 plt.show()

```

1.9.43 timeseries_clustering_flu.py

```

1  """
2  Created on 11 Apr 2019
3
4  @author: jhkwakkel
5  """
6  import matplotlib.pyplot as plt
7  import seaborn as sns
8
9  from ema_workbench import load_results
10 from ema_workbench.analysis import clusterer, plotting, Density
11
12 experiments, outcomes = load_results("./data/1000 flu cases no policy.tar.gz")

```

(continues on next page)

(continued from previous page)

```
13 data = outcomes["infected fraction R1"]
14
15 # calculate distances
16 distances = clusterer.calculate_cid(data)
17
18 # plot dedrog
19 clusterer.plot_dendrogram(distances)
20
21 # do agglomerative clustering on the distances
22 clusters = clusterer.apply_agglomerative_clustering(distances, n_clusters=5)
23
24 # show the clusters in the output space
25 x = experiments.copy()
26 x["clusters"] = clusters.astype("object")
27 plotting.lines(x, outcomes, group_by="clusters", density=Density.BOXPLOT)
28
29 # show the input space
30 sns.pairplot(
31     x,
32     hue="clusters",
33     vars=[
34         "infection ratio region 1",
35         "root contact rate region 1",
36         "normal contact rate region 1",
37         "recovery time region 1",
38         "permanent immune population fraction R1",
39     ],
40     plot_kws=dict(s=7),
41 )
42 plt.show()
```

1.10 Best practices

1.10.1 Separate experimentation and analysis

It is strongly recommended to cleanly separate the various steps in your exploratory modeling pipeline. So, separately execute your experiments or perform your optimization, save the results, and next analyze these results. Moreover, since parallel execution can be troublesome within the Jupyter Lab / notebook environment, I personally run my experiments and optimizations either from the command line or through an IDE using a normal python file. Jupyter Lab is then used to analyze the results.

1.10.2 Keeping things organized

A frequently recurring cause of problems when using the workbench stems from not properly organizing your files. In particular when using multiprocessing it is key that you keep things well organized. The way the workbench works with multiprocessing is that it copies the entire working directory of the model to a temporary folder for each subprocess. This temporary folder is located in the same folder as the python or notebook file from which you are running. If the working directory of your model is the same as the directory in which the run file resides, you can easily fill up your hard disk in minutes. To avoid these kinds of problems, I suggest to use a directory structure as outlined below.

```
project
├─ model_files
│   ├── a_model.nlogo
│   └─ some_input.csv
├─ results
│   ├── 100k_nfe_seed1.csv
│   └─ 1000_experiments.tar.gz
├─ figures
│   └─ pairwise_scatter.png
├─ experiments.py
├─ optimization.py
├─ analysis.ipynb
└─ model_definition.py
```

Also, if you are not familiar with absolute and relative paths, please read up on that first and only use relative paths when using the workbench. Not only will this reduce the possibility for errors, it will also mean that moving your code from one machine to another will be a lot easier.

1.11 Vensim Tips and Tricks

- *Debugging a model*

1.11.1 Debugging a model

A common occurring problem is that some of the runs of a Vensim model do not complete correctly. In the logger, we see a message stating that a run did not complete correct, with a description of the case that did not complete correctly attached to it. Typically, this error is due to a division by zero somewhere in the model during the simulation. The easiest way of finding the source of the division by zero is via Vensim itself. However, this requires that the model is parameterized as specified by the case that created the error. It is of course possible to set all the parameters by hand, however this becomes annoying on larger models, or if one has to do it multiple times. Since the Vensim DLL does not have a way to save a model, we cannot use the DLL. Instead, we can use the fact that one can save a Vensim model as a text file. By changing the required parameters in this text file via the workbench, we can then open the modified model in Vensim and spot the error.

The following script can be used for this purpose.

```
1  """
2  Created on 11 aug. 2011
```

(continues on next page)

(continued from previous page)

```

3
4 .. codeauthor:: wauping <w.auping (at) student (dot) tudelft (dot) nl>
5                jhkwakkel <j.h.kwakkel (at) tudelft (dot) nl>
6
7

```

To be able to debug the Vensim model, a few steps are needed:

1. The case that gave a bug, needs to be saved in a text file. The entire case description should be on a single line.
2. Reform and clean your model (In the Vensim menu: Model, Reform and Clean). Choose
 - * Equation Order: Alphabetical by group (not really necessary)
 - * Equation Format: Terse
3. Save your model as text (File, Save as..., Save as Type: Text Format Models)
4. Run this script
5. If the print in the end is not set([]), but set([array]), the array gives the values that where not found and changed
5. Run your new model (for example 'new text.mdl')
6. Vensim tells you about your critical mistake

```

25
26 """
27 fileSpecifyingError = ""
28
29 pathToExistingModel = "/salinization/Verziltting_aanpassingen incorrect.mdl"
30 pathToNewModel = "/models/salinization/Verziltting_aanpassingen correct.mdl"
31 newModel = open(pathToNewModel, "w")
32
33 # line = open(fileSpecifyingError).read()
34
35 line = "rainfall : 0.154705633188; adaptation time from non irrigated agriculture : 0.
    ↳ 915157119079; salt effect multiplier : 1.11965969891; adaptation time to non irrigated
    ↳ agriculture : 0.48434342934; adaptation time to irrigated agriculture : 0.330990830832;
    ↳ water shortage multiplier : 0.984356102036; delay time salt seepage : 6.0; adaptation
    ↳ time : 6.90258192256; births multiplier : 1.14344734715; diffusion lookup : [(0, 8.0),
    ↳ (10, 8.0), (20, 8.0), (30, 8.0), (40, 7.9999999999999005), (50, 4.0), (60, 9.
    ↳ 982194802803703e-14), (70, 1.2455526635140464e-27), (80, 1.5541686655435471e-41), (90,
    ↳ 1.9392517969836692e-55)]; salinity effect multiplier : 1.10500381093; technological
    ↳ developments in irrigation : 0.0117979353255; adaptation time from irrigated
    ↳ agriculture : 1.58060947607; food shortage multiplier : 0.955325345996; deaths
    ↳ multiplier : 0.875605669911; "
36
37 # we assume the case specification was copied from the logger
38 splitOne = line.split(",")
39 variable = {}
40 for n in range(len(splitOne) - 1):
41     splitTwo = splitOne[n].split(":")
42     variableElement = splitTwo[0]
43     # Delete the spaces and other rubbish on the sides of the variable name
44     variableElement = variableElement.lstrip()

```

(continues on next page)

(continued from previous page)

```

45     variableElement = variableElement.lstrip("")
46     variableElement = variableElement.rstrip()
47     variableElement = variableElement.rstrip("")
48     print(variableElement)
49     valueElement = splitTwo[1]
50     valueElement = valueElement.lstrip()
51     valueElement = valueElement.rstrip()
52     variable[variableElement] = valueElement
53     print(variable)
54
55     # This generates a new (text-formatted) model
56     changeNextLine = False
57     settedValues = []
58     for line in open(pathToExistingModel):
59         if line.find("=") != -1:
60             elements = line.split("=")
61             value = elements[0]
62             value = value.strip()
63             if value in variable:
64                 elements[1] = variable.get(value)
65                 line = elements[0] + " = " + elements[1]
66                 settedValues.append(value)
67
68     newModel.write(line)
69     newModel.close() # in order to be able to open the model in Vensim
70     notSet = set(variable.keys()) - set(settedValues)
71     print(notSet)

```

1.12 Glossary

parameter uncertainty

An uncertainty is a parameter uncertainty if the range is continuous from the lower bound to the upper bound. A parameter uncertainty can be either real valued or discrete valued.

categorical uncertainty

An uncertainty is categorical if there is not a range but a set of possibilities over which one wants to sample.

lookup uncertainty

vensim specific extension to categorical uncertainty for handling lookups in various ways

uncertainty space

the space created by the set of uncertainties

ensemble

a python class responsible for running a series of computational experiments.

model interface

a python class that provides an interface to an underlying model

working directory

a directory that contains files that a model needs

classification trees

a category of machine learning algorithms for rule induction

prim (patient rule induction method)

a rule induction algorithm

coverage

a metric developed for scenario discovery

density

a metric developed for scenario discovery

scenario discovery

a use case of EMA

case

A case specifies the input parameters for a run of a model. It is a dict instance, with the names of the uncertainties as key, and their sampled values as value.

experiment

An experiment is a complete specification for a run. It specifies the case, the name of the policy, and the name of the model.

policy

a policy is by definition an object with a name attribute. So, `policy['name']` most return the name of the policy

result

the combination of an experiment and the associated outcomes for the experiment

outcome

the data of interest produced by a model given an experiment

1.13 EMA Modules

1.13.1 Exploratory modeling framework

model

This module specifies the abstract base class for interfacing with models. Any model that is to be controlled from the workbench is controlled via an instance of an extension of this abstract base class.

class `ema_workbench.em_framework.model.AbstractModel(name)`

`ModelStructureInterface` is one of the the two main classes used for performing EMA. This is an abstract base class and cannot be used directly.

uncertainties

list of parameter instances

Type

list

levers

list of parameter instances

Type

list

outcomes

list of outcome instances

Type

list

name

alphanumerical name of model structure interface

Type

str

output

this should be a dict with the names of the outcomes as key

Type

dict

When extending this class :meth:`model_init` and

:meth:`run_model` have to be implemented.

as_dict()

returns a dict representation of the model

cleanup()

This model is called after finishing all the experiments, but just prior to returning the results. This method gives a hook for doing any cleanup, such as closing applications.

In case of running in parallel, this method is called during the cleanup of the pool, just prior to removing the temporary directories.

initialized(*policy*)

check if model has been initialized

Parameters

policy (a *Policy* instance) –

model_init(*policy*)

Method called to initialize the model.

Parameters

policy (*dict*) – policy to be run.

Note: This method should always be implemented. Although in simple cases, a simple pass can suffice.

reset_model()

Method for resetting the model to its initial state. The default implementation only sets the outputs to an empty dict.

retrieve_output()

Method for retrieving output after a model run.

Return type

dict with the results of a model run.

run_model(*scenario*, *policy*)

Method for running an instantiated model structure.

Parameters

- **scenario** (*Scenario instance*) –
- **policy** (*Policy instance*) –

class ema_workbench.em_framework.model.**FileModel**(*name*, *wd=None*, *model_file=None*)

as_dict()

returns a dict representation of the model

class ema_workbench.em_framework.model.**Model**(*name*, *function=None*)

class ema_workbench.em_framework.model.**Replicator**(*name*)

run_model(*scenario*, *policy*)

Method for running an instantiated model structure.

Parameters

- **scenario** (*Scenario instance*) –
- **policy** (*Policy instance*) –

class ema_workbench.em_framework.model.**ReplicatorModel**(*name*, *function=None*)

class ema_workbench.em_framework.model.**SingleReplication**(*name*)

run_model(*scenario*, *policy*)

Method for running an instantiated model structure.

Parameters

- **scenario** (*Scenario instance*) –
- **policy** (*Policy instance*) –

parameters

parameters and related helper classes and functions

class ema_workbench.em_framework.parameters.**BooleanParameter**(*name*, *default=None*,
variable_name=None, *pff=False*)

boolean model input parameter

A BooleanParameter is similar to a CategoricalParameter, except the category values can only be True or False.

Parameters

- **name** (*str*) –
- **variable_name** (*str*, or *list of str*) –

class ema_workbench.em_framework.parameters.**CategoricalParameter**(*name*, *categories*,
default=None,
variable_name=None,
pff=False, *multivalue=False*)

categorical model input parameter

Parameters

- **name** (*str*) –
- **categories** (*collection of obj*) –
- **variable_name** (*str, or list of str*) –
- **multivalue** (*boolean*) – if categories have a set of values, for each variable_name a different one.
- **TODO** (*#*) –
- **TODO** –

cat_for_index(*index*)

return category given index

Parameters**index** (*int*) –**Return type**

object

from_dist(*name, dist*)

alternative constructor for creating a parameter from a frozen scipy.stats distribution directly

Parameters

- **dist** (*scipy stats frozen dist*) –
- ****kwargs** (*valid keyword arguments for Parameter instance*) –

index_for_cat(*category*)

return index of category

Parameters**category** (*object*) –**Return type**

int

class ema_workbench.em_framework.parameters.**Category**(*name, value*)**class** ema_workbench.em_framework.parameters.**Constant**(*name, value*)

Constant class,

can be used for any parameter that has to be set to a fixed value

class ema_workbench.em_framework.parameters.**IntegerParameter**(*name, lower_bound, upper_bound, resolution=None, default=None, variable_name=None, pff=False*)

integer valued model input parameter

Parameters

- **name** (*str*) –
- **lower_bound** (*int*) –
- **upper_bound** (*int*) –
- **resolution** (*iterable*) –
- **variable_name** (*str, or list of str*) –

Raises

- **ValueError** – if lower bound is larger than upper bound
- **ValueError** – if entries in resolution are outside range of lower_bound and upper_bound, or not an integer instance
- **ValueError** – if lower_bound or upper_bound is not an integer instance

classmethod `from_dist(name, dist, **kwargs)`

alternative constructor for creating a parameter from a frozen scipy.stats distribution directly

Parameters

- `dist` (*scipy stats frozen dist*) –
- `**kwargs` (*valid keyword arguments for Parameter instance*) –

class `ema_workbench.em_framework.parameters.Parameter(name, lower_bound, upper_bound, resolution=None, default=None, variable_name=None, pff=False)`

Base class for any model input parameter

Parameters

- `name` (*str*) –
- `lower_bound` (*int or float*) –
- `upper_bound` (*int or float*) –
- `resolution` (*collection*) –
- `pff` (*bool*) – if true, sample over this parameter using resolution in case of partial factorial sampling

Raises

- **ValueError** – if lower bound is larger than upper bound
- **ValueError** – if entries in resolution are outside range of lower_bound and upper_bound

classmethod `from_dist(name, dist, **kwargs)`

alternative constructor for creating a parameter from a frozen scipy.stats distribution directly

Parameters

- `dist` (*scipy stats frozen dist*) –
- `**kwargs` (*valid keyword arguments for Parameter instance*) –

class `ema_workbench.em_framework.parameters.RealParameter(name, lower_bound, upper_bound, resolution=None, default=None, variable_name=None, pff=False)`

real valued model input parameter

Parameters

- `name` (*str*) –
- `lower_bound` (*int or float*) –
- `upper_bound` (*int or float*) –
- `resolution` (*iterable*) –
- `variable_name` (*str, or list of str*) –

Raises

- **ValueError** – if lower bound is larger than upper bound
- **ValueError** – if entries in resolution are outside range of lower_bound and upper_bound

classmethod `from_dist(name, dist, **kwargs)`

alternative constructor for creating a parameter from a frozen scipy.stats distribution directly

Parameters

- `dist` (*scipy stats frozen dist*) –
- `**kwargs` (*valid keyword arguments for Parameter instance*) –

`ema_workbench.em_framework.parameters.parameters_from_csv(uncertainties, **kwargs)`

Helper function for creating many Parameters based on a DataFrame or csv file

Parameters

- `uncertainties` (*str, DataFrame*) –
- `**kwargs` (*dict, arguments to pass to pandas.read_csv*) –

Return type

list of Parameter instances

This helper function creates uncertainties. It assumes that the DataFrame or csv file has a column titled 'name', optionally a type column {int, real, cat}, can be included as well. the remainder of the columns are handled as values for the parameters. If type is not specified, the function will try to infer type from the values.

Note that this function does not support the resolution and default kwargs on parameters.

An example of a csv:

```
NAME,TYPE,, a_real,real,0,1.1, an_int,int,1,9, a_categorical,cat,a,b,c
```

this CSV file would result in

```
[RealParameter('a_real', 0, 1.1, resolution=[], default=None),
 IntegerParameter('an_int', 1, 9, resolution=[], default=None), CategoricalParameter('a_categorical', ['a',
 'b', 'c'], default=None)]
```

`ema_workbench.em_framework.parameters.parameters_to_csv(parameters, file_name)`

Helper function for writing a collection of parameters to a csv file

Parameters

- `parameters` (*collection of Parameter instances*) –
- `file_name` (*str*) –

The function iterates over the collection and turns these into a data frame prior to storing them. The resulting csv can be loaded using the `parameters_from_csv` function. Note that currently we don't store resolution and default attributes.

outcomes

Module for outcome classes

```
class ema_workbench.em_framework.outcomes.AbstractOutcome(name, kind=0, variable_name=None,
                                                           function=None, expected_range=None,
                                                           shape=None, dtype=None)
```

Base Outcome class

Parameters

- **name** (*str*) – Name of the outcome.
- **kind** (*{INFO, MINIMIZE, MAXIMIZE}, optional*) –
- **variable_name** (*str, optional*) – if the name of the outcome in the underlying model is different from the name of the outcome, you can supply the variable name as an optional argument, if not provided, defaults to name
- **function** (*callable, optional*) – a callable to perform postprocessing on data retrieved from model
- **expected_range** (*2 tuple, optional*) – expected min and max value for outcome, used by HyperVolume convergence metric
- **shape** (*{tuple, None} optional*) – must be used in conjunction with dtype. Enables pre-allocation of data structure for storing results.
- **dtype** (*datatype, optional*) – must be used in conjunction with shape. Enables pre-allocation of data structure for storing results.

name

Type

str

kind

Type

int

variable_name

Type

str

function

Type

callable

shape

Type

tuple

dtype

Type

datatype

abstract classmethod from_disk(*filename*, *archive*)

helper function for loading from disk

Parameters

- **filename** (*str*) –
- **archive** (*Tarfile*) –

abstract classmethod to_disk(*values*)

helper function for writing outcome to disk

Parameters

values (*obj*) – data to store

Return type

BytesIO

```
class ema_workbench.em_framework.outcomes.ArrayOutcome(name, variable_name=None,
                                                         function=None, expected_range=None,
                                                         shape=None, dtype=None)
```

Array Outcome class for n-dimensional arrays

Parameters

- **name** (*str*) – Name of the outcome.
- **variable_name** (*str*, *optional*) – if the name of the outcome in the underlying model is different from the name of the outcome, you can supply the variable name as an optional argument, if not provided, defaults to name
- **function** (*callable*, *optional*) – a callable to perform postprocessing on data retrieved from model
- **expected_range** (*2 tuple*, *optional*) – expected min and max value for outcome, used by HyperVolume convergence metric
- **shape** (*{tuple, None}*, *optional*) – must be used in conjunction with dtype. Enables pre-allocation of data structure for storing results.
- **dtype** (*datatype*, *optional*) – must be used in conjunction with shape. Enables pre-allocation of data structure for storing results.

name

Type

str

kind

Type

int

variable_name

Type

str

function

Type

callable

shape

Type
tuple

expected_range

Type
tuple

dtype

Type
datatype

classmethod from_disk(*filename*, *archive*)

helper function for loading from disk

Parameters

- **filename** (*str*) –
- **archive** (*Tarfile*) –

classmethod to_disk(*values*)

helper function for writing outcome to disk

Parameters

values (*ND array*) –

Returns

- *BytesIO*
- *filename*

class `ema_workbench.em_framework.outcomes.Constraint`(*name*, *parameter_names=None*,
outcome_names=None, *function=None*)

Constraints class that can be used when defining constrained optimization problems.

Parameters

- **name** (*str*) –
- **parameter_names** (*str or collection of str*) –
- **outcome_names** (*str or collection of str*) –
- **function** (*callable*) –

name

Type
str

parameter_names

name(s) of the uncertain parameter(s) and/or lever parameter(s) to which the constraint applies

Type
str, list of *str*

outcome_names

name(s) of the outcome(s) to which the constraint applies

Type

str, list of str

function

The function should return the distance from the feasibility threshold, given the model outputs with a variable name. The distance should be 0 if the constraint is met.

Type

callable

```
class ema_workbench.em_framework.outcomes.ScalarOutcome(name, kind=0, variable_name=None,
                                                         function=None, expected_range=None,
                                                         dtype=None)
```

Scalar Outcome class

Parameters

- **name** (*str*) – Name of the outcome.
- **kind** (*{INFO, MINIMIZE, MAXIMIZE}, optional*) –
- **variable_name** (*str, optional*) – if the name of the outcome in the underlying model is different from the name of the outcome, you can supply the variable name as an optional argument, if not provided, defaults to name
- **function** (*callable, optional*) – a callable to perform post processing on data retrieved from model
- **expected_range** (*collection, optional*) – expected min and max value for outcome, used by HyperVolume convergence metric
- **dtype** (*datatype, optional*) – Enables pre-allocation of data structure for storing results.

name**Type**

str

kind**Type**

int

variable_name**Type**

str

function**Type**

callable

shape**Type**

tuple

expected_range

Type
tuple

dtype

Type
datatype

classmethod from_disk(*filename*, *archive*)

helper function for loading from disk

Parameters

- **filename** (*str*) –
- **archive** (*Tarfile*) –

classmethod to_disk(*values*)

helper function for writing outcome to disk

Parameters

values (*1D array*) –

Returns

- *BytesIO*
- *filename*

```
class ema_workbench.em_framework.outcomes.TimeSeriesOutcome(name, variable_name=None,  
                                                             function=None,  
                                                             expected_range=None, shape=None,  
                                                             dtype=None)
```

TimeSeries Outcome class for 1D arrays

Parameters

- **name** (*str*) – Name of the outcome.
- **variable_name** (*str*, *optional*) – if the name of the outcome in the underlying model is different from the name of the outcome, you can supply the variable name as an optional argument, if not provided, defaults to name
- **function** (*callable*, *optional*) – a callable to perform postprocessing on data retrieved from model
- **expected_range** (*2 tuple*, *optional*) – expected min and max value for outcome, used by HyperVolume convergence metric
- **shape** (*{tuple, None}* *optional*) – must be used in conjunction with dtype. Enables pre-allocation of data structure for storing results.
- **dtype** (*datatype*, *optional*) – must be used in conjunction with shape. Enables pre-allocation of data structure for storing results.

name

Type
str

kind

Type
int

variable_name

Type
str

function

Type
callable

shape

Type
tuple

expected_range

Type
tuple

dtype

Type
datatype

Notes

Time series outcomes are currently assumed to be 1D arrays. If you are dealing with higher dimensional outputs (e.g., multiple replications resulting in 2D arrays), use `ArrayOutcome` instead.

classmethod from_disk(*filename*, *archive*)

helper function for loading from disk

Parameters

- **filename** (*str*) –
- **archive** (*Tarfile*) –

classmethod to_disk(*values*)

helper function for writing outcome to disk

Parameters

values (*DataFrame*) –

Returns

- *StringIO*
- *filename*

evaluators

collection of evaluators for performing experiments, optimization, and robust optimization

class ema_workbench.em_framework.evaluators.**IpyparallelEvaluator**(*msis, client, **kwargs*)

evaluator for using an ipyparallel pool

evaluate_experiments(*scenarios, policies, callback, combine='factorial'*)

used by ema_workbench

finalize()

finalize the evaluator

initialize()

initialize the evaluator

class ema_workbench.em_framework.evaluators.**MultiprocessingEvaluator**(*msis, n_processes=None, maxtasksperchild=None, **kwargs*)

evaluator for experiments using a multiprocessing pool

Parameters

- **msis** (*collection of models*) –
- **n_processes** (*int (optional)*) – A negative number can be inputted to use the number of logical cores minus the negative cores. For example, on a 12 thread processor, -2 results in using 10 threads.
- **max_tasks** (*int (optional)*) –

note that the maximum number of available processes is either multiprocessing.cpu_count() and in case of windows, this never can be higher then 61

evaluate_experiments(*scenarios, policies, callback, combine='factorial'*)

used by ema_workbench

finalize()

finalize the evaluator

initialize()

initialize the evaluator

class ema_workbench.em_framework.evaluators.**Samplers**(*value*)

Enum for different kinds of samplers

class ema_workbench.em_framework.evaluators.**SequentialEvaluator**(*msis*)

evaluate_experiments(*scenarios, policies, callback, combine='factorial'*)

used by ema_workbench

finalize()

finalize the evaluator

initialize()

initialize the evaluator

```
ema_workbench.em_framework.evaluators.optimize(models, algorithm=<class
    'platypus.algorithms.EpsNSGAI'>, nfe=10000,
    searchover='levers', evaluator=None, reference=None,
    convergence=None, constraints=None,
    convergence_freq=1000, logging_freq=5,
    variator=None, **kwargs)
```

optimize the model

Parameters

- **models** (1 or more *Model* instances) –
- **algorithm** (a valid Platypus optimization algorithm) –
- **nfe** (*int*) –
- **searchover** ({'uncertainties', 'levers'}) –
- **evaluator** (*evaluator instance*) –
- **reference** (*Policy or Scenario instance, optional*) – overwrite the default scenario in case of searching over levers, or default policy in case of searching over uncertainties
- **convergence** (*function or collection of functions, optional*) –
- **constraints** (*list, optional*) –
- **convergence_freq** (*int*) – nfe between convergence check
- **logging_freq** (*int*) – number of generations between logging of progress
- **variator** (*platypus GAOperator instance, optional*) – if None, it falls back on the defaults in platypus-opts which is SBX with PM
- **kwargs** (*any additional arguments will be passed on to algorithm*) –

Return type

pandas DataFrame

Raises

- **EMAError** if searchover is not one of 'uncertainties' or 'levers' –
- **NotImplementedError** if len(models) > 1 –

```
ema_workbench.em_framework.evaluators.perform_experiments(models, scenarios=0, policies=0,
    evaluator=None,
    reporting_interval=None,
    reporting_frequency=10,
    uncertainty_union=False,
    lever_union=False,
    outcome_union=False,
    uncertainty_sampling=Samplers.LHS,
    lever_sampling=Samplers.LHS,
    callback=None, return_callback=False,
    combine='factorial',
    log_progress=False)
```

sample uncertainties and levers, and perform the resulting experiments on each of the models

Parameters

- **models** (*one or more AbstractModel instances*) –
- **scenarios** (*int or collection of Scenario instances, optional*) –

- **policies** (*int or collection of Policy instances, optional*) –
- **evaluator** (*Evaluator instance, optional*) –
- **reporting_interval** (*int, optional*) –
- **reporting_frequency** (*int, optional*) –
- **uncertainty_union** (*boolean, optional*) –
- **lever_union** (*boolean, optional*) –
- **outcome_union** (*boolean, optional*) –
- **uncertainty_sampling** (*{LHS, MC, FF, PFF, SOBOL, MORRIS, FAST}, optional*) –
- **lever_sampling** (*{LHS, MC, FF, PFF, SOBOL, MORRIS, FAST}, optional* *TODO:: update doc*) –
- **callback** (*Callback instance, optional*) –
- **return_callback** (*boolean, optional*) –
- **log_progress** (*bool, optional*) –
- **combine** (*{'factorial', 'zipover'}, optional*) – how to combine uncertainties and levers? In case of ‘factorial’, both are sampled separately using their respective samplers. Next the resulting designs are combined in a full factorial manner. In case of ‘zipover’, both are sampled separately and then combined by cycling over the shortest of the the two sets of designs until the longest set of designs is exhausted.

Returns

the experiments as a dataframe, and a dict with the name of an outcome as key, and the associated values as numpy array. Experiments and outcomes are aligned on index.

Return type

tuple

optimization

```
class ema_workbench.em_framework.optimization.ArchiveLogger(directory, decision_varnames,  
                                                            outcome_varnames,  
                                                            base_filename='archives.tar.gz')
```

Helper class to write the archive to disk at each iteration

Parameters

- **directory** (*str*) –
- **decision_varnames** (*list of str*) –
- **outcome_varnames** (*list of str*) –
- **base_filename** (*str, optional*) –

```
classmethod load_archives(filename)
```

load the archives stored with the ArchiveLogger

Parameters

filename (*str*) – relative path to file

Return type

dict with nfe as key and dataframe as vlaue

```
class ema_workbench.em_framework.optimization.Convergence(metrics, max_nfe,
                                                         convergence_freq=1000,
                                                         logging_freq=5, log_progress=False)
```

helper class for tracking convergence of optimization

```
class ema_workbench.em_framework.optimization.EpsilonIndicatorMetric(reference_set, problem,
                                                                    **kwargs)
```

EpsilonIndicator metric

Parameters

- **reference_set** (*DataFrame*) –
- **problem** (*PlatypusProblem* instance) –

this is a thin wrapper around EpsilonIndicator as provided by platypus to make it easier to use in conjunction with the workbench.

```
class ema_workbench.em_framework.optimization.EpsilonProgress
```

epsilon progress convergence metric class

```
class ema_workbench.em_framework.optimization.GenerationalDistanceMetric(reference_set,
                                                                           problem, **kwargs)
```

GenerationalDistance metric

Parameters

- **reference_set** (*DataFrame*) –
- **problem** (*PlatypusProblem* instance) –
- **d** (*int*, *default=1*) – the power in the intergenerational distance function

This is a thin wrapper around GenerationalDistance as provided by platypus to make it easier to use in conjunction with the workbench.

see https://link.springer.com/content/pdf/10.1007/978-3-319-15892-1_8.pdf for more information

```
class ema_workbench.em_framework.optimization.HypervolumeMetric(reference_set, problem,
                                                                **kwargs)
```

Hypervolume metric

Parameters

- **reference_set** (*DataFrame*) –
- **problem** (*PlatypusProblem* instance) –

this is a thin wrapper around Hypervolume as provided by platypus to make it easier to use in conjunction with the workbench.

```
class ema_workbench.em_framework.optimization.InvertedGenerationalDistanceMetric(reference_set,
                                                                                   problem,
                                                                                   **kwargs)
```

InvertedGenerationalDistance metric

Parameters

- **reference_set** (*DataFrame*) –
- **problem** (*PlatypusProblem* instance) –
- **d** (*int*, *default=1*) – the power in the inverted intergenerational distance function

This is a thin wrapper around `InvertedGenerationalDistance` as provided by platypus to make it easier to use in conjunction with the workbench.

see https://link.springer.com/content/pdf/10.1007/978-3-319-15892-1_8.pdf for more information

class `ema_workbench.em_framework.optimization.OperatorProbabilities(name, index)`

OperatorProbabiliy convergence tracker for use with auto adaptive operator selection.

Parameters

- **name** (*str*) –
- **index** (*int*) –

State of the art MOEAs like Borg (and `GenerationalBorg` provided by the workbench) use autoadaptive operator selection. The algorithm has multiple different evolutionary operators. Over the run, it tracks how well each operator is doing in producing fitter offspring. The probability of the algorithm using a given evolutionary operator is proportional to how well this operator has been doing in producing fitter offspring in recent generations. This class can be used to track these probabilities over the run of the algorithm.

class `ema_workbench.em_framework.optimization.Problem(searchover, parameters, outcome_names, constraints, reference=None)`

small extension to Platypus problem object, includes information on the names of the decision variables, the names of the outcomes, and the type of search

class `ema_workbench.em_framework.optimization.RobustProblem(parameters, outcome_names, scenarios, robustness_functions, constraints)`

small extension to Problem object for robust optimization, adds the scenarios and the robustness functions

class `ema_workbench.em_framework.optimization.SpacingMetric(problem)`

Spacing metric

Parameters

- **problem** (*PlatypusProblem instance*) –

this is a thin wrapper around Spacing as provided by platypus to make it easier to use in conjunction with the workbench.

`ema_workbench.em_framework.optimization.epsilon_nondominated(results, epsilons, problem)`

Merge the list of results into a single set of non dominated results using the provided epsilon values

Parameters

- **results** (*list of DataFrames*) –
- **epsilons** (*epsilon values for each objective*) –
- **problem** (*PlatypusProblem instance*) –

Return type

DataFrame

Notes

this is a platypus based alternative to pareto.py (<https://github.com/matthewjwoodruff/pareto.py>)

`ema_workbench.em_framework.optimization.rebuild_platypus_population(archive, problem)`

rebuild a population of platypus Solution instances

Parameters

- **archive** (*DataFrame*) –
- **problem** (*PlatypusProblem* instance) –

Return type

list of platypus Solutions

`ema_workbench.em_framework.optimization.to_problem(model, searchover, reference=None, constraints=None)`

helper function to create Problem object

Parameters

- **model** (*AbstractModel* instance) –
- **searchover** (*str*) –
- **reference** (*Policy* or *Scenario* instance, optional) – overwrite the default scenario in case of searching over levers, or default policy in case of searching over uncertainties
- **constraints** (*list*, optional) –

Return type

Problem instance

`ema_workbench.em_framework.optimization.to_robust_problem(model, scenarios, robustness_functions, constraints=None)`

helper function to create RobustProblem object

Parameters

- **model** (*AbstractModel* instance) –
- **scenarios** (*collection*) –
- **robustness_functions** (*iterable of ScalarOutcomes*) –
- **constraints** (*list*, optional) –

Return type

RobustProblem instance

outputspace_exploration

Provides a genetic algorithm based on novelty search for output space exploration.

The algorithm is inspired by Chérel et al (2015). In short, from Chérel et al, we have taken the idea of the HitBox. Basically, this is an epsilon archive where one keeps track of how many solutions have fallen into each grid cell. Next, tournament selection based on novelty is used as the selective pressure. Novelty is defined as 1/nr. of solutions in same grid cell. This is then combined with auto-adaptive population sizing as used in e-NSGAIL. This replaces the use of adaptive Cauchy mutation as used by Chérel et al. There is also an more sophisticated algorithm that adds auto-adaptive operator selection as used in BORG.

The algorithm can be used in combination with the optimization functionality of the workbench. Just pass an `OutputSpaceExploration` instance as algorithm to optimize.

```
class ema_workbench.em_framework.outputspace_exploration.AutoAdaptiveOutputSpaceExploration(problem,  
                                                                                          grid_spec=None,  
                                                                                          popu-  
                                                                                          u-  
                                                                                          la-  
                                                                                          tion_size=100,  
                                                                                          gen-  
                                                                                          er-  
                                                                                          a-  
                                                                                          tor=<platypus  
                                                                                          ob-  
                                                                                          ject>,  
                                                                                          vari-  
                                                                                          a-  
                                                                                          tor=None,  
                                                                                          **kwargs)
```

A combination of auto-adaptive operator selection with `OutputSpaceExploration`.

The parametrization of all operators is based on the default values as used in Borg 1.9.

Parameters

- **problem** (*a platypus Problem instance*) –
- **grid_spec** (*list of tuples*) – with min, max, and epsilon for each outcome of interest
- **population_size** (*int, optional*) –

Notes

Limited to `RealParameters` only.

```
class ema_workbench.em_framework.outputspace_exploration.OutputSpaceExploration(problem,  
                                                                                   grid_spec=None,  
                                                                                   popula-  
                                                                                   tion_size=100,  
                                                                                   genera-  
                                                                                   tor=<platypus.operators.Ran-  
                                                                                   object>,  
                                                                                    varia-  
                                                                                   tor=None,  
                                                                                   **kwargs)
```

Basic genetic algorithm for output space exploration using novelty search.

Parameters

- **problem** (*a platypus Problem instance*) –
- **grid_spec** (*list of tuples*) – with min, max, and epsilon for each outcome of interest
- **population_size** (*int, optional*) –

The algorithm defines novelty using an epsilon-like grid in the output space. Novelty is one divided by the number of seen solutions in a given grid cell. Tournament selection using novelty is used to create offspring. Crossover is done using simulated binary crossover and mutation is done using polynomial mutation.

The epsilon like grid structure for tracking novelty is implemented using an archive, the Hit Box. Per epsilon grid cell, a single solution closes to the centre of the cell is maintained. This makes the algorithm behave virtually identical to -NSGAI. The archive is returned as results and epsilon progress is defined.

To deal with a stalled search, adaptive time continuation, identical to -NSGAI is used.

Notes

Output space exploration relies on the optimization functionality of the workbench. Therefore, outcomes of kind INFO are ignored. For output space exploration the direction (i.e. minimize or maximize) does not matter.

samplers

This module contains various classes that can be used for specifying different types of samplers. These different samplers implement basic sampling techniques including Full Factorial sampling, Latin Hypercube sampling, and Monte Carlo sampling.

class `ema_workbench.em_framework.samplers.AbstractSampler`

Abstract base class from which different samplers can be derived.

In the simplest cases, only the sample method needs to be overwritten. `generate_designs`` is the only method called from outside. The other methods are used internally to generate the designs.

generate_designs(*parameters*, *nr_samples*)

external interface for Sampler. Returns the computational experiments over the specified parameters, for the given number of samples for each parameter.

Parameters

- **parameters** (*list*) – a list of parameters for which to generate the experimental designs
- **nr_samples** (*int*) – the number of samples to draw for each parameter

Returns

- *generator* – a generator object that yields the designs resulting from combining the parameters
- *int* – the number of experimental designs

generate_samples(*parameters*, *size*)

The main method of :class: `~sampler.Sampler` and its children. This will call the sample method for each of the parameters and return the resulting designs.

Parameters

- **parameters** (*collection*) – a collection of `RealParameter`, `IntegerParameter`, and `CategoricalParameter` instances.
- **size** (*int*) – the number of samples to generate.

Returns

dict with the parameter.name as key, and the sample as value

Return type

dict

sample(*distribution*, *size*)

method for sampling a number of samples from a particular distribution. The various samplers differ with respect to their implementation of this method.

Parameters

- **distribution** (*scipy frozen distribution*) –
- **size** (*int*) – the number of samples to generate

Returns

the samples for the distribution and specified parameters

Return type

numpy array

class `ema_workbench.em_framework.samplers.DefaultDesigns`(*designs, parameters, n*)

iterable for the experimental designs

class `ema_workbench.em_framework.samplers.FullFactorialSampler`

generates a full factorial sample.

If the parameter is non categorical, the resolution is set the number of samples. If the parameter is categorical, the specified value for samples will be ignored and each category will be used instead.

determine_nr_of_designs(*sampled_parameters*)

Helper function for determining the number of experiments that will be generated given the sampled parameters.

Parameters

sampled_parameters (*list*) – a list of sampled parameters, as the values return by `generate_samples`

Returns

the total number of experimental design

Return type

int

generate_designs(*parameters, nr_samples*)

This method provides an alternative implementation to the default implementation provided by `Sampler`. This version returns a full factorial design across the parameters.

Parameters

- **parameters** (*list*) – a list of parameters for which to generate the experimental designs
- **nr_samples** (*int*) – the number of intervals to use on each Parameter. Categorical parameters always return all their categories

Returns

- *generator* – a generator object that yields the designs resulting from combining the parameters
- *int* – the number of experimental designs

generate_samples(*parameters, size*)

The main method of :class: `~sampler.Sampler` and its children. This will call the `sample` method for each of the parameters and return the resulting samples

Parameters

- **parameters** (*collection*) – a collection of `Parameter` instances
- **size** (*int*) – the number of samples to generate.

Returns

with the paramertainty.name as key, and the sample as value

Return type

dict

class ema_workbench.em_framework.samplers.LHSSampler

generates a Latin Hypercube sample for each of the parameters

sample(*distribution, size*)

generate a Latin Hypercube Sample.

Parameters

- **distribution** (*scipy frozen distribution*) –
- **size** (*int*) – the number of samples to generate

Returns

with the paramertainty.name as key, and the sample as value

Return type

dict

class ema_workbench.em_framework.samplers.MonteCarloSampler

generates a Monte Carlo sample for each of the parameters.

sample(*distribution, size*)

generate a Monte Carlo Sample.

Parameters

- **distribution** (*scipy frozen distribution*) –
- **size** (*int*) – the number of samples to generate

Returns

with the paramertainty.name as key, and the sample as value

Return type

dict

class ema_workbench.em_framework.samplers.UniformLHSSampler

generate_samples(*parameters, size*)

Parameters

- **parameters** (*collection*) –
- **size** (*int*) –

Returns

dict with the parameter.name as key, and the sample as value

Return type

dict

ema_workbench.em_framework.samplers.**determine_parameters**(*models, attribute, union=True*)

determine the parameters over which to sample

Parameters

- **models** (*a collection of AbstractModel instances*) –

- **attribute** (*{'uncertainties', 'levers'}*) –
- **union** (*bool, optional*) – in case of multiple models, sample over the union of levers, or over the intersection of the levers

Return type

collection of Parameter instances

```
ema_workbench.em_framework.samplers.sample_levers(models, n_samples, union=True, sampler=<ema_workbench.em_framework.samplers.LHSSampler object>)
```

generate policies by sampling over the levers

Parameters

- **models** (*a collection of AbstractModel instances*) –
- **n_samples** (*int*) –
- **union** (*bool, optional*) – in case of multiple models, sample over the union of levers, or over the intersection of the levers
- **sampler** (*Sampler instance, optional*) –

Return type

generator yielding Policy instances

```
ema_workbench.em_framework.samplers.sample_parameters(parameters, n_samples, sampler=<ema_workbench.em_framework.samplers.LHSSampler object>, kind=<class 'ema_workbench.em_framework.points.Point'>)
```

generate cases by sampling over the parameters

Parameters

- **parameters** (*collection of AbstractParameter instances*) –
- **n_samples** (*int*) –
- **sampler** (*Sampler instance, optional*) –
- **kind** (*{Case, Scenario, Policy}, optional*) – the class into which the samples are collected

Return type

generator yielding Case, *Scenario*, or Policy instances

```
ema_workbench.em_framework.samplers.sample_uncertainties(models, n_samples, union=True, sampler=<ema_workbench.em_framework.samplers.LHSSampler object>)
```

generate scenarios by sampling over the uncertainties

Parameters

- **models** (*a collection of AbstractModel instances*) –
- **n_samples** (*int*) –
- **union** (*bool, optional*) – in case of multiple models, sample over the union of uncertainties, or over the intersection of the uncertainties
- **sampler** (*Sampler instance, optional*) –

Return type

generator yielding Scenario instances

salib_samplers

Samplers for working with SALib

class ema_workbench.em_framework.salib_samplers.FASTSampler(*m=4*)

Sampler generating a Fourier Amplitude Sensitivity Test (FAST) using SALib

Parameters

m (*int* (*default: 4*)) – The interference parameter, i.e., the number of harmonics to sum in the Fourier series decomposition

class ema_workbench.em_framework.salib_samplers.MorrisSampler(*num_levels=4*,
optimal_trajectories=None,
local_optimization=True)

Sampler generating a morris design using SALib

Parameters

- **num_levels** (*int*) – The number of grid levels
- **grid_jump** (*int*) – The grid jump size
- **optimal_trajectories** (*int*, *optional*) – The number of optimal trajectories to sample (between 2 and N)
- **local_optimization** (*bool*, *optional*) – Flag whether to use local optimization according to Ruano et al. (2012) Speeds up the process tremendously for bigger N and num_levels. Setting this variable to be true causes the function to ignore gurobi.

class ema_workbench.em_framework.salib_samplers.SobolSampler(*second_order=True*)

Sampler generating a Sobol design using SALib

Parameters

second_order (*bool*, *optional*) – indicates whether second order effects should be included

ema_workbench.em_framework.salib_samplers.get_SALib_problem(*uncertainties*)

returns a dict with a problem specification as required by SALib

experiment_runner

helper module for running experiments and keeping track of which model has been initialized with which policy.

class ema_workbench.em_framework.experiment_runner.ExperimentRunner(*msis*)

Helper class for running the experiments

This class contains the logic for initializing models properly, running the experiment, getting the results, and cleaning up afterwards.

Parameters

- **msis** (*dict*) –
- **model_kwargs** (*dict*) –

msi_initialization

keeps track of which model is initialized with which policy.

Type

dict

msis

models indexed by name

Type

dict

model_kwargs

keyword arguments for model_init

Type

dict

run_experiment(*experiment*)

The logic for running a single experiment. This code makes sure that model(s) are initialized correctly.

Parameters

experiment (*Case instance*) –

Returns

- **experiment_id** (*int*)
- **case** (*dict*)
- **policy** (*str*)
- **model_name** (*str*)
- **result** (*dict*)

Raises

- **EMAError** – if the model instance raises an EMA error, these are reraised.
- **Exception** – Catch all for all other exceptions being raised by the model. These are reraised.

callbacks

This module provides an abstract base class for a callback and a default implementation.

If you want to store the data in a way that is different from the functionality provided by the default callback, you can write your own extension of callback. For example, you can easily implement a callback that stores the data in e.g. a NoSQL file.

The only method to implement is the `__call__` magic method. To use logging of progress, always call `super`.

```
class ema_workbench.em_framework.callbacks.AbstractCallback(uncertainties, levers, outcomes,  
                                                         nr_experiments,  
                                                         reporting_interval=None,  
                                                         reporting_frequency=10,  
                                                         log_progress=False)
```

Abstract base class from which different call back classes can be derived. Callback is responsible for storing the results of the runs.

Parameters

- **uncertainties** (*list*) – list of uncertain parameters
- **levers** (*list*) – list of lever parameters
- **outcomes** (*list*) – a list of outcomes

- **nr_experiments** (*int*) – the total number of experiments to be executed
- **reporting_interval** (*int*, *optional*) – the interval between progress logs
- **reporting_frequency** (*int*, *optional*) – the total number of progress logs
- **log_progress** (*bool*, *optional*) – if true, progress is logged, if false, use tqdm progress bar.

i

a counter that keeps track of how many experiments have been saved

Type
int

nr_experiments

Type
int

outcomes

Type
list

parameters

combined list of uncertain parameters and lever parameters

Type
list

reporting_interval

the interval between progress logs

Type
int,

abstract get_results()

method for retrieving the results. Called after all experiments have been completed. Any extension of AbstractCallback needs to implement this method.

```
class ema_workbench.em_framework.callbacks.DefaultCallback(uncertainties, levers, outcomes,  
                                                         nr_experiments,  
                                                         reporting_interval=100,  
                                                         reporting_frequency=10,  
                                                         log_progress=False)
```

Default callback class

Parameters

- **uncertainties** (*list*) – list of uncertain parameters
- **levers** (*list*) – list of lever parameters
- **outcomes** (*list*) – a list of outcomes
- **nr_experiments** (*int*) – the total number of experiments to be executed
- **reporting_interval** (*int*, *optional*) – the interval between progress logs
- **reporting_frequency** (*int*, *optional*) – the total number of progress logs
- **log_progress** (*bool*, *optional*) – if true, progress is logged, if false, use tqdm progress bar.

Callback can be used in `perform_experiments` as a means for specifying the way in which the results should be handled. If no callback is specified, this default implementation is used. This one can be overwritten or replaced with a callback of your own design. For example if you prefer to store the result in a database or write them to a text file.

get_results()

method for retrieving the results. Called after all experiments have been completed. Any extension of `AbstractCallback` needs to implement this method.

```
class ema_workbench.em_framework.callbacks.FileBasedCallback(uncertainties, levers, outcomes,  
                                                            nr_experiments,  
                                                            reporting_interval=100,  
                                                            reporting_frequency=10)
```

Callback that stores data in csv files while running th model

Parameters

- **uncertainties** (*list*) – list of uncertain parameters
- **levers** (*list*) – list of lever parameters
- **outcomes** (*list*) – a list of outcomes
- **nr_experiments** (*int*) – the total number of experiments to be executed
- **reporting_interval** (*int, optional*) – the interval between progress logs
- **reporting_frequency** (*int, optional*) – the total number of progress logs
- **log_progress** (*bool, optional*) – if true, progress is logged, if false, use tqdm progress bar.

Warning: This class is still in beta. the data is stored in `./temp`, relative to the current working directory. If this directory already exists, it will be overwritten.

get_results()

method for retrieving the results. Called after all experiments have been completed. Any extension of `AbstractCallback` needs to implement this method.

points

classes for representing points in parameter space, as well as associated hellper functions

```
class ema_workbench.em_framework.points.Experiment(name, model_name, policy, scenario,  
                                                    experiment_id)
```

A convenience object that contains a specification of the model, policy, and scenario to run

name

Type

str

model_name

Type

str

policy

Type

Policy instance

scenario

Type

Scenario instance

experiment_id

Type

int

```
class ema_workbench.em_framework.points.ExperimentReplication(scenario, policy, constants,  
                                                             replication=None)
```

helper class that combines scenario, policy, any constants, and replication information (seed etc) into a single dictionary.

This class represent the complete specification of parameters to run for a given experiment.

```
class ema_workbench.em_framework.points.Point(name=None, unique_id=None, **kwargs)
```

```
class ema_workbench.em_framework.points.Policy(name=None, **kwargs)
```

Helper class representing a policy

name

Type

str, int, or float

id

Type

int

all keyword arguments are wrapped into a dict.

```
class ema_workbench.em_framework.points.Scenario(name=None, **kwargs)
```

Helper class representing a scenario

name

Type

str, int, or float

id

Type

int

all keyword arguments are wrapped into a dict.

```
ema_workbench.em_framework.points.combine_cases_factorial(*point_collections)
```

Combine collections of cases in a full factorial manner

Parameters

point_collections (*collection of collections of Point instances*) –

Yields

Point

`ema_workbench.em_framework.points.combine_cases_sampling(*point_collection)`

Combine collections of cases by iterating over the longest collection while sampling with replacement from the others

Parameters

point_collection (*collection of collection of Point instances*) –

Yields

Point

`ema_workbench.em_framework.points.experiment_generator(scenarios, model_structures, policies, combine='factorial')`

generator function which yields experiments

Parameters

- **scenarios** (*iterable of dicts*) –
- **model_structures** (*list*) –
- **policies** (*list*) –
- **{'factorial' (combine =)}** – controls how to combine scenarios, policies, and model_structures into experiments.
- **sample'}** – controls how to combine scenarios, policies, and model_structures into experiments.

Notes

if combine is 'factorial' then this generator is essentially three nested loops: for each model structure, for each policy, for each scenario, return the experiment. This means that designs should not be a generator because this will be exhausted after the running the first policy on the first model. if combine is 'zipover' then this generator cycles over scenarios, policies and model structures until the longest of the three collections is exhausted.

ema_multiprocessing

support for using the multiprocessing library in combination with the workbench

`ema_workbench.em_framework.ema_multiprocessing.setup_working_directories(models, root_dir)`

copies the working directory of each model to a process specific temporary directory and update the working directory of the model

Parameters

- **models** (*list*) –
- **root_dir** (*str*) –

ema_ipyparallel**util**

utilities used throughout em_framework

class ema_workbench.em_framework.util.**Counter**(startfrom=0)

helper function for generating counter based names for NamedDicts

class ema_workbench.em_framework.util.**NamedDict**(name=<function representation>, **kwargs)

class ema_workbench.em_framework.util.**ProgressTrackingMixIn**(N, reporting_interval, logger, log_progress=False, log_func=<function ProgressTrackingMixIn.<lambda>>)

Mixin for monitoring progress

Parameters

- **N** (*int*) – total number of experiments
- **reporting_interval** (*int*) – nfe between logging progress
- **logger** (*logger instance*) –
- **log_progress** (*bool, optional*) –
- **log_func** (*callable, optional*) – function called with self as only argument, should invoke self._logger with custom log message

i

Type

int

reporting_interval

Type

int

log_progress

Type

bool

log_func

Type

callable

pbar

if log_progress is true, None, if false tqdm.tqdm instance

Type

{None, tqdm.tqdm instance}

ema_workbench.em_framework.util.**combine**(*args)

combine scenario and policy into a single experiment dict

Parameters

args (*two or more dicts that need to be combined*) –

Return type

a single unified dict containing the entries from all dicts

Raises

[*EMAError*](#) – if a keyword argument exists in more than one dict

`ema_workbench.em_framework.util.determine_objects(models, attribute, union=True)`

determine the parameters over which to sample

Parameters

- **models** (*a collection of AbstractModel instances*) –
- **attribute** (*{'uncertainties', 'levers', 'outcomes'}*) –
- **union** (*bool, optional*) – in case of multiple models, sample over the union of levers, or over the intersection of the levers

Return type

collection of Parameter instances

`ema_workbench.em_framework.util.representation(named_dict)`

helper function for generating repr based names for NamedDicts

1.13.2 Connectors

`vensim`

`vensimDLLwrapper`

`pysd_connector`

`netlogo`

`simio`

`excel`

This module provides a base class that can be used to perform EMA on Excel models. It relies on [win32com](#)

```
class ema_workbench.connectors.excel.ExcelModel(name, wd=None, model_file=None,  
                                                default_sheet=None, pointers=None)
```

1.13.3 Analysis

`prim`

A scenario discovery oriented implementation of PRIM.

The implementation of `prim` provided here is data type aware, so categorical variables will be handled appropriately. It also uses a non-standard objective function in the peeling and pasting phase of the algorithm. This algorithm looks at the increase in the mean divided by the amount of data removed. So essentially, it uses something akin to the first order derivative of the original objective function.

The implementation is designed for interactive use in combination with the jupyter notebook.

class `ema_workbench.analysis.prim.PRIMObjectiveFunctions`(*value*)

An enumeration.

class `ema_workbench.analysis.prim.Prim`(*x*, *y*, *threshold*,
 obj_function=`PRIMObjectiveFunctions.LENIENT1`,
 peel_alpha=0.05, *paste_alpha*=0.05, *mass_min*=0.05,
 threshold_type=1, *mode*=`RuleInductionType.BINARY`,
 update_function='default')

Patient rule induction algorithm

The implementation of Prim is tailored to interactive use in the context of scenario discovery

Parameters

- **x** (*DataFrame*) – the independent variables
- **y** (*1d ndarray*) – the dependent variable
- **threshold** (*float*) – the density threshold that a box has to meet
- **obj_function** (*{LENIENT1, LENIENT2, ORIGINAL}*) – the objective function used by PRIM. Defaults to a lenient objective function based on the gain of mean divided by the loss of mass.
- **peel_alpha** (*float, optional*) – parameter controlling the peeling stage (default = 0.05).
- **paste_alpha** (*float, optional*) – parameter controlling the pasting stage (default = 0.05).
- **mass_min** (*float, optional*) – minimum mass of a box (default = 0.05).
- **threshold_type** (*{ABOVE, BELOW}*) – whether to look above or below the threshold value
- **mode** (*{RuleInductionType.BINARY, RuleInductionType.REGRESSION}, optional*) – indicated whether PRIM is used for regression, or for scenario classification in which case *y* should be a binary vector
- **'default'** (*update_function =*) – controls behavior of PRIM after having found a first box. use either the default behavior were all points are removed, or the procedure suggested by guivarch et al (2016) doi:10.1016/j.envsoft.2016.03.006 to simply set all points to be no longer of interest (only valid in binary mode).
- **'guivarch'** – controls behavior of PRIM after having found a first box. use either the default behavior were all points are removed, or the procedure suggested by guivarch et al (2016) doi:10.1016/j.envsoft.2016.03.006 to simply set all points to be no longer of interest (only valid in binary mode).
- **optional** – controls behavior of PRIM after having found a first box. use either the default behavior were all points are removed, or the procedure suggested by guivarch et al (2016) doi:10.1016/j.envsoft.2016.03.006 to simply set all points to be no longer of interest (only valid in binary mode).

See also:

`cart`

property `boxes`

Property for getting a list of box limits

determine_coi(*indices*)

Given a set of indices on *y*, how many cases of interest are there in this set.

Parameters

indices (*ndarray*) – a valid index for y

Returns

the number of cases of interest.

Return type

int

Raises

ValueError – if threshold_type is not either ABOVE or BELOW

find_box()

Execute one iteration of the PRIM algorithm. That is, find one box, starting from the current state of Prim.

property stats

property for getting a list of dicts containing the statistics for each box

class ema_workbench.analysis.prim.**PrimBox**(*prim, box_lims, indices*)

A class that holds information for a specific box

coverage

coverage of currently selected box

Type

float

density

density of currently selected box

Type

float

mean

mean of currently selected box

Type

float

res_dim

number of restricted dimensions of currently selected box

Type

int

mass

mass of currently selected box

Type

float

peeling_trajectory

stats for each box in peeling trajectory

Type

DataFrame

box_lims

list of box lims for each box in peeling trajectory

Type

list

by default, the currently selected box is the last box on the peeling trajectory, unless this is changed via `PrimBox.select()`.

drop_restriction(*uncertainty=""*, *i=-1*)

Drop the restriction on the specified dimension for box *i*

Parameters

- **i** (*int*, *optional*) – defaults to the currently selected box, which defaults to the latest box on the trajectory
- **uncertainty** (*str*) –

Replace the limits in box *i* with a new box where for the specified uncertainty the limits of the initial box are being used. The resulting box is added to the peeling trajectory.

inspect(*i=None*, *style='table'*, ***kwargs*)

Write the stats and box limits of the user specified box to standard out. if *i* is not provided, the last box will be printed

Parameters

- **i** (*int or list of ints*, *optional*) – the index of the box, defaults to currently selected box
- **style** (*{'table', 'graph', 'data'}*) – the style of the visualization. ‘table’ prints the stats and boxlim. ‘graph’ creates a figure. ‘data’ returns a list of tuples, where each tuple contains the stats and the box_lims.
- **that** (*additional kwargs are passed to the helper function*) –
- **graph** (*generates the table or*) –

resample(*i=None*, *iterations=10*, *p=0.5*)

Calculate resample statistics for candidate box *i*

Parameters

- **i** (*int*, *optional*) –
- **iterations** (*int*, *optional*) –
- **p** (*float*, *optional*) –

Return type

DataFrame

select(*i*)

select an entry from the peeling and pasting trajectory and update the prim box to this selected box.

Parameters

i (*int*) – the index of the box to select.

show_pairs_scatter(*i=None*, *dims=None*, *cdf=False*)

Make a pair wise scatter plot of all the restricted dimensions with color denoting whether a given point is of interest or not and the boxlims superimposed on top.

Parameters

- **i** (*int*, *optional*) –
- **dims** (*list of str*, *optional*) – dimensions to show, defaults to all restricted dimensions
- **cdf** (*bool*, *optional*) – plot diag as cdf or pdf

Return type

seaborn PairGrid

show_ppt()

show the peeling and pasting trajectory in a figure

show_tradeoff(*cmap=<matplotlib.colors.ListedColormap object>, annotated=False*)

Visualize the trade off between coverage and density. Color is used to denote the number of restricted dimensions.

Parameters

- **cmap** (*valid matplotlib colormap*) –
- **annotated** (*bool, optional. Shows point labels if True.*) –

Return type

a Figure instance

update(*box_lims, indices*)

update the box to the provided box limits.

Parameters

- **box_lims** (*DataFrame*) – the new box_lims
- **indices** (*ndarray*) – the indices of y that are inside the box

write_ppt_to_stdout()

write the peeling and pasting trajectory to stdout

ema_workbench.analysis.prim.pca_preprocess(*experiments, y, subsets=None, exclude={}*)

perform PCA to preprocess experiments before running PRIM

Pre-process the data by performing a pca based rotation on it. This effectively turns the algorithm into PCA-PRIM as described in [Dalal et al \(2013\)](#)**Parameters**

- **experiments** (*DataFrame*) –
- **y** (*ndarray*) – one dimensional binary array
- **subsets** (*dict, optional*) – expects a dictionary with group name as key and a list of uncertainty names as values. If this is used, a constrained PCA-PRIM is executed
- **exclude** (*list of str, optional*) – the uncertainties that should be excluded from the rotation

Returns

- *rotated_experiments* – DataFrame
- *rotation_matrix* – DataFrame

Raises**RuntimeError** – if mode is not binary (i.e. y is not a binary classification). if X contains non numeric columns**ema_workbench.analysis.prim.run_constrained_prim**(*experiments, y, issignificant=True, **kwargs*)

Run PRIM repeatedly while constraining the maximum number of dimensions available in x

Improved usage of PRIM as described in [Kwakkel \(2019\)](#).**Parameters**

- **x** (*DataFrame*) –
- **y** (*numpy array*) –
- **issignificant** (*bool, optional*) – if True, run prim only on subsets of dimensions that are significant for the initial PRIM on the entire dataset.
- ****kwargs** (*any additional keyword arguments are passed on to PRIM*) –

Return type

PrimBox instance

```
ema_workbench.analysis.prim.setup_prim(results, classify, threshold, incl_unc=[], **kwargs)
```

Helper function for setting up the prim algorithm

Parameters

- **results** (*tuple*) – tuple of DataFrame and dict with numpy arrays the return from perform_experiments().
- **classify** (*str or callable*) – either a string denoting the outcome of interest to use or a function.
- **threshold** (*double*) – the minimum score on the density of the last box on the peeling trajectory. In case of a binary classification, this should be between 0 and 1.
- **incl_unc** (*list of str, optional*) – list of uncertainties to include in prim analysis
- **kwargs** (*dict*) – valid keyword arguments for prim.Prim

Return type

a Prim instance

Raises

- **PrimException** – if data resulting from classify is not a 1-d array.
- **TypeError** – if classify is not a string or a callable.

cart

A scenario discovery oriented implementation of CART. It essentially is a wrapper around scikit-learn's version of CART.

```
class ema_workbench.analysis.cart.CART(x, y, mass_min=0.05, mode=RuleInductionType.BINARY)
```

CART algorithm

can be used in a manner similar to PRIM. It provides access to the underlying tree, but it can also show the boxes described by the tree in a table or graph form similar to prim.

Parameters

- **x** (*DataFrame*) –
- **y** (*1D ndarray*) –
- **mass_min** (*float, optional*) – a value between 0 and 1 indicating the minimum fraction of data points in a terminal leaf. Defaults to 0.05, identical to prim.
- **mode** (*{BINARY, CLASSIFICATION, REGRESSION}*) – indicates the mode in which CART is used. Binary indicates binary classification, classification is multiclass, and regression is regression.

boxes

list of DataFrame box lims

Type

list

stats

list of dicts with stats

Type

list

Notes

This class is a wrapper around scikit-learn's CART algorithm. It provides an interface to CART that is more oriented towards scenario discovery, and shared some methods with PRIM

See also:

`prim`

property boxes

rtype: list with boxlims for each terminal leaf

build_tree()

train CART on the data

show_tree(*mplfig=True, format='png'*)

return a png (defaults) or svg of the tree

On Windows, graphviz needs to be installed with conda.

Parameters

- **mplfig** (*bool, optional*) – if true (default) returns a matplotlib figure with the tree, otherwise, it returns the output as bytes
- **format** (*{'png', 'svg'}, default 'png'*) – Gives a format of the output.

property stats

rtype: list with scenario discovery statistics for each terminal leaf

`ema_workbench.analysis.cart.setup_cart(results, classify, incl_unc=None, mass_min=0.05)`

helper function for performing cart in combination with data generated by the workbench.

Parameters

- **results** (*tuple of DataFrame and dict with numpy arrays*) – the return from `perform_experiments()`.
- **classify** (*string, function or callable*) – either a string denoting the outcome of interest to use or a function.
- **incl_unc** (*list of strings, optional*) –
- **mass_min** (*float, optional*) –

Raises

TypeError – if classify is not a string or a callable.

clusterer

This module provides time series clustering functionality using complex invariant distance. For details see [Steinmann et al \(2020\)](#)

```
ema_workbench.analysis.clusterer.apply_agglomerative_clustering(distances, n_clusters,
                                                                metric='precomputed',
                                                                linkage='average')
```

apply agglomerative clustering to the distances

Parameters

- **distances** (*ndarray*) –
- **n_clusters** (*int*) –
- **metric** (*str, optional. The distance metric to use. The default is 'precomputed'. For a list of available metrics, see the documentation of `scipy.spatial.distance.pdist`.*) –
- **linkage** (*{'average', 'complete', 'single'}*) –

Return type

1D ndarray with cluster assignment

```
ema_workbench.analysis.clusterer.calculate_cid(data, condensed_form=False)
```

calculate the complex invariant distance between all rows

Parameters

- **data** (*2d ndarray*) –
- **condensed_form** (*bool, optional*) –

Returns

a 2D ndarray with the distances between all time series, or condensed form similar to `scipy.spatial.distance.pdist`

Return type

distances

```
ema_workbench.analysis.clusterer.plot_dendrogram(distances)
```

plot dendrogram for distances

logistic_regression

This module implements logistic regression for scenario discovery.

The module draws its inspiration from Quinn et al (2018) 10.1029/2018WR022743 and Lamontagne et al (2019). The implementation here generalizes their work and embeds it in a more typical scenario discovery workflow with a posteriori selection of the appropriate number of dimensions to include. It is modeled as much as possible on the api used for PRIM and CART.

```
class ema_workbench.analysis.logistic_regression.Logit(x, y, threshold=0.95)
```

Implements an interactive version of logistic regression using BIC based forward selection

Parameters

- **x** (*DataFrame*) –
- **y** (*numpy Array*) –

- **threshold** (*float*) –

coverage

coverage of currently selected model

Type

float

density

density of currently selected model

Type

float

res_dim

number of restricted dimensions of currently selected model

Type

int

peeling_trajectory

stats for each model in peeling trajectory

Type

DataFrame

models

list of models associated with each model on the peeling trajectory

Type

list

inspect (*i*, *step=0.1*)

Inspect one of the models by showing the threshold tradeoff and summary2

Parameters

- **i** (*int*) –
- **step** (*float between [0, 1]*) –

plot_pairwise_scatter (*i*, *threshold=0.95*)

plot pairwise scatter plot of data points, with contours as background

Parameters

- **i** (*int*) –
- **threshold** (*float*) –

Return type

Figure instance

The lower triangle background is a binary contour based on the specified threshold. All axis not shown are set to a default value in the middle of their range

The upper triangle shows a contour map with the conditional probability, again setting all non shown dimensions to a default value in the middle of their range.

run (***kwargs*)

run logistic regression using forward selection using a Bayesian Information Criterion for selecting whether and if so which dimension to add

Parameters

- **details** (*kwargs are passed on to model.fit. For*) –
- **https** (*see*) –

show_threshold_tradeoff(*i*, *cmap*=<matplotlib.colors.ListedColormap object>, *step*=0.1)

Visualize the trade off between coverage and density for a given model *i* across the range of threshold values

Parameters

- **i** (*int*) –
- **cmap** (*valid matplotlib colormap*) –
- **step** (*float, optional*) –

Return type

a Figure instance

show_tradeoff(*cmap*=<matplotlib.colors.ListedColormap object>, *annotated*=False)

Visualize the trade off between coverage and density. Color is used to denote the number of restricted dimensions.

Parameters

- **cmap** (*valid matplotlib colormap*) –
- **annotated** (*bool, optional. Shows point labels if True.*) –

Return type

a Figure instance

update(*model*, *selected*)

helper function for adding a model to the collection of models and update the associated attributes

Parameters

- **model** (*statsmodel fitted logit model*) –
- **selected** (*list of str*) –

dimensional_stacking

This module provides functionality for doing dimensional stacking of uncertain factors in order to reveal patterns in the values for a single outcome of interests. It is inspired by the work reported [here](#) with one deviation.

Rather than using association rules to identify the uncertain factors to use, this code uses random forest based feature scoring instead. It is also possible to use the code provided here in combination with any other feature scoring or factor prioritization technique instead, or by simply selecting uncertain factors in some other manner.

`ema_workbench.analysis.dimensional_stacking.create_pivot_plot`(*x*, *y*, *nr_levels*=3, *labels*=True, *categories*=True, *nbins*=3, *bin_labels*=False)

convenience function for easily creating a pivot plot

Parameters

- **x** (*DataFrame*) –
- **y** (*1d ndarray*) –
- **nr_levels** (*int, optional*) – the number of levels in the pivot table. The number of uncertain factors included in the pivot table is two times the number of levels.
- **labels** (*bool, optional*) – display names of uncertain factors

- **categories** (*bool, optional*) – display category names for each uncertain factor
- **nbins** (*int, optional*) – number of bins to use when discretizing continuous uncertain factors
- **bin_labels** (*bool, optional*) – if True show bin interval / name, otherwise show only a number

Return type

Figure

This function performs feature scoring using random forests, selects a number of high scoring factors based on the specified number of levels, creates a pivot table, and visualizes the table. This is a convenience function. For more control over the process, use the code in this function as a template.

regional_sa

Module offers support for performing basic regional sensitivity analysis. The module can be used to perform regional sensitivity analysis on all uncertainties specified in the experiment array, as well as the ability to zoom in on any given uncertainty in more detail.

`ema_workbench.analysis.regional_sa.plot_cdfs(x, y, ccdf=False)`

plot cumulative density functions for each column in x, based on the classification specified in y.

Parameters

- **x** (*DataFrame*) – the experiments to use in the cdfs
- **y** (*ndarray*) – the categorization for the data
- **ccdf** (*bool, optional*) – if true, plot a complementary cdf instead of a normal cdf.

Return type

a matplotlib Figure instance

feature_scoring

Feature scoring functionality

`ema_workbench.analysis.feature_scoring.CHI2(X, y)`

Compute chi-squared stats between each non-negative feature and class.

This score can be used to select the *n_features* features with the highest values for the test chi-squared statistic from X, which must contain only **non-negative features** such as booleans or frequencies (e.g., term counts in document classification), relative to the classes.

Recall that the chi-square test measures dependence between stochastic variables, so using this function “weeds out” the features that are the most likely to be independent of class and therefore irrelevant for classification.

Read more in the User Guide.

Parameters

- **X** (*{array-like, sparse matrix} of shape (n_samples, n_features)*) – Sample vectors.
- **y** (*array-like of shape (n_samples,)*) – Target vector (class labels).

Returns

- **chi2** (*ndarray of shape (n_features,)*) – Chi2 statistics for each feature.

- **p_values** (*ndarray of shape (n_features,)*) – P-values for each feature.

See also:

f_classif

ANOVA F-value between label/feature for classification tasks.

f_regression

F-value between label/feature for regression tasks.

Notes

Complexity of this algorithm is $O(n_classes * n_features)$.

`ema_workbench.analysis.feature_scoring.F_CLASSIFICATION(X, y)`

Compute the ANOVA F-value for the provided sample.

Read more in the User Guide.

Parameters

- **X** (*{array-like, sparse matrix} of shape (n_samples, n_features)*) – The set of regressors that will be tested sequentially.
- **y** (*ndarray of shape (n_samples,)*) – The target vector.

Returns

- **f_statistic** (*ndarray of shape (n_features,)*) – F-statistic for each feature.
- **p_values** (*ndarray of shape (n_features,)*) – P-values associated with the F-statistic.

See also:

chi2

Chi-squared stats of non-negative features for classification tasks.

f_regression

F-value between label/feature for regression tasks.

`ema_workbench.analysis.feature_scoring.F_REGRESSION(X, y, *, center=True, force_finite=True)`

Univariate linear regression tests returning F-statistic and p-values.

Quick linear model for testing the effect of a single regressor, sequentially for many regressors.

This is done in 2 steps:

1. The cross correlation between each regressor and the target is computed using `r_regression()` as:

$$E[(X[:, i] - \text{mean}(X[:, i])) * (y - \text{mean}(y))] / (\text{std}(X[:, i]) * \text{std}(y))$$

2. It is converted to an F score and then to a p-value.

`f_regression()` is derived from `r_regression()` and will rank features in the same order if all the features are positively correlated with the target.

Note however that contrary to `f_regression()`, `r_regression()` values lie in $[-1, 1]$ and can thus be negative. `f_regression()` is therefore recommended as a feature selection criterion to identify potentially predictive feature for a downstream classifier, irrespective of the sign of the association with the target variable.

Furthermore `f_regression()` returns p-values while `r_regression()` does not.

Read more in the User Guide.

Parameters

- **X** (*{array-like, sparse matrix} of shape (n_samples, n_features)*) – The data matrix.
- **y** (*array-like of shape (n_samples,)*) – The target vector.
- **center** (*bool, default=True*) – Whether or not to center the data matrix *X* and the target vector *y*. By default, *X* and *y* will be centered.
- **force_finite** (*bool, default=True*) – Whether or not to force the F-statistics and associated p-values to be finite. There are two cases where the F-statistic is expected to not be finite:
 - when the target *y* or some features in *X* are constant. In this case, the Pearson’s R correlation is not defined leading to obtain *np.nan* values in the F-statistic and p-value. When *force_finite=True*, the F-statistic is set to *0.0* and the associated p-value is set to *1.0*.
 - when a feature in *X* is perfectly correlated (or anti-correlated) with the target *y*. In this case, the F-statistic is expected to be *np.inf*. When *force_finite=True*, the F-statistic is set to *np.finfo(dtype).max* and the associated p-value is set to *0.0*.

New in version 1.1.

Returns

- **f_statistic** (*ndarray of shape (n_features,)*) – F-statistic for each feature.
- **p_values** (*ndarray of shape (n_features,)*) – P-values associated with the F-statistic.

See also:

r_regression

Pearson’s R between label/feature for regression tasks.

f_classif

ANOVA F-value between label/feature for classification tasks.

chi2

Chi-squared stats of non-negative features for classification tasks.

SelectKBest

Select features based on the k highest scores.

SelectFpr

Select features based on a false positive rate test.

SelectFdr

Select features based on an estimated false discovery rate.

SelectFwe

Select features based on family-wise error rate.

SelectPercentile

Select features based on percentile of the highest scores.

```
ema_workbench.analysis.feature_scoring.get_ex_feature_scores(x, y,
                                                            mode=RuleInductionType.CLASSIFICATION,
                                                            nr_trees=100, max_features=None,
                                                            max_depth=None,
                                                            min_samples_split=2,
                                                            min_samples_leaf=None,
                                                            min_weight_fraction_leaf=0,
                                                            max_leaf_nodes=None,
                                                            bootstrap=True, oob_score=True,
                                                            random_state=None)
```

Get feature scores using extra trees

Parameters

- **x** (*DataFrame*) –
- **y** (*1D nd.array*) –
- **mode** (*{RuleInductionType.CLASSIFICATION, RuleInductionType.REGRESSION}*) –
- **nr_trees** (*int, optional*) – nr. of trees in forest (default=250)
- **max_features** (*int, float, string or None, optional*) – by default, it will use number of features/3, following Jaxa-Rozen & Kwakkel (2018) doi: 10.1016/j.envsoft.2018.06.011 see <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html>
- **max_depth** (*int, optional*) – see <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html>
- **min_samples_split** (*int, optional*) – see <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html>
- **min_samples_leaf** (*int, optional*) – defaults to 1 for N=1000 or lower, from there on proportional to sqrt of N (see discussion in Jaxa-Rozen & Kwakkel (2018) doi: 10.1016/j.envsoft.2018.06.011) see <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html>
- **min_weight_fraction_leaf** (*float, optional*) – see <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html>
- **max_leaf_nodes** (*int or None, optional*) – see <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html>
- **bootstrap** (*bool, optional*) – see <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html>
- **oob_score** (*bool, optional*) – see <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html>
- **random_state** (*int, optional*) – see <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html>

Returns

- *pandas DataFrame* – sorted in descending order of tuples with uncertainty and feature scores
- *object* – either ExtraTreesClassifier or ExtraTreesRegressor

```
ema_workbench.analysis.feature_scoring.get_feature_scores_all(x, y, alg='extra trees',
                                                            mode=RuleInductionType.REGRESSION,
                                                            **kwargs)
```

perform feature scoring for all outcomes using the specified feature scoring algorithm

Parameters

- **x** (*DataFrame*) –
- **y** (*dict of 1d numpy arrays*) – the outcomes, with a string as key, and a 1D array for each outcome
- **alg** (*{'extra trees', 'random forest', 'univariate'}, optional*) –
- **mode** (*{RuleInductionType.REGRESSION, RuleInductionType.CLASSIFICATION}, optional*) –
- **kwargs** (*dict, optional*) – any remaining keyword arguments will be passed to the specific feature scoring algorithm

Return type

DataFrame instance

```
ema_workbench.analysis.feature_scoring.get_rf_feature_scores(x, y,  
                                                            mode=RuleInductionType.CLASSIFICATION,  
                                                            nr_trees=250, max_features='sqrt',  
                                                            max_depth=None,  
                                                            min_samples_split=2,  
                                                            min_samples_leaf=1,  
                                                            bootstrap=True, oob_score=True,  
                                                            random_state=None)
```

Get feature scores using a random forest

Parameters

- **x** (*DataFrame*) –
- **y** (*1D nd.array*) –
- **mode** (*{RuleInductionType.CLASSIFICATION, RuleInductionType.REGRESSION}*) –
- **nr_trees** (*int, optional*) – nr. of trees in forest (default=250)
- **max_features** (*int, optional*) – see <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>
- **max_depth** (*int, optional*) – see <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>
- **min_samples** (*int, optional*) – see <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>
- **min_samples_leaf** (*int, optional*) – see <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>
- **bootstrap** (*bool, optional*) – see <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>
- **oob_score** (*bool, optional*) – see <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>
- **random_state** (*int, optional*) – see <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>

Returns

- *pandas DataFrame* – sorted in descending order of tuples with uncertainty and feature scores
- *object* – either `RandomForestClassifier` or `RandomForestRegressor`

`ema_workbench.analysis.feature_scoring.get_univariate_feature_scores(x, y,`

*score_func=<function
f_classif>*)

calculate feature scores using univariate statistical tests. In case of categorical data, chi square or the Anova F value is used. In case of continuous data the Anova F value is used.

Parameters

- **x** (*DataFrame*) –
- **y** (*1D nd.array*) –
- **score_func** (*{F_CLASSIFICATION, F_REGRESSION, CHI2}*) – the score function to use, one of `f_regression` (regression), or `f_classification` or `chi2` (classification).

Returns

sorted in descending order of tuples with uncertainty and feature scores (i.e. p values in this case).

Return type

pandas DataFrame

plotting

this module provides functions for generating some basic figures. The code can be used as is, or serve as an example for writing your own code.

`ema_workbench.analysis.plotting.envelopes(experiments, outcomes, outcomes_to_show=None,`
group_by=None, grouping_specifiers=None, density=None,
fill=False, legend=True, titles={}, ylabel={}, log=False)

Make envelop plots.

An envelope shows over time the minimum and maximum value for a set of runs over time. It is thus to be used in case of time series data. The function will try to find a result labeled “TIME”. If this is present, these values will be used on the X-axis. In case of Vensim models, TIME is present by default.

Parameters

- **experiments** (*DataFrame*) –
- **outcomes** (*OutcomesDict*) –
- **outcomes_to_show** –
- **str** (*list of*) –
- **str** –
- **optional** –
- **group_by** (*str, optional*) – name of the column in the experimentsto group results by. Alternatively, *index* can be used to use indexing arrays as the basis for grouping.
- **grouping_specifiers** (*iterable or dict, optional*) – set of categories to be used as a basis for grouping by. *Grouping_specifiers* is only meaningful if *group_by* is provided as well. In case of grouping by index, the grouping specifiers should be in a dictionary where the key denotes the name of the group.
- **density** (*{None, HIST, KDE, VIOLIN, BOXPLOT}, optional*) –

- **fill** (*bool, optional*) –
- **legend** (*bool, optional*) –
- **titles** (*dict, optional*) – a way for controlling whether each of the axes should have a title. There are three possibilities. If set to None, no title will be shown for any of the axes. If set to an empty dict, the default, the title is identical to the name of the outcome of interest. If you want to override these default names, provide a dict with the outcome of interest as key and the desired title as value. This dict need only contain the outcomes for which you want to use a different title.
- **ylabels** (*dict, optional*) – way for controlling the ylabels. Works identical to titles.
- **log** (*bool, optional*) – log scale density plot

Returns

- **Figure** (*Figure instance*)
- **axes** (*dict*) – dict with outcome as key, and axes as value. Density axes' are indexed by the outcome followed by `_density`.

Note: the current implementation is limited to seven different categories in case of `group_by`, `categories`, and/or `discretesize`. This limit is due to the colors specified in `COLOR_LIST`.

Examples

```
>>> import util as util
>>> data = util.load_results(r'1000 flu cases.cPickle')
>>> envelopes(data, group_by='policy')
```

will show an envelope for three three different policies, for all the outcomes of interest. while

```
>>> envelopes(data, group_by='policy', categories=['static policy',
          'adaptive policy'])
```

will only show results for the two specified policies, ignoring any results associated with 'no policy'.

```
ema_workbench.analysis.plotting.kde_over_time(experiments, outcomes, outcomes_to_show=None,
                                              group_by=None, grouping_specifiers=None,
                                              colormap='viridis', log=True)
```

Plot a KDE over time. The KDE is is visualized through a heatmap

Parameters

- **experiments** (*DataFrame*) –
- **outcomes** (*OutcomesDict*) –
- **outcomes_to_show** (*list of str, optional*) – list of outcome of interest you want to plot. If empty, all outcomes are plotted. **Note:** just names.
- **group_by** (*str, optional*) – name of the column in the cases array to group results by. Alternatively, *index* can be used to use indexing arrays as the basis for grouping.
- **grouping_specifiers** (*iterable or dict, optional*) – set of categories to be used as a basis for grouping by. Grouping_specifiers is only meaningful if `group_by` is provided

as well. In case of grouping by index, the grouping specifiers should be in a dictionary where the key denotes the name of the group.

- **colormap** (*str*, *optional*) – valid matplotlib color map name
- **log** (*bool*, *optional*) –

Returns

- *list of Figure instances* – a figure instance for each group for each outcome
- *dict* – dict with outcome as key, and axes as value. Density axes' are indexed by the outcome followed by `_density`

```
ema_workbench.analysis.plotting.lines(experiments, outcomes, outcomes_to_show=[], group_by=None,
                                     grouping_specifiers=None, density="", legend=True, titles={},
                                     ylabels={}, experiments_to_show=None, show_envelope=False,
                                     log=False)
```

This function takes the results from `perform_experiments()` and visualizes these as line plots. It is thus to be used in case of time series data. The function will try to find a result labeled “TIME”. If this is present, these values will be used on the X-axis. In case of Vensim models, TIME is present by default.

Parameters

- **experiments** (*DataFrame*) –
- **outcomes** (*OutcomesDict*) –
- **outcomes_to_show** (*list of str*, *optional*) – list of outcome of interest you want to plot. If empty, all outcomes are plotted. **Note:** just names.
- **group_by** (*str*, *optional*) – name of the column in the cases array to group results by. Alternatively, *index* can be used to use indexing arrays as the basis for grouping.
- **grouping_specifiers** (*iterable or dict*, *optional*) – set of categories to be used as a basis for grouping by. Grouping_specifiers is only meaningful if *group_by* is provided as well. In case of grouping by index, the grouping specifiers should be in a dictionary where the key denotes the name of the group.
- **density** (*{None, HIST, KDE, VIOLIN, BOXPLOT}*, *optional*) –
- **legend** (*bool*, *optional*) –
- **titles** (*dict*, *optional*) – a way for controlling whether each of the axes should have a title. There are three possibilities. If set to *None*, no title will be shown for any of the axes. If set to an empty dict, the default, the title is identical to the name of the outcome of interest. If you want to override these default names, provide a dict with the outcome of interest as key and the desired title as value. This dict need only contain the outcomes for which you want to use a different title.
- **ylabels** (*dict*, *optional*) – way for controlling the ylabels. Works identical to titles.
- **experiments_to_show** (*ndarray*, *optional*) – indices of experiments to show lines for, defaults to *None*.
- **show_envelope** (*bool*, *optional*) – show envelope of outcomes. This envelope is the based on the minimum at each column and the maximum at each column.
- **log** (*bool*, *optional*) – log scale density plot

Returns

- **fig** (*Figure instance*)

- **axes** (*dict*) – dict with outcome as key, and axes as value. Density axes’ are indexed by the outcome followed by `_density`.

Note: the current implementation is limited to seven different categories in case of `group_by`, `categories`, and/or `discretesize`. This limit is due to the colors specified in `COLOR_LIST`.

```
ema_workbench.analysis.plotting.multiple_densities(experiments, outcomes, points_in_time=None,
                                                    outcomes_to_show=None, group_by=None,
                                                    grouping_specifiers=None, density=Density.KDE,
                                                    legend=True, titles={}, ylabels={},
                                                    experiments_to_show=None,
                                                    plot_type=PlotType.ENVELOPE, log=False,
                                                    **kwargs)
```

Make an envelope plot with multiple density plots over the run time

Parameters

- **experiments** (*DataFrame*) –
- **outcomes** (*OutcomesDict*) –
- **points_in_time** (*list*) – a list of points in time for which you want to see the density. At the moment up to 6 points in time are supported
- **outcomes_to_show** (*list of str, optional*) – list of outcome of interest you want to plot. If empty, all outcomes are plotted. **Note:** just names.
- **group_by** (*str, optional*) – name of the column in the cases array to group results by. Alternatively, *index* can be used to use indexing arrays as the basis for grouping.
- **grouping_specifiers** (*iterable or dict, optional*) – set of categories to be used as a basis for grouping by. *Grouping_specifiers* is only meaningful if *group_by* is provided as well. In case of grouping by index, the grouping specifiers should be in a dictionary where the key denotes the name of the group.
- **density** (*{Density.KDE, Density.HIST, Density.VIOLIN, Density.BOXPLOT},*) – optional
- **legend** (*bool, optional*) –
- **titles** (*dict, optional*) – a way for controlling whether each of the axes should have a title. There are three possibilities. If set to *None*, no title will be shown for any of the axes. If set to an empty dict, the default, the title is identical to the name of the outcome of interest. If you want to override these default names, provide a dict with the outcome of interest as key and the desired title as value. This dict need only contain the outcomes for which you want to use a different title.
- **ylabels** (*dict, optional*) – way for controlling the ylabels. Works identical to titles.
- **experiments_to_show** (*ndarray, optional*) – indices of experiments to show lines for, defaults to *None*.
- **plot_type** (*{PlotType.ENVELOPE, PlotType.ENV_LIN, PlotType.LINES}, optional*) –
- **log** (*bool, optional*) –

Returns

- **fig** (*Figure instance*)

- **axes** (*dict*) – dict with outcome as key, and axes as value. Density axes' are indexed by the outcome followed by `_density`.

Note: the current implementation is limited to seven different categories in case of `group_by`, `categories`, and/or `discretize`. This limit is due to the colors specified in `COLOR_LIST`.

Note: the connection patches are for some reason not drawn if log scaling is used for the density plots. This appears to be an issue in matplotlib itself.

pairs_plotting

This module provides R style pairs plotting functionality.

```
ema_workbench.analysis.pairs_plotting.pairs_density(experiments, outcomes, outcomes_to_show=[],
                                                    group_by=None, grouping_specifiers=None,
                                                    ylabels={}, point_in_time=-1, log=True,
                                                    gridsize=50, colormap='coolwarm',
                                                    filter_scalar=True)
```

Generate a R style `pairs` hexbin density multiplot. In case of time-series data, the end states are used.

`hexbin` makes hexagonal binning plot of `x` versus `y`, where `x`, `y` are 1-D sequences of the same length, `N`. If `C` is `None` (the default), this is a histogram of the number of occurrences of the observations at `(x[i],y[i])`. For further detail see [matplotlib on hexbin](#)

Parameters

- **experiments** (*DataFrame*) –
- **outcomes** (*dict*) –
- **outcomes_to_show** (*list of str, optional*) – list of outcome of interest you want to plot.
- **group_by** (*str, optional*) – name of the column in the cases array to group results by. Alternatively, *index* can be used to use indexing arrays as the basis for grouping.
- **grouping_specifiers** (*dict, optional*) – dict of categories to be used as a basis for grouping by. `Grouping_specifiers` is only meaningful if `group_by` is provided as well. In case of grouping by index, the grouping specifiers should be in a dictionary where the key denotes the name of the group.
- **ylabels** (*dict, optional*) – `ylabels` is a dictionary with the outcome names as keys, the specified values will be used as labels for the y axis.
- **point_in_time** (*float, optional*) – the point in time at which the scatter is to be made. If `None` is provided (default), the end states are used. `point_in_time` should be a valid value on time
- **log** (*bool, optional*) – indicating whether density should be log scaled. Defaults to `True`.
- **gridsize** (*int, optional*) – controls the gridsize for the hexagonal binning. (default = 50)
- **cmap** (*str*) – color map that is to be used in generating the hexbin. For details on the available maps, see [pylab](#). (Defaults = `coolwarm`)

- **filter_scalar** (*bool*, *optional*) – remove the non-time-series outcomes. Defaults to `True`.

Returns

- *fig* – the figure instance
- *dict* – key is tuple of names of outcomes, value is associated axes instance

```
ema_workbench.analysis.pairs_plotting.pairs_lines(experiments, outcomes, outcomes_to_show=[],  
                                                  group_by=None, grouping_specifiers=None,  
                                                  ylabels={}, legend=True, **kwargs)
```

Generate a R style `pairs` lines multiplot. It shows the behavior of two outcomes over time against each other. The origin is denoted with a circle and the end is denoted with a '+'.

Parameters

- **experiments** (*DataFrame*) –
- **outcomes** (*dict*) –
- **outcomes_to_show** (*list of str*, *optional*) – list of outcome of interest you want to plot.
- **group_by** (*str*, *optional*) – name of the column in the cases array to group results by. Alternatively, *index* can be used to use indexing arrays as the basis for grouping.
- **grouping_specifiers** (*dict*, *optional*) – dict of categories to be used as a basis for grouping by. Grouping_specifiers is only meaningful if *group_by* is provided as well. In case of grouping by index, the grouping specifiers should be in a dictionary where the key denotes the name of the group.
- **ylabels** (*dict*, *optional*) – ylabels is a dictionary with the outcome names as keys, the specified values will be used as labels for the y axis.
- **legend** (*bool*, *optional*) – if true, and *group_by* is given, show a legend.
- **point_in_time** (*float*, *optional*) – the point in time at which the scatter is to be made. If `None` is provided (default), the end states are used. *point_in_time* should be a valid value on time

Returns

- *fig* – the figure instance
- *dict* – key is tuple of names of outcomes, value is associated axes instance

```
ema_workbench.analysis.pairs_plotting.pairs_scatter(experiments, outcomes, outcomes_to_show=[],  
                                                    group_by=None, grouping_specifiers=None,  
                                                    ylabels={}, legend=True, point_in_time=-1,  
                                                    filter_scalar=False, **kwargs)
```

Generate a R style `pairs` scatter multiplot. In case of time-series data, the end states are used.

Parameters

- **experiments** (*DataFrame*) –
- **outcomes** (*dict*) –
- **outcomes_to_show** (*list of str*, *optional*) – list of outcome of interest you want to plot.
- **group_by** (*str*, *optional*) – name of the column in the cases array to group results by. Alternatively, *index* can be used to use indexing arrays as the basis for grouping.

- **grouping_specifiers** (*dict, optional*) – dict of categories to be used as a basis for grouping by. Grouping_specifiers is only meaningful if group_by is provided as well. In case of grouping by index, the grouping specifiers should be in a dictionary where the key denotes the name of the group.
- **ylabels** (*dict, optional*) – ylabels is a dictionary with the outcome names as keys, the specified values will be used as labels for the y axis.
- **legend** (*bool, optional*) – if true, and group_by is given, show a legend.
- **point_in_time** (*float, optional*) – the point in time at which the scatter is to be made. If None is provided (default), the end states are used. point_in_time should be a valid value on time
- **filter_scalar** (*bool, optional*) – remove the non-time-series outcomes. Defaults to True.

Returns

- **fig** (*Figure instance*) – the figure instance
- **axes** (*dict*) – key is tuple of names of outcomes, value is associated axes instance
- .. note:: the current implementation is limited to seven different – categories in case of column, categories, and/or discretesize. This limit is due to the colors specified in COLOR_LIST.

parcoords

This module offers a general purpose parallel coordinate plotting Class using matplotlib.

class ema_workbench.analysis.parcoords.**ParallelAxes**(*limits, formatter=None, fontsize=14, rot=90*)

Base class for creating a parallel axis plot.

Parameters

- **limits** (*DataFrame*) – A DataFrame specifying the limits for each dimension in the data set. For categorical data, the first cell should contain all categories. See get_limits for more details.
- **formatter** (*dict, optional*) – dict with precision format strings for minima and maxima, use column name as key. If column is not present, or no formatter dict is provided, precision formatting defaults to .2f
- **fontsize** (*int, optional*) – fontsize for defaults text items
- **rot** (*float, optional*) – rotation of axis labels

limits

A DataFrame specifying the limits for each dimension in the data set. For categorical data, the first cell should contain all categories.

Type

DataFrame

recoding

non numeric columns are converting to integer variables

Type

dict

flipped_axes

set of Axes that are to be shown flipped

Type

set

axis_labels**Type**

list of str

fontsize**Type**

int

fig**Type**

a matplotlib Figure instance

axes**Type**

list of matplotlib Axes instances

ticklabels**Type**

list of str

datalabels

labels associated with lines

Type

list of str

Notes

The basic setup of the Parallel Axis plot is a row of mpl Axes instances, with all whitespace in between removed. The number of Axes is the number of columns - 1.

invert_axis(*axis*)

flip direction for specified axis

Parameters

axis (*str or list of str*) –

legend()

add a legend to the figure

plot(*data*, *color=None*, *label=None*, *kwargs*)**

plot data on parallel axes

Parameters

- **data** (*DataFrame or Series*) –
- **color** (*valid mpl color, optional*) –
- **label** (*str, optional*) –

- **plot** (any additional kwargs will be passed to matplotlib's) –
- **method.** –
- **initializing** (Data is normalized using the limits specified when) –
- **ParallelAxis.** –

`ema_workbench.analysis.parcoords.get_limits(data)`

helper function to get limits of a DataFrame that can serve as input to ParallelAxis

Parameters

data (*DataFrame*) –

Return type

DataFrame

b_and_w_plotting

This module offers basic functionality for converting a matplotlib figure to black and white. The provided functionality is largely determined by what is needed for the workbench.

`ema_workbench.analysis.b_and_w_plotting.set_fig_to_bw(fig, style='hatching', line_style='continuous')`

TODO it would be nice if for lines you can select either markers, gray scale, or simple black

Take each axes in the figure and transform its content to black and white. Lines are transformed based on different line styles. Fills such as can be used in *meth*:envelopes are gray-scaled. Heathmaps are also gray-scaled.

derived from and expanded for my use from: <https://stackoverflow.com/questions/7358118/matplotlib-black-white-colormap-with-dashes-dots-etc>

Parameters

- **fig** (*figure*) – the figure to transform to black and white
- **style** (*{HATCHING, GREYSCALE}*) – parameter controlling how collections are transformed.
- **line_style** (*str, {'continuous', 'black', None}*) –

plotting_util

Plotting utility functions

```
ema_workbench.analysis.plotting_util.COLOR_LIST = [(0.12156862745098039,
0.4666666666666667, 0.7058823529411765), (1.0, 0.4980392156862745, 0.054901960784313725),
(0.17254901960784313, 0.6274509803921569, 0.17254901960784313), (0.8392156862745098,
0.15294117647058825, 0.1568627450980392), (0.5803921568627451, 0.403921568627451,
0.7411764705882353), (0.5490196078431373, 0.33725490196078434, 0.29411764705882354),
(0.8901960784313725, 0.4666666666666667, 0.7607843137254902), (0.4980392156862745,
0.4980392156862745, 0.4980392156862745), (0.7372549019607844, 0.7411764705882353,
0.13333333333333333), (0.09019607843137255, 0.7450980392156863, 0.8117647058823529)]
```

Default color list

`class ema_workbench.analysis.plotting_util.Density(value)`

Enum for different types of density plots

BOXENPLOT = 'boxenplot'

constant for plotting density as a boxenplot

BOXPLOT = 'boxplot'

constant for plotting density as a boxplot

HIST = 'hist'

constant for plotting density as a histogram

KDE = 'kde'

constant for plotting density as a kernel density estimate

VIOLIN = 'violin'

constant for plotting density as a violin plot, which combines a Gaussian density estimate with a boxplot

class ema_workbench.analysis.plotting_util.**LegendEnum**(*value*)

Enum for different styles of legends

class ema_workbench.analysis.plotting_util.**PlotType**(*value*)

An enumeration.

ENVELOPE = 'envelope'

constant for plotting envelopes

ENV_LIN = 'env_lin'

constant for plotting envelopes with lines

LINES = 'lines'

constant for plotting lines

scenario_discovery_util

Scenario discovery utilities used by both cart and prim

class ema_workbench.analysis.scenario_discovery_util.**RuleInductionType**(*value*)

An enumeration.

BINARY = 'binary'

constant indicating binary classification mode. This is the most common used mode in scenario discovery

CLASSIFICATION = 'classification'

constant indicating classification mode

REGRESSION = 'regression'

constant indicating regression mode

1.13.4 Util

ema_exceptions

Exceptions and warning used internally by the EMA workbench. In line with advice given in [PEP 8](#).

exception ema_workbench.util.ema_exceptions.**CaseError**(*message*, *case*, *policy=None*)

error to be used when a particular run creates an error. The character of the error can be specified as the message, and the actual case that gave rise to the error.

exception ema_workbench.util.ema_exceptions.**EMAError**(*args)

Base EMA error

exception `ema_workbench.util.ema_exceptions.EMAParallelError(*args)`

parallel EMA error

exception `ema_workbench.util.ema_exceptions.EMAWarning(*args)`

base EMA warning class

ema_logging

This module contains code for logging EMA processes. It is modeled on the default [logging approach that comes with Python](#). This logging system will also work in case of multiprocessing.

`ema_workbench.util.ema_logging.get_rootlogger()`

Returns root logger used by the EMA workbench

Return type

the logger of the EMA workbench

`ema_workbench.util.ema_logging.log_to_stderr(level=None)`

Turn on logging and add a handler which prints to stderr

Parameters

level (*int*) – minimum level of the messages that will be logged

`ema_workbench.util.ema_logging.temporary_filter(name='EMA', level=0, funcname=None)`

temporary filter log message

Parameters

- **name** (*str or list of str, optional*) – logger on which to apply the filter.
- **level** (*int, or list of int, optional*) – don't log message of this level or lower
- **funcname** (*str or list of str, optional*) – don't log message of this function
- **logger** (*all modules have their own unique*) –
- **ema_workbench.analysis.prim** (*(e.g.)*) –

utilities

This module provides various convenience functions and classes.

`ema_workbench.util.utilities.load_results(file_name)`

load the specified bz2 file. the file is assumed to be saved using `save_results`.

Parameters

file_name (*str*) – the path to the file

Raises

IOError if file not found –

`ema_workbench.util.utilities.merge_results(results1, results2)`

convenience function for merging the return from `perform_experiments()`.

The function merges `results2` with `results1`. For the experiments, it generates an empty array equal to the size of the sum of the experiments. As dtype is used the dtype from the experiments in `results1`. The function assumes that the ordering of dtypes and names is identical in both results.

A typical use case for this function is in combination with `experiments_to_cases()`. Using `experiments_to_cases()` one extracts the cases from a first set of experiments. One then performs these

cases on a different model or policy, and then one wants to merge these new results with the old result for further analysis.

Parameters

- **results1** (*tuple*) – first results to be merged
- **results2** (*tuple*) – second results to be merged

Return type

the merged results

`ema_workbench.util.utilities.process_replications(data, aggregation_func=<function mean>)`

Convenience function for processing the replications of a stochastic model's outcomes.

The default behavior is to take the mean of the replications. This reduces the dimensionality of the outcomes from (experiments * replications * outcome_shape) to (experiments * outcome_shape), where outcome_shape is 0-d for scalars, 1-d for time series, and 2-d for arrays.

The function can take either the outcomes (dictionary: keys are outcomes of interest, values are arrays of data) or the results (tuple: experiments as DataFrame, outcomes as dictionary) of a set of simulation experiments.

Parameters

- **data** (*dict*, *tuple*) – outcomes or results of a set of experiments
- **aggregation_func** (*callable*, *optional*) – aggregation function to be applied, defaults to `np.mean`.

Return type

dict, tuple

`ema_workbench.util.utilities.save_results(results, file_name)`

save the results to the specified tar.gz file.

The way in which results are stored depends. Experiments are saved as csv. Outcomes depend on the outcome type. Scalar, and <3D arrays are saved as csv files. Higher dimensional arrays are stored as .npz files.

Parameters

- **results** (*tuple*) – the return of `perform_experiments`
- **file_name** (*str*) – the path of the file

Raises

IOError if file not found –

1.14 Indices and tables

- [genindex](#)
- [modindex](#)
- [search](#)
- [Glossary](#)

PYTHON MODULE INDEX

e

- `ema_workbench.analysis.b_and_w_plotting`, 201
- `ema_workbench.analysis.cart`, 183
- `ema_workbench.analysis.clusterer`, 185
- `ema_workbench.analysis.dimensional_stacking`, 187
- `ema_workbench.analysis.feature_scoring`, 188
- `ema_workbench.analysis.logistic_regression`, 185
- `ema_workbench.analysis.pairs_plotting`, 197
- `ema_workbench.analysis.parcoords`, 199
- `ema_workbench.analysis.plotting`, 193
- `ema_workbench.analysis.plotting_util`, 201
- `ema_workbench.analysis.prim`, 178
- `ema_workbench.analysis.regional_sa`, 188
- `ema_workbench.analysis.scenario_discovery_util`, 202
- `ema_workbench.connectors.excel`, 178
- `ema_workbench.em_framework.callbacks`, 172
- `ema_workbench.em_framework.ema_multiprocessing`, 176
- `ema_workbench.em_framework.evaluators`, 160
- `ema_workbench.em_framework.experiment_runner`, 171
- `ema_workbench.em_framework.model`, 148
- `ema_workbench.em_framework.optimization`, 162
- `ema_workbench.em_framework.outcomes`, 154
- `ema_workbench.em_framework.outputspace_exploration`, 165
- `ema_workbench.em_framework.parameters`, 150
- `ema_workbench.em_framework.points`, 174
- `ema_workbench.em_framework.salib_samplers`, 171
- `ema_workbench.em_framework.samplers`, 167
- `ema_workbench.em_framework.util`, 177
- `ema_workbench.util.ema_exceptions`, 202
- `ema_workbench.util.ema_logging`, 203
- `ema_workbench.util.utilities`, 203

INDEX

A

AbstractCallback (class in *ema_workbench.em_framework.callbacks*), 172

AbstractModel (class in *ema_workbench.em_framework.model*), 148

AbstractOutcome (class in *ema_workbench.em_framework.outcomes*), 154

AbstractSampler (class in *ema_workbench.em_framework.samplers*), 167

apply_agglomerative_clustering() (in module *ema_workbench.analysis.clusterer*), 185

ArchiveLogger (class in *ema_workbench.em_framework.optimization*), 162

ArrayOutcome (class in *ema_workbench.em_framework.outcomes*), 155

as_dict() (*ema_workbench.em_framework.model.AbstractModel* method), 149

as_dict() (*ema_workbench.em_framework.model.FileModel* method), 150

AutoAdaptiveOutputSpaceExploration (class in *ema_workbench.em_framework.outputspace_exploration*), 166

axes (*ema_workbench.analysis.parcoords.ParallelAxes* attribute), 200

axis_labels (*ema_workbench.analysis.parcoords.ParallelAxes* attribute), 200

B

BINARY (*ema_workbench.analysis.scenario_discovery_util.RuleInductionType* attribute), 202

BooleanParameter (class in *ema_workbench.em_framework.parameters*), 150

box_lims (*ema_workbench.analysis.prim.PrimBox* attribute), 180

BOXENPLOT (*ema_workbench.analysis.plotting_util.Density* attribute), 201

boxes (*ema_workbench.analysis.cart.CART* attribute), 183

boxes (*ema_workbench.analysis.cart.CART* property), 184

boxes (*ema_workbench.analysis.prim.Prim* property), 179

BOXPLOT (*ema_workbench.analysis.plotting_util.Density* attribute), 201

build_tree() (*ema_workbench.analysis.cart.CART* method), 184

C

calculate_cid() (in module *ema_workbench.analysis.clusterer*), 185

CART (class in *ema_workbench.analysis.cart*), 183

CaseError, 202

cat_for_index() (*ema_workbench.em_framework.parameters.CategoricalParameter* method), 151

CategoricalParameter (class in *ema_workbench.em_framework.parameters*), 150

Category (class in *ema_workbench.em_framework.parameters*), 151

CHI2() (in module *ema_workbench.analysis.feature_scoring*), 188

CLASSIFICATION (*ema_workbench.analysis.scenario_discovery_util.RuleInductionType* attribute), 202

cleanup() (*ema_workbench.em_framework.model.AbstractModel* method), 149

COLOR_LIST (in module *ema_workbench.analysis.plotting_util*), 201

combine() (in module *ema_workbench.em_framework.util*), 177

combine_cases_factorial() (in module *ema_workbench.em_framework.points*), 175

combine_cases_sampling() (in module *ema_workbench.em_framework.points*), 175

Constant (class in *ema_workbench.em_framework.parameters*), 151

Constraint (class in *ema_workbench.em_framework.outcomes*), 156

Convergence (class in *ema_workbench.em_framework.optimization*), 162

`ema_workbench.em_framework.optimization)`,
163
`Counter` (class in `ema_workbench.em_framework.util`),
177
`coverage` (`ema_workbench.analysis.logistic_regression.Logit`
attribute), 186
`coverage` (`ema_workbench.analysis.prim.PrimBox` at-
tribute), 180
`create_pivot_plot()` (in module
`ema_workbench.analysis.dimensionality_stacking`),
187

D

`datalabels` (`ema_workbench.analysis.parcoords.ParallelAxis`
attribute), 200
`DefaultCallback` (class in
`ema_workbench.em_framework.callbacks`),
173
`DefaultDesigns` (class in
`ema_workbench.em_framework.samplers`),
168
`Density` (class in `ema_workbench.analysis.plotting_util`),
201
`density` (`ema_workbench.analysis.logistic_regression.Logit`
attribute), 186
`density` (`ema_workbench.analysis.prim.PrimBox`
attribute), 180
`determine_coi()` (`ema_workbench.analysis.prim.Prim`
method), 179
`determine_nr_of_designs()`
(`ema_workbench.em_framework.samplers.FullFactorial`
method), 168
`determine_objects()` (in module
`ema_workbench.em_framework.util`), 178
`determine_parameters()` (in module
`ema_workbench.em_framework.samplers`),
169
`drop_restriction()` (`ema_workbench.analysis.prim.PrimBox`
method), 181
`dtype` (`ema_workbench.em_framework.outcomes.AbstractOutcome`
attribute), 154
`dtype` (`ema_workbench.em_framework.outcomes.ArrayOutcome`
attribute), 156
`dtype` (`ema_workbench.em_framework.outcomes.ScalarOutcome`
attribute), 158
`dtype` (`ema_workbench.em_framework.outcomes.TimeSeriesOutcome`
attribute), 159

E

`ema_workbench.analysis.b_and_w_plotting`
module, 201
`ema_workbench.analysis.cart`
module, 183
`ema_workbench.analysis.clusterer`
module, 185
`ema_workbench.analysis.dimensionality_stacking`
module, 187
`ema_workbench.analysis.feature_scoring`
module, 188
`ema_workbench.analysis.logistic_regression`
module, 185
`ema_workbench.analysis.pairs_plotting`
module, 197
`ema_workbench.analysis.parcoords`
module, 199
`ema_workbench.analysis.plotting`
module, 193
`ema_workbench.analysis.plotting_util`
module, 201
`ema_workbench.analysis.prim`
module, 178
`ema_workbench.analysis.regional_sa`
module, 188
`ema_workbench.analysis.scenario_discovery_util`
module, 202
`ema_workbench.connectors.excel`
module, 178
`ema_workbench.em_framework.callbacks`
module, 172
`ema_workbench.em_framework.ema_multiprocessing`
module, 176
`ema_workbench.em_framework.evaluators`
module, 160
`ema_workbench.em_framework.experiment_runner`
module, 171
`ema_workbench.em_framework.model`
module, 148
`ema_workbench.em_framework.optimization`
module, 162
`ema_workbench.em_framework.outcomes`
module, 154
`ema_workbench.em_framework.outputspace_exploration`
module, 165
`ema_workbench.em_framework.parameters`
module, 150
`ema_workbench.em_framework.points`
module, 174
`ema_workbench.em_framework.salib_sampler`
module, 171
`ema_workbench.em_framework.samplers`
module, 167
`ema_workbench.em_framework.util`
module, 177
`ema_workbench.util.ema_exceptions`
module, 202
`ema_workbench.util.ema_logging`
module, 203
`ema_workbench.util.utilities`

module, 203
 EMAError, 202
 EMAParallelError, 202
 EMAWarning, 203
 ENV_LIN (*ema_workbench.analysis.plotting_util.PlotType* attribute), 202
 ENVELOPE (*ema_workbench.analysis.plotting_util.PlotType* attribute), 202
 envelopes() (in module *ema_workbench.analysis.plotting*), 193
 epsilon_nondominated() (in module *ema_workbench.em_framework.optimization*), 164
 EpsilonIndicatorMetric (class in *ema_workbench.em_framework.optimization*), 163
 EpsilonProgress (class in *ema_workbench.em_framework.optimization*), 163
 evaluate_experiments() (*ema_workbench.em_framework.evaluators.IpyparallelEvaluator* method), 160
 evaluate_experiments() (*ema_workbench.em_framework.evaluators.MultiprocessingEvaluator* method), 160
 evaluate_experiments() (*ema_workbench.em_framework.evaluators.SequentialEvaluator* method), 160
 ExcelModel (class in *ema_workbench.connectors.excel*), 178
 expected_range (*ema_workbench.em_framework.outcomes.ArrayOutcome* attribute), 156
 expected_range (*ema_workbench.em_framework.outcomes.ScalarOutcome* attribute), 157
 expected_range (*ema_workbench.em_framework.outcomes.TimeSeriesOutcome* attribute), 159
 Experiment (class in *ema_workbench.em_framework.points*), 174
 experiment_generator() (in module *ema_workbench.em_framework.points*), 176
 experiment_id (*ema_workbench.em_framework.points.Experiment* attribute), 175
 ExperimentReplication (class in *ema_workbench.em_framework.points*), 175
 ExperimentRunner (class in *ema_workbench.em_framework.experiment_runner*), 171
F
 F_CLASSIFICATION() (in module *ema_workbench.analysis.feature_scoring*), 189
 F_REGRESSION() (in module *ema_workbench.analysis.feature_scoring*), 189
 FASTSampler (class in *ema_workbench.em_framework.salib_samplers*), 171
 fig (*ema_workbench.analysis.parcoords.ParallelAxes* attribute), 200
 FileBasedCallback (class in *ema_workbench.em_framework.callbacks*), 174
 FileModel (class in *ema_workbench.em_framework.model*), 150
 finalize() (*ema_workbench.em_framework.evaluators.IpyparallelEvaluator* method), 160
 finalize() (*ema_workbench.em_framework.evaluators.MultiprocessingEvaluator* method), 160
 finalize() (*ema_workbench.em_framework.evaluators.SequentialEvaluator* method), 160
 find_box() (*ema_workbench.analysis.prim.Prim* method), 180
 flipped_axes (*ema_workbench.analysis.parcoords.ParallelAxes* attribute), 199
 fontsize (*ema_workbench.analysis.parcoords.ParallelAxes* attribute), 200
 from_disk() (*ema_workbench.em_framework.outcomes.AbstractOutcome* class method), 154
 from_disk() (*ema_workbench.em_framework.outcomes.ArrayOutcome* class method), 156
 from_disk() (*ema_workbench.em_framework.outcomes.ScalarOutcome* class method), 158
 from_disk() (*ema_workbench.em_framework.outcomes.TimeSeriesOutcome* class method), 159
 from_dist() (*ema_workbench.em_framework.parameters.CategoricalParameter* method), 151
 from_dist() (*ema_workbench.em_framework.parameters.IntegerParameter* method), 152
 from_dist() (*ema_workbench.em_framework.parameters.Parameter* class method), 152
 from_dist() (*ema_workbench.em_framework.parameters.RealParameter* class method), 153
 FullFactorialSampler (class in *ema_workbench.em_framework.samplers*), 168
 function (*ema_workbench.em_framework.outcomes.AbstractOutcome* attribute), 154
 function (*ema_workbench.em_framework.outcomes.ArrayOutcome* attribute), 155
 function (*ema_workbench.em_framework.outcomes.Constraint* attribute), 157
 function (*ema_workbench.em_framework.outcomes.ScalarOutcome* attribute), 157
 function (*ema_workbench.em_framework.outcomes.TimeSeriesOutcome* attribute), 159

G

[generate_designs\(\)](#) (*ema_workbench.em_framework.samplers.AbstractSampler* method), 167
[generate_designs\(\)](#) (*ema_workbench.em_framework.samplers.FullFactorialSampler* method), 168
[generate_samples\(\)](#) (*ema_workbench.em_framework.samplers.AbstractSampler* method), 167
[generate_samples\(\)](#) (*ema_workbench.em_framework.samplers.FullFactorialSampler* method), 168
[generate_samples\(\)](#) (*ema_workbench.em_framework.samplers.UniformHSSampler* method), 169
[GenerationalDistanceMetric](#) (class in *ema_workbench.em_framework.optimization*), 163
[get_ex_feature_scores\(\)](#) (in module *ema_workbench.analysis.feature_scoring*), 190
[get_feature_scores_all\(\)](#) (in module *ema_workbench.analysis.feature_scoring*), 191
[get_limits\(\)](#) (in module *ema_workbench.analysis.parcoords*), 201
[get_results\(\)](#) (*ema_workbench.em_framework.callbacks.AbstractCallback* method), 173
[get_results\(\)](#) (*ema_workbench.em_framework.callbacks.DefaultCallback* method), 174
[get_results\(\)](#) (*ema_workbench.em_framework.callbacks.FileBasedCallback* method), 174
[get_rf_feature_scores\(\)](#) (in module *ema_workbench.analysis.feature_scoring*), 192
[get_rootlogger\(\)](#) (in module *ema_workbench.util.ema_logging*), 203
[get_SALib_problem\(\)](#) (in module *ema_workbench.em_framework.salib_samplers*), 171
[get_univariate_feature_scores\(\)](#) (in module *ema_workbench.analysis.feature_scoring*), 193

H

[HIST](#) (*ema_workbench.analysis.plotting_util.Density* attribute), 202
[HypervolumeMetric](#) (class in *ema_workbench.em_framework.optimization*), 163

I

[i](#) (*ema_workbench.em_framework.callbacks.AbstractCallback* attribute), 173
[i](#) (*ema_workbench.em_framework.util.ProgressTrackingMixin* attribute), 177
[id](#) (*ema_workbench.em_framework.points.Policy* attribute), 175

[id](#) (*ema_workbench.em_framework.points.Scenario* attribute), 175
[index_for_cat\(\)](#) (*ema_workbench.em_framework.parameters.CategoricalParameter* method), 151
[initialize\(\)](#) (*ema_workbench.em_framework.evaluators.IpyparallelEvaluator* method), 160
[initialize\(\)](#) (*ema_workbench.em_framework.evaluators.MultiprocessingEvaluator* method), 160
[initialize\(\)](#) (*ema_workbench.em_framework.evaluators.SequentialEvaluator* method), 160
[initialized\(\)](#) (*ema_workbench.em_framework.model.AbstractModel* method), 149
[inspect\(\)](#) (*ema_workbench.analysis.logistic_regression.Logit* method), 186
[inspect\(\)](#) (*ema_workbench.analysis.prim.PrimBox* method), 181
[IntegerParameter](#) (class in *ema_workbench.em_framework.parameters*), 151
[invert_axis\(\)](#) (*ema_workbench.analysis.parcoords.ParallelAxes* method), 200

[InvertedGenerationalDistanceMetric](#) (class in *ema_workbench.em_framework.optimization*), 163

[IpyparallelEvaluator](#) (class in *ema_workbench.em_framework.evaluators*), 160

K

[KDE](#) (*ema_workbench.analysis.plotting_util.Density* attribute), 202
[kde_over_time\(\)](#) (in module *ema_workbench.analysis.plotting*), 194
[kind](#) (*ema_workbench.em_framework.outcomes.AbstractOutcome* attribute), 154
[kind](#) (*ema_workbench.em_framework.outcomes.ArrayOutcome* attribute), 155
[kind](#) (*ema_workbench.em_framework.outcomes.ScalarOutcome* attribute), 157
[kind](#) (*ema_workbench.em_framework.outcomes.TimeSeriesOutcome* attribute), 158

L

[legend\(\)](#) (*ema_workbench.analysis.parcoords.ParallelAxes* method), 200
[LegendEnum](#) (class in *ema_workbench.analysis.plotting_util*), 202
[levers](#) (*ema_workbench.em_framework.model.AbstractModel* attribute), 148
[LHSSampler](#) (class in *ema_workbench.em_framework.samplers*), 169
[limits](#) (*ema_workbench.analysis.parcoords.ParallelAxes* attribute), 199

LINES (*ema_workbench.analysis.plotting_util.PlotType* attribute), 202
 lines() (in module *ema_workbench.analysis.plotting*), 195
 load_archives() (*ema_workbench.em_framework.optimization* class method), 162
 load_results() (in module *ema_workbench.util.utilities*), 203
 log_func (*ema_workbench.em_framework.util.ProgressTracking* attribute), 177
 log_progress (*ema_workbench.em_framework.util.ProgressTracking* attribute), 177
 log_to_stderr() (in module *ema_workbench.util.ema_logging*), 203
 Logit (class in *ema_workbench.analysis.logistic_regression*), 185
M
 mass (*ema_workbench.analysis.prim.PrimBox* attribute), 180
 mean (*ema_workbench.analysis.prim.PrimBox* attribute), 180
 merge_results() (in module *ema_workbench.util.utilities*), 203
 Model (class in *ema_workbench.em_framework.model*), 150
 model_init() (*ema_workbench.em_framework.model.AbstractModel* method), 149
 model_kwargs (*ema_workbench.em_framework.experiment_runner.ExperimentRunner* attribute), 172
 model_name (*ema_workbench.em_framework.points.Experiment* attribute), 174
 models (*ema_workbench.analysis.logistic_regression.Logit* attribute), 186
 module
 ema_workbench.analysis.b_and_w_plotting, 201
 ema_workbench.analysis.cart, 183
 ema_workbench.analysis.clusterer, 185
 ema_workbench.analysis.dimensional_stacking, 187
 ema_workbench.analysis.feature_scoring, 188
 ema_workbench.analysis.logistic_regression, 185
 ema_workbench.analysis.pairs_plotting, 197
 ema_workbench.analysis.parcoords, 199
 ema_workbench.analysis.plotting, 193
 ema_workbench.analysis.plotting_util, 201
 ema_workbench.analysis.prim, 178
 ema_workbench.analysis.regional_sa, 188
 ema_workbench.analysis.scenario_discovery, 202
 ema_workbench.connectors.excel, 178
 ema_workbench.em_framework.callbacks, 172
 ema_workbench.em_framework.ema_multiprocessing, 176
 ema_workbench.em_framework.evaluators, 160
 ema_workbench.em_framework.experiment_runner, 171
 ema_workbench.em_framework.model, 148
 ema_workbench.em_framework.optimization, 161
 ema_workbench.em_framework.outcomes, 154
 ema_workbench.em_framework.outputspace_exploration, 165
 ema_workbench.em_framework.parameters, 150
 ema_workbench.em_framework.points, 174
 ema_workbench.em_framework.salib_samplers, 171
 ema_workbench.em_framework.samplers, 167
 ema_workbench.em_framework.util, 177
 ema_workbench.util.ema_exceptions, 202
 ema_workbench.util.ema_logging, 203
 ema_workbench.util.utilities, 203
 MonteCarloSampler (class in *ema_workbench.em_framework.samplers*), 169
 MorrisSampler (class in *ema_workbench.em_framework.salib_samplers*), 171
 multi_initialization (*ema_workbench.em_framework.experiment_runner.ExperimentRunner* attribute), 171
 msis (*ema_workbench.em_framework.experiment_runner.ExperimentRunner* attribute), 171
 multiple_densities() (in module *ema_workbench.analysis.plotting*), 196
 MultiprocessingEvaluator (class in *ema_workbench.em_framework.evaluators*), 160
N
 name (*ema_workbench.em_framework.model.AbstractModel* attribute), 149
 name (*ema_workbench.em_framework.outcomes.AbstractOutcome* attribute), 154
 name (*ema_workbench.em_framework.outcomes.ArrayOutcome* attribute), 155
 name (*ema_workbench.em_framework.outcomes.Constraint* attribute), 156
 name (*ema_workbench.em_framework.outcomes.ScalarOutcome* attribute), 157
 name (*ema_workbench.em_framework.outcomes.TimeSeriesOutcome* attribute), 158

[name \(ema_workbench.em_framework.points.Experiment attribute\), 174](#)
[name \(ema_workbench.em_framework.points.Policy attribute\), 175](#)
[name \(ema_workbench.em_framework.points.Scenario attribute\), 175](#)
[NamedDict \(class in ema_workbench.em_framework.util\), 177](#)
[nr_experiments \(ema_workbench.em_framework.callbacks.AbstractCallback attribute\), 173](#)

O

[OperatorProbabilities \(class in ema_workbench.em_framework.optimization\), 164](#)
[optimize\(\) \(in module ema_workbench.em_framework.evaluators\), 160](#)
[outcome_names \(ema_workbench.em_framework.outcomes.Constraint attribute\), 156](#)
[outcomes \(ema_workbench.em_framework.callbacks.AbstractCallback attribute\), 173](#)
[outcomes \(ema_workbench.em_framework.model.AbstractModel attribute\), 148](#)
[output \(ema_workbench.em_framework.model.AbstractModel attribute\), 149](#)
[OutputSpaceExploration \(class in ema_workbench.em_framework.outputspace_exploration\), 166](#)

P

[pairs_density\(\) \(in module ema_workbench.analysis.pairs_plotting\), 197](#)
[pairs_lines\(\) \(in module ema_workbench.analysis.pairs_plotting\), 198](#)
[pairs_scatter\(\) \(in module ema_workbench.analysis.pairs_plotting\), 198](#)
[ParallelAxes \(class in ema_workbench.analysis.parcoords\), 199](#)
[Parameter \(class in ema_workbench.em_framework.parameters\), 152](#)
[parameter_names \(ema_workbench.em_framework.outcomes.Constraint attribute\), 156](#)
[parameters \(ema_workbench.em_framework.callbacks.AbstractCallback attribute\), 173](#)
[parameters_from_csv\(\) \(in module ema_workbench.em_framework.parameters\), 153](#)
[parameters_to_csv\(\) \(in module ema_workbench.em_framework.parameters\), 153](#)
[pbar \(ema_workbench.em_framework.util.ProgressTrackingMixin attribute\), 177](#)
[pca_preprocess\(\) \(in module ema_workbench.analysis.prim\), 182](#)
[peeling_trajectory \(ema_workbench.analysis.logistic_regression.Logit attribute\), 186](#)
[peeling_trajectory \(ema_workbench.analysis.prim.PrimBox attribute\), 180](#)
[perform_experiments\(\) \(in module ema_workbench.em_framework.evaluators\), 161](#)
[plot\(\) \(ema_workbench.analysis.parcoords.ParallelAxes method\), 200](#)
[plot_cdfs\(\) \(in module ema_workbench.analysis.regional_sa\), 188](#)
[plot_dendrogram\(\) \(in module ema_workbench.analysis.clusterer\), 185](#)
[plot_pairwise_scatter\(\) \(ema_workbench.analysis.logistic_regression.Logit method\), 186](#)
[PlotType \(class in ema_workbench.analysis.plotting_util\), 202](#)
[Point \(class in ema_workbench.em_framework.points\), 175](#)
[Policy \(class in ema_workbench.em_framework.points\), 175](#)
[policy \(ema_workbench.em_framework.points.Experiment attribute\), 174](#)
[Prim \(class in ema_workbench.analysis.prim\), 179](#)
[PrimBox \(class in ema_workbench.analysis.prim\), 180](#)
[PRIMObjectiveFunctions \(class in ema_workbench.analysis.prim\), 178](#)
[Problem \(class in ema_workbench.em_framework.optimization\), 164](#)
[process_replications\(\) \(in module ema_workbench.util.utilities\), 204](#)
[ProgressTrackingMixin \(class in ema_workbench.em_framework.util\), 177](#)

R

[RealParameter \(class in ema_workbench.em_framework.parameters\), 152](#)
[rebuild_platypus_population\(\) \(in module ema_workbench.em_framework.optimization\), 165](#)
[recording \(ema_workbench.analysis.parcoords.ParallelAxes attribute\), 199](#)
[REGRESSION \(ema_workbench.analysis.scenario_discovery_util.RuleInduction attribute\), 202](#)
[Replicator \(class in ema_workbench.em_framework.model\), 150](#)
[ReplicatorModel \(class in ema_workbench.em_framework.model\), 150](#)

`reporting_interval` (`ema_workbench.em_framework.callbacks.ResultsCallback` (in module `ema_workbench.util.utilities`), 204
 attribute), 173
`reporting_interval` (`ema_workbench.em_framework.util.ScalarOutcomeMixin` (class in `ema_workbench.em_framework.outcomes`),
 attribute), 177
`representation`() (in module 157
`ema_workbench.em_framework.util`), 178
`res_dim` (`ema_workbench.analysis.logistic_regression.Logit` 175
 attribute), 186
`res_dim` (`ema_workbench.analysis.prim.PrimBox` 175
 attribute), 180
`resample`() (`ema_workbench.analysis.prim.PrimBox`
 method), 181
`reset_model`() (`ema_workbench.em_framework.model.AbstractModel` (in `ema_workbench.em_framework.evaluators`),
 method), 149
`retrieve_output`() (`ema_workbench.em_framework.model.AbstractModel` (in module
 method), 149
`ema_workbench.analysis.b_and_w_plotting`),
`RobustProblem` (class in 201
`ema_workbench.em_framework.optimization`), 164
`RuleInductionType` (class in `setup_cart`() (in module
`ema_workbench.analysis.scenario_discovery_util`), `ema_workbench.analysis.cart`), 184
202
`run`() (`ema_workbench.analysis.logistic_regression.Logit` `setup_prim`() (in module
 method), 186
`ema_workbench.analysis.prim`), 183
`run_constrained_prim`() (in module `setup_working_directories`() (in module
`ema_workbench.analysis.prim`), 183
`run_experiment`() (`ema_workbench.em_framework.experiments.ExperimentRunner` (in module
 method), 172
`ema_workbench.em_framework.ema_multiprocessing`),
`run_model`() (`ema_workbench.em_framework.model.AbstractModel` (in module 176
 method), 149
`run_model`() (`ema_workbench.em_framework.model.Replication` (in module
 method), 150
`run_model`() (`ema_workbench.em_framework.model.SingleReplication` (in module
 method), 150
`ema_workbench.analysis.prim.PrimBox`
 method), 181
S `show_ppt`() (`ema_workbench.analysis.prim.PrimBox`
 method), 182
`sample`() (`ema_workbench.em_framework.samplers.AbstractSampler` (in module
 method), 167
`show_threshold_tradeoff`()
`sample`() (`ema_workbench.em_framework.samplers.LHSSampler` (in module
 method), 169
`ema_workbench.analysis.logistic_regression.Logit`
 method), 187
`sample`() (`ema_workbench.em_framework.samplers.MonteCarloSampler` (in module
 method), 169
`show_tradeoff`() (`ema_workbench.analysis.logistic_regression.Logit`
 method), 187
`sample_levers`() (in module `show_tradeoff`() (`ema_workbench.analysis.prim.PrimBox`
`ema_workbench.em_framework.samplers`),
 method), 182
170
`sample_parameters`() (in module `show_tree`() (`ema_workbench.analysis.cart.CART`
`ema_workbench.em_framework.samplers`),
 method), 184
170
`sample_uncertainties`() (in module `SingleReplication` (class in
`ema_workbench.em_framework.samplers`),
 class in `ema_workbench.em_framework.model`), 150
170
`Samplers` (class in `ema_workbench.em_framework.evaluators`, 171
 class in `ema_workbench.em_framework.salib_samplers`),
 class in `ema_workbench.em_framework.optimization`),
 164
`SpacingMetric` (class in 164

stats (*ema_workbench.analysis.cart.CART* attribute), [184](#)

stats (*ema_workbench.analysis.cart.CART* property), [184](#)

stats (*ema_workbench.analysis.prim.Prim* property), [180](#)

W

write_ppt_to_stdout() (*ema_workbench.analysis.prim.PrimBox* method), [182](#)

T

temporary_filter() (in module *ema_workbench.util.ema_logging*), [203](#)

ticklabels (*ema_workbench.analysis.parcoords.ParallelAxes* attribute), [200](#)

TimeSeriesOutcome (class in *ema_workbench.em_framework.outcomes*), [158](#)

to_disk() (*ema_workbench.em_framework.outcomes.AbstractOutcome* class method), [155](#)

to_disk() (*ema_workbench.em_framework.outcomes.ArrayOutcome* class method), [156](#)

to_disk() (*ema_workbench.em_framework.outcomes.ScalarOutcome* class method), [158](#)

to_disk() (*ema_workbench.em_framework.outcomes.TimeSeriesOutcome* class method), [159](#)

to_problem() (in module *ema_workbench.em_framework.optimization*), [165](#)

to_robust_problem() (in module *ema_workbench.em_framework.optimization*), [165](#)

U

uncertainties (*ema_workbench.em_framework.model.AbstractModel* attribute), [148](#)

UniformLHSSampler (class in *ema_workbench.em_framework.samplers*), [169](#)

update() (*ema_workbench.analysis.logistic_regression.Logit* method), [187](#)

update() (*ema_workbench.analysis.prim.PrimBox* method), [182](#)

V

variable_name (*ema_workbench.em_framework.outcomes.AbstractOutcome* attribute), [154](#)

variable_name (*ema_workbench.em_framework.outcomes.ArrayOutcome* attribute), [155](#)

variable_name (*ema_workbench.em_framework.outcomes.ScalarOutcome* attribute), [157](#)

variable_name (*ema_workbench.em_framework.outcomes.TimeSeriesOutcome* attribute), [159](#)

VIOLIN (*ema_workbench.analysis.plotting_util.Density* attribute), [202](#)