

2022_SEAI_C6

Fuzzy rule-based classifiers using PL_NSGA_II in
the ROC space

Students: Yuli Orozco and Rambod Rahmani
Supervisor: Marco Cococcioni

June 7, 2022

0 Introduction

The project focused on porting the Python implementation of SK-MOEFS [5] (acronym of SciKit-Multi Objective Evolutionary Fuzzy System) to Julia. SK-MOEFS is a Python library for designing accurate and explainable fuzzy models. Fuzzy Rule-Based Systems (FRBSs) are recognized world-wide as transparent and interpretable tools: they can provide explanations in terms of linguistic rules. Moreover, FRBSs may achieve accuracy comparable to those achieved by less transparent models, such as neural networks and statistical models. SK-MOEFS, is a new Python library that allows the user to easily and quickly design FRBSs, employing Multi-Objective Evolutionary Algorithms. Indeed, a set of FRBSs, characterized by different trade-offs between their accuracy and their explainability, can be generated by SK-MOEFS. The user, then, will be able to select the most suitable model for his/her specific application.

The current version of SK-MOEFS includes an implementation of a specific MOEFS, namely PAES-RCS, introduced in [1]. PAES-RCS selects a reduced number of rules and conditions, from an initial set of rules, during the multiobjective evolutionary learning process. Precisely, they implemented PAES-RCS-FDT, which adopts a fuzzy decision tree (FDT) for generating the initial set of rules [4]. Then, a MOEA is in charge of carrying out the learning process, which stops when a specific condition is reached (for example, when the algorithm reaches the maximum number of fitness function evaluations). Finally, it returns an approximated Pareto front of FRBSs, which are sorted by an ascending order per accuracy. The first model, labeled as the FIRST solution, is the one characterized by the highest accuracy and by the lowest explainability. On the contrary, we marked the model with the highest explainability but the lowest accuracy as the LAST solution.

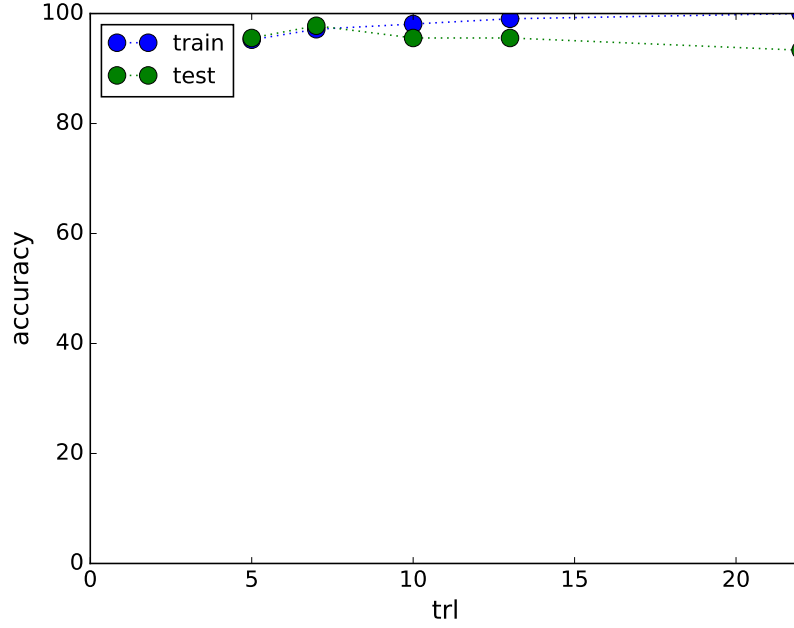


Figure 1: Pareto Front approximation.

```

1 RULE BASE
2 1: IF PetalLength is VL AND PetalWidth is L THEN Class is 0
3 2: IF SepalLength is L AND PetalLength is M AND PetalWidth is M THEN Class is 1
4 3: IF SepalLength is M AND PetalLength is M AND PetalWidth is M THEN Class is 1
5 4: IF PetalLength is VH AND PetalWidth is M THEN Class is 2
6 5: IF PetalWidth is VH THEN Class is 2
7 6: IF SepalLength is M AND SepalWidth is M AND PetalLength is M AND PetalWidth is
   H THEN Class is 1
8 7: IF SepalLength is M AND PetalLength is H AND PetalWidth is H THEN Class is 2

```

Listing 1: First solution.

```

1 RULE BASE
2 1: IF PetalLength is VL AND PetalWidth is L THEN Class is 0
3 2: IF PetalLength is L AND PetalWidth is M THEN Class is 1
4 3: IF PetalLength is VH AND PetalWidth is M THEN Class is 2
5 4: IF PetalWidth is VH THEN Class is 2

```

Listing 2: Median solution.

```

1 RULE BASE
2 1: IF PetalLength is VL AND PetalWidth is L THEN Class is 0
3 2: IF PetalLength is L AND PetalWidth is M THEN Class is 1
4 3: IF PetalWidth is VH THEN Class is 2

```

Listing 3: Last solution.

The Julia Programming Language was designed from the beginning for high performance. Julia programs compile to efficient native code for multiple platforms via LLVM. Julia is dynamically typed, feels like a scripting language, and has good support for interactive use.¹

0.1 Why Julia?

The Julia programming language [3] is a relatively new language, first released in 2012, and aims to be both easy and fast. It "runs like C but reads like Python" [7]. It was made for scientific computing, capable of handling large amounts of data and computation while still being fairly easy to manipulate, create, and prototype code.

The creators of Julia explained why they created Julia in a 2012 blogpost². They said:

We are greedy: we want more. We want a language that's open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that's homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled.

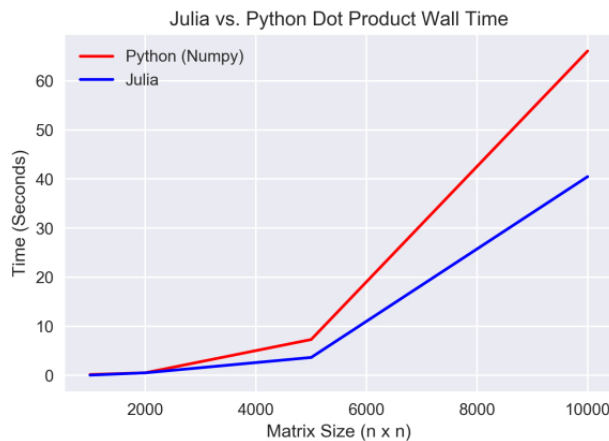


Figure 2: Pareto Front approximation.

¹<https://julialang.org/>

²<https://julialang.org/blog/2012/02/why-we-created-julia/>

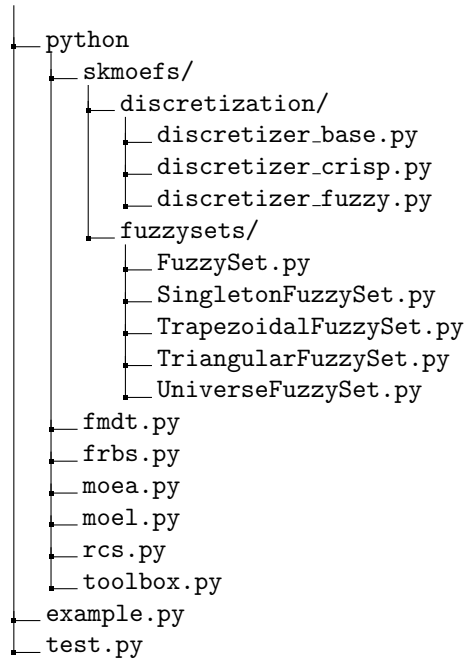
As a simple test to compare scalability of Julia and Python, Figure 2 shows the computation of the dot product of increasingly large matrices using the base Julia language and Numpy in Python (both running in Jupyter Notebooks on a Windows desktop).

Most users are attracted to Julia because of the superior speed. After all, Julia is a member of a prestigious and exclusive club. The petaflop club is comprised of languages who can exceed speeds of one petaflop per second at peak performance. Currently only C, C++, Fortran, and Julia belong to the petaflop club [6].

1 Source Code Porting

This section provides a matching between the Python classes of the original source code and the corresponding Julia implementations. Given that porting Python code to Julia in a rather straightforward procedure (for the most of it), only the most noteworthy cases are documented in details.

The structure of the Python library is depicted in the following:



A fork was created starting from the original implementation³ and can be found in this repository: <https://github.com/rambodrahmani/skmoefs>.

³<https://github.com/GionatanG/skmoefs>

1.1 Datasets

Before getting into the code, it should be pointed out that the ported source code was tested on many different provided benchmark datasets. The repository contains a dedicated `dataset` directory with files in the `.dat` format:

```
dataset/
├── appendicitis.dat
├── bupa.dat
├── glass.dat
├── haberman.dat
├── hayes-roth.dat
├── heart.dat
├── ionosphere.dat
├── iris.dat
└── ...
```

As an example of content, consider the following:

```
1 @relation iris
2 @attribute SepalLength real [4.3, 7.9]
3 @attribute SepalWidth real [2.0, 4.4]
4 @attribute PetalLength real [1.0, 6.9]
5 @attribute PetalWidth real [0.1, 2.5]
6 @attribute Class {0, 1, 2}
7 @inputs SepalLength, SepalWidth, PetalLength, PetalWidth
8 @outputs Class
9 @data
10 5.1, 3.5, 1.4, 0.2, 0
11 4.9, 3.0, 1.4, 0.2, 0
12 4.6, 3.1, 1.5, 0.2, 0
```

Listing 4: Snippet from `iris.dat`.

The custom function `function load_dataset(name::String)`, implemented in `toolbox.jl`, is in charge of parsing such `.dat` files and returning

- `X::Matrix{Float64}`: representing the input data samples;
- `y::Array{Int64}`: representing the class labels
- `attributes::Array{Array{Float64}}`: range of values in the format `[min, max]` for each feature;
- `inputs::Array{String}`: names of the input features;
- `outputs::Array{String}`: names of the output features;

1.2 Classes

Julia does not have classes in the object-oriented sense⁴. Instead we define new types and then define methods on those types. The closest analogy to a Python

⁴<https://discourse.julialang.org/t/is-it-reasonable-to-mimic-a-python-class-with-mutable-structs/57516>

class is a `mutable struct`.

Actually, there is no need to mimic python class, because it already exists in Julia. In a sense, python classes are subset of Julia capabilities. The following two implementations provide an example of the pattern that was used to port Python classes to Julia:

```
1 class PyClass:
2     def __init__(self):
3         self.a = 1.0
4         self.b = 2.0
5
6     def method1(self):
7         return self.a + self.b
8
9 mim = PyClass()
10 mim.method1()
```

Listing 5: Python Class.

```
1 mutable struct Pyclass
2     a::Float
3     b::Float
4 end
5
6 PyClass() = PyClass(0.0, 0.0)
7
8 function __init__(self::PyClass)
9     self.a = 1.0
10    self.b = 2.0
11 end
12
13 function method1(self::PyClass)
14     return self.a + self.b
15 end
16
17 mim = __init__(PyClass())
18 method1(mim)
```

Listing 6: Julia Porting.

If one looks closely, there is almost no difference between Python and Julia definitions. The only observable difference is slight change of syntax, but in a sense, Julia is more consistent.

In python you have

```
1 definition: method - class instance - arguments
2 usage: class instance - dot - method - arguments
```

In Julia you have

```
1 definition: method - struct instance - arguments
2 usage: method - struct instance - arguments
```

The same pattern was used for Abstract Base Classes (ABCs) as well.

1.3 Text I/O

The `__str__` method in Python represents the class objects as a string – it can be used for classes. The `__str__` method should be defined in a way that is easy to read and outputs all the members of the class. This method is also used as a debugging tool when the members of a class need to be checked.

```
1 def __str__(self):
2     return "a=%f, b=%f, c=%f" % (self.a, self.b, self.c)
```

Listing 7: Snippet from `TriangularFuzzySet.py`.

In Julia `show(io::IO = stdout, x)` is used to write a text representation of a value `x` to the output stream `io`. New types `T` should overload `show(io::IO, x::T)`. The representation used by `show` generally includes Julia-specific formatting and type information, and should be parseable Julia code when possible⁵:

```
1 show(io::IO, self::TriangularFuzzySet) = print(io,
2     "a=$(self.a), b=$(self.b), c=$(self.c)"
3 )
```

Listing 8: Snippet from `TriangularFuzzySet.jl`.

1.4 Logging

In Python, the `logging` module defines functions and classes which implement a flexible event logging system for applications and libraries. Loggers should always be instantiated through the module-level function `logging.getLogger(name)`. The threshold for the logger is set to `level` using `setLevel(level)`. Finally, different calls can be used to log messages with different logging levels, such as `logger.debug()`, `logger.info()`, `logger.warning()`, `logger.error()`, etc...

```
1 logger = logging.getLogger('CrispMDLFilter')
2 logger.setLevel(logging.DEBUG)
3 logger.debug("BUILDING HISTOGRAMS.")
```

Listing 9: Snippet from `discretizer_crisp.py`.

In Julia, the `Logging` module provides a way to record the history and progress of a computation as a log of events. Events are created by inserting a logging statement into the source code. The logging functionalities are implemented by means of the macros `@debug`, `@info`, `@warn` and `@error`⁶:

```
1 global_logger(ConsoleLogger(stdout, Logging.Debug))
2 @debug "Building histograms."
```

Listing 10: Snippet from `discretizer_crisp.jl`.

1.5 PyCall

The `PyCall` package provides the ability to directly call and fully interoperate with Python from the Julia language. It allows to import arbitrary Python modules from Julia, call Python functions (with automatic conversion of types between Julia and Python), define Python classes from Julia methods, and share large data structures between Julia and Python without copying them⁷.

The following example demonstrates how `PyCall` allows to use the `Platypus` Python package directly in Julia⁸:

⁵<https://docs.julialang.org/en/v1/base/io-network/#Text-I/O>

⁶<https://docs.julialang.org/en/v1/stdlib/Logging/>

⁷<https://github.com/JuliaPy/PyCall.jl>

⁸Platypus is a framework for evolutionary computing in Python with a focus on multi-objective evolutionary algorithms (MOEAs). <https://platypus.readthedocs.io/en/latest/>

```

1 using PyCall
2 platypus = pyimport("platypus")
3
4 mutable struct NSGAIIS
5     """
6     Extended version of NSGA2 algorithm with added support for
7     snapshots.
8     """
9     snapshots::Array{Any}
10    algorithm::PyObject
11 end
12 NSGAIIS() = NSGAIIS([], platypus.algorithms.NSGAII)
13
14 function initialize(self::NSGAIIS)
15     self.snapshots = []
16     algorithm = platypus.algorithms.NSGAII()
17     algorithm.initialize()
18
19     return self
20 end
21
22 function iterate(self::NSGAIIS)
23     if (self.algorithm.nfe % 100) == 0
24         println("Fitness evaluations " * self.algorithm.nfe)
25     end
26     if length(self.snapshots) < length(milestones) &&
27         self.algorithm.nfe >= milestones[length(self.snapshots)]
28         print("new milestone at " * string(self.algorithm.nfe))
29         append!(self.snapshots, self.algorithm.archive)
30     end
31     self.algorithm.iterate()
32 end

```

Listing 11: Snippet from moea.jl.

In order to be able to work with local Python packages (such as `skmoefs`), its path must be appended to `sys.path`⁹:

```

1 using PyCall
2
3 # add path for importing local skmoefs package
4 pushfirst!(PyVector(pyimport("sys")["path"]), "python/")
5
6 # import python modules
7 skmoefs_py_toolbox = pyimport("skmoefs.toolbox")
8 skmoefs_py_rcs = pyimport("skmoefs.rcs")
9
10 function test1(seed::Int64)
11     set_rng_seed(seed)
12
13     X, y, attributes, inputs, outputs = load_dataset("newthyroid")
14     X_n, y_n = normalize(X, y, attributes)
15     Xtr, Xte, ytr, yte = train_test_split(X_n, y_n, test_size=0.3)
16

```

⁹<https://github.com/JuliaPy/PyCall.jl/issues/48>


```

17 my_moefs = skmoefs_py_toolbox.MPAES_RCS(capacity=32, variator=
    skmoefs_py_rcs.RCSVariator(), initializer=skmoefs_py_rcs.
    RCSInitializer())
18 my_moefs.fit(Xtr, ytr, max_evals=1000)
19
20 my_moefs.show_pareto()
21 my_moefs.show_pareto(Xte, yte)
22 my_moefs.show_model("median", inputs=inputs, outputs=outputs)
23 end

```

Listing 12: Snippet from `example.jl`.

The lines 13-15 where the dataset is loaded, normalized and then split into train and test splits, and line 17 give us a clear example of the automatic conversion of types between Julia and Python provided by `PyCall.jl`.

1.6 Julia Data format (JLD)

Once a set of FRBSs, characterized by different trade-offs between their accuracy and their explainability, is generated, SK-MOEFS allows the user to store the final result relying on the `pickle` module as far as it concerns the Python implementation.

In Julia, the Julia Data format (JLD) was used. JLD, for which files conventionally have the extension `.jld`, is a widely-used format for data storage with the Julia programming language. JLD is a specific "dialect" of HDF5, a cross-platform, multi-language data storage format most frequently used for scientific data. By comparison with "plain" HDF5, JLD files automatically add attributes and naming conventions to preserve type information for each object¹⁰.

```

1 function is_object_present(name)
2     """
3     Check if file exists on the filesystem.
4     """
5     return isfile(name * ".obj")
6 end
7
8 function store_object(filename::String, name::String, object::Any)
9     """
10    Save object as Julia Data format (JLD).
11    """
12    save(filename * ".jld", name, object)
13 end
14
15 function load_object(filename::String)
16     """
17    Load JLD file to object.
18    """
19    return load(filename * ".jld")
20 end

```

Listing 13: Snippet from `toolbox.jl`

¹⁰<https://github.com/JuliaIO/JLD.jl>

1.7 ScikitLearn.jl

The SK-MOEFS Python library was developed under the Scikit-Learn environment [8]. The latter is an Open Source toolbox that provides state-of-the-art implementations of many well-known ML algorithms. SK-MOEFS was designed according to Scikit-Learn’s design principles. Indeed, available data structures and methods in the Scikit-Learn library were exploited. As a result, the user is allowed, under the same framework, to easily and quickly design, evaluate, and use several ML models, including MOEFSs.

ScikitLearn.jl implements the popular `scikit-learn` interface and algorithms in Julia. It supports both models from the Julia ecosystem and those of the `scikit-learn` Python library (via `PyCall.jl`).

Among the many available ones, the train and test split one was used:

```
1 using ScikitLearn.CrossValidation: train_test_split
2
3 set_rng_seed(seed)
4 X, y, attributes, inputs, outputs = load_dataset("newthyroid")
5 X_n, y_n = normalize(X, y, attributes)
6 Xtr, Xte, ytr, yte = train_test_split(X_n, y_n, test_size=0.3)
```

Listing 14: Snippet from `example.jl`.

1.8 porting.jl

This file contains the implementation of Python methods for which a corresponding Julia method is not available.

1.8.1 `numpy.isscalar()`

This `numpy` method returns `True` if the type of element is a scalar type. A scalar is an element of a field which is used to define a vector space. That is to say, you need to define a vector space, based on a field, before you can determine if something is, or is not a scalar (relative to that vector space). For the right vector space, tuples could be a scalar. Of course we are not looking for a mathematically rigorous definition. Just a pragmatic one. The only meaningful way in which a scalar can be defined in Julia, is of the behavior of broadcast. As of Julia 1¹¹:

```
1 isscalar(x::T) where T = isscalar(T)
2 isscalar(::Type{T}) where T = BroadcastStyle(T)
3                               isa Broadcast.DefaultArrayStyle{0}
```

Listing 15: Julia implementation of `numpy.isscalar`.

`Broadcast.DefaultArrayStyleN()` is a `BroadcastStyle` indicating that an object behaves as an N -dimensional array for broadcasting.

¹¹<https://stackoverflow.com/questions/47762777/how-to-check-if-a-variable-is-scalar-in-julia>

1.8.2 `numpy.nan_to_num()`

This `numpy` method replaces `NaN` with zero and `infinity` with large finite numbers. An equivalent Julia implementation is provided by the following¹²:

```
1 nan_to_num(v::Float64) = map(x -> isnan(x) ? zero(x) :  
2      isinf(x) ? prevfloat(typemax(Float64)) : x, v)
```

Listing 16: Julia implementation of `numpy.nan_to_num`.

1.9 discretization

The FRBS design process aims: i) to determine the optimal set of rules for managing regression or classification problems, and ii) to find the appropriate number of fuzzy sets for each attribute and their parameters. The objective of the design process is to concurrently maximize the system accuracy and, possibly, the model explainability. The accuracy of an FRBR is usually maximized by means of a minimization process of the estimation error of the output values. On the other hand, the accuracy of an FRBC is usually calculated in terms of percentage of correctly classified patterns. As regards the explainability, when dealing with FRBS we usually talk about their *intepretability*, namely the capability of explaining how predictions have been done, using terms understandable to humans. Thus, the simplicity of the fuzzy inference engine, adopted to deduce conclusions from facts and rules, assumes a special importance. Moreover, the intepretability is strictly related to the *transparency* of the model, namely to the capability of understanding the structure of the model itself. FRBSs can be characterized by a high transparency level, whenever the linguistic RB is composed of a reduced number of rules and conditions and the fuzzy partitions have a good *integrity*.

As stated in the Introduction, in the last decade, MOEAs have been successfully adopted for designing FRBSs by concurrently optimizing both their accuracy and explainability, leading to the so-called MOEFSs. Indeed, MOEAs allow us to approach an optimization process in which two or more conflicting objectives should be optimized at the same time, such as accuracy and explainability of FRBSs. MOEAs return a set of non-dominated solutions, characterized by different trade-offs between the objectives, which represents an approximation of Pareto front. Adopting a Multi-Objective Learning Scheme (MOEL) it is possible to learn the structure of FRBSs using different strategies, such as learning only the RB considering pre-defined fuzzy partitions, optimizing only the fuzzy set parameters, selecting rules and conditions, from an initial set of rules, and learning/selecting rules concurrently with the optimization of the fuzzy set parameters.

In SK-MOEFS the actual implementation of an MOEL scheme for classification

¹²<https://discourse.julialang.org/t/replace-all-nans-with-zeros-in-dataframe/10001>

problems si provided, namely PAES-RCS-FDT [1]. The implemented algorithm adopts the rule and condition selection (RCS) learning scheme [1] for classification problems. In the learning scheme, an initial set of candidate rules must be generated through a heuristic or provided by an expert. In our implementation, the set of candidate rules is generated exploiting the fuzzy multi-way decision trees (FMDT) [9]: each path from the root to a leaf node translates into a rule. Before learning the FMDT, we need to define an initial strong fuzzy partition for each attribute. **The adopted FMDT algorithm embeds a discretization algorithm** that is in charge of generating such partitions.

There are two kinds of discretization: crisp and fuzzy. In crisp discretization the range of a continuous value is split into several intervals. Elements of an interval are considered as equivalent and each interval is handled as a discrete value. There are different methods of crisp discretization. For instance, some of them take into account the length of the interval, or the frequency of the values, while others are entropy-based. In some domains, the crisp discretization shows some counter-intuitive behavior around the thresholds of the intervals: values around the threshold of two adjacent intervals are considered as different but may be they are not so. For this reason, sometimes it is interesting to build a fuzzy discretization from a crisp one [2].

In this subdirectory, the implementation for different types of discretizations is provided:

- `discretizer_base.jl`: crisp equal-width and equal-frequency discretization;
- `discretizer_crisp.jl`: crisp entropy-based discretization;
- `discretizer_fuzzy.jl`: fuzzy discretization.

1.9.1 `discretizer_base.jl`

This implements the base fuzzy discretizer available in the SK-MOEFS library (`discretizer_base.py`). The number of bins must be specified (`numSet`) and the available discretization methods are:

- **uniform**: this discretization method implements the most popular equal width binning (i.e. all bins have equal width, or represent an equal range of the original variable values, no matter how many cases are in each bin);
- **equifreq**: this discretization method is performed by equal frequency binning (i.e. the thresholds of all bins is selected in a way that all bins contain the same number of numerical values).

As an example, this type of discretization applied to the `iris` dataset will produce the following splits:

- **uniform**:

- SepalLength real [4.3, 7.9]: [4.3, 5.19, 6.1, 7.01, 7.9];
 - SepalWidth real [2.0, 4.4]: [2.0, 2.6, 3.2, 3.803, 4.4];
 - PetalLength real [1.0, 6.9]: [1.0, 2.475, 3.95, 5.42501, 6.9];
 - PetalWidth real [0.1, 2.5]: [0.1, 0.7, 1.3, 1.9, 2.5];
- equifreq:
 - SepalLength real [4.3, 7.9]: [4.3, 5.1, 5.8, 6.4, 7.9];
 - SepalWidth real [2.0, 4.4]: [2.0, 2.8, 3.0, 3.3, 4.4];
 - PetalLength real [1.0, 6.9]: [1.0, 1.6, 4.3, 5.1, 6.9];
 - PetalWidth real [0.1, 2.5]: [0.1, 0.3, 1.3, 1.8, 2.5].

No normalization was used in the previous example.

1.9.2 discretizer_crisp.jl

This implements the crisp MDL discretizer available in the SK-MOEFS library (`discretizer_crisp.py`).

This discretizer is entropy based. The candidate splits are obtained by considering the unique elements in the sorted features array. For each feature a histogram is built using the binary histogram function `__simpleHist(self::CrispMDLFilter, e::Float64, fIndex::Int64)`. Finally, *information gain* is used to compute the best split points:

$$Gain(A) = Info(D) - Info_A(D)$$

where

$$Info(D) = - \sum_{i=1}^m p_i \log_2(p_i)$$

and

$$Info_A(D) = \sum_{j=1}^v \frac{|D_j|}{|D|} \times Info(D_j)$$

where D is the input dataset (`data::MatrixFloat64` in `discretizer_crisp.py`), and A is the current considered feature after being discretized by means of histograms.

```

1 function entropy(self::CrispMDLFilter, counts::Array{Float64},
2   totalCount)
3   if totalCount == 0
4     return 0
5   end
6   numClasses = length(counts)
7   impurity = 0.0
8   classIndex = 1
9   while (classIndex <= numClasses)
10    classCount = counts[classIndex]

```

```

11         if classCount != 0
12             freq = classCount / totalCount
13             if freq != 0
14                 impurity -= freq * log2(freq)
15             end
16         end
17         classIndex += 1
18     end
19     return impurity
20 end

```

Listing 17: $Info(D)$ implementation in Julia.

As an example, this type of discretization applied to the `iris` dataset will produce the following cut points:

- SepalLength real [4.3, 7.9]: [5.6, 6.2];
- SepalWidth real [2.0, 4.4]: [3.0, 3.4];
- PetalLength real [1.0, 6.9]: [3.0, 4.8];
- PetalWidth real [0.1, 2.5]: [1.0, 1.8].

No normalization was used in the previous example.

As far as it concerns the source code:

- logging functionalities are used to debug which features are accepted, which ones are rejected and the associated $Gain(A)$;

```

[ Debug: Building histograms.
@ Main ~/DevOps/skmoefs/julia/skmoefs/discretization/discretizer_crisp.jl:83
[ Debug: Feature 1 index 39, gain 0.5572326878069267 ACCEPTED
@ Main ~/DevOps/skmoefs/julia/skmoefs/discretization/discretizer_crisp.jl:196
[ Debug: Feature 1 index 36, gain 0.13788086600590665 REJECTED
@ Main ~/DevOps/skmoefs/julia/skmoefs/discretization/discretizer_crisp.jl:193
[ Debug: Feature 1 index 57, gain 0.15667748943781934 ACCEPTED
@ Main ~/DevOps/skmoefs/julia/skmoefs/discretization/discretizer_crisp.jl:196
[ Debug: Feature 1 index 48, gain 0.07106520930105642 REJECTED
@ Main ~/DevOps/skmoefs/julia/skmoefs/discretization/discretizer_crisp.jl:193
[ Debug: Feature 1 index 84, gain 0.1253941057173097 REJECTED
@ Main ~/DevOps/skmoefs/julia/skmoefs/discretization/discretizer_crisp.jl:193
[ Debug: Feature 2 index 39, gain 0.26791136918926517 ACCEPTED
@ Main ~/DevOps/skmoefs/julia/skmoefs/discretization/discretizer_crisp.jl:196
[ Debug: Feature 2 index 27, gain 0.1422875251231126 ACCEPTED
@ Main ~/DevOps/skmoefs/julia/skmoefs/discretization/discretizer_crisp.jl:196
[ Debug: Feature 2 index 12, gain 0.07349161750856292 REJECTED
@ Main ~/DevOps/skmoefs/julia/skmoefs/discretization/discretizer_crisp.jl:193
[ Debug: Feature 2 index 30, gain 0.021087947737252533 REJECTED
@ Main ~/DevOps/skmoefs/julia/skmoefs/discretization/discretizer_crisp.jl:193
[ Debug: Feature 2 index 42, gain 0.0490325021324306 REJECTED
@ Main ~/DevOps/skmoefs/julia/skmoefs/discretization/discretizer_crisp.jl:193
[ Debug: Feature 3 index 27, gain 0.9182958340544894 ACCEPTED
@ Main ~/DevOps/skmoefs/julia/skmoefs/discretization/discretizer_crisp.jl:196

```

Figure 3: Crisp MDL Discretizer debug logs.

- the `BisectPy.jl`¹³ Julia package, which implements Python `bisect` module, was used;
 - since Julia’s array index starts from 1 but Python starts from 0, the returned index of either `bisect_left` or `bisect_right` is always their Python’s correspondence plus 1; also, the behavior of Python’s `a[:i]` where `a` is an array is also different from Julia: Julia array includes the i -th item but Python does not.
- `floor{Int64, val}` is preferred to `Int64(val)` since the latter might lead to Julia `InexactError`; in general an `InexactError` happens when you try to convert a value to an exact type (like integer types, but unlike floating-point types) in which the original value cannot be exactly represented; other programming languages arbitrarily chose some way of rounding here (often truncation but sometimes rounding to nearest). Julia doesn’t guess and requires you to be explicit;
- `numpy.linspace(start, stop, num, ...)` is replaced by `LinRange(start, stop, len)` which return a range with `len` linearly spaced elements between its start and stop;
- `numpy.unique(ar, ...)` is replaced by `unique(itr)` which returns an array containing only the unique elements of collection `itr`.

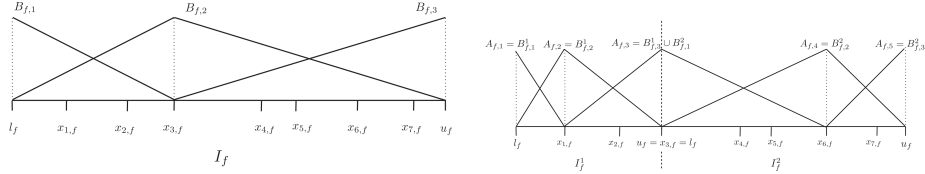
1.9.3 discretizer.fuzzy.jl

This implements the fuzzy MDL discretizer available in the SK-MOEFS library (`discretizer.fuzzy.py`). A strong fuzzy partition is determined on each continuous attribute by using a novel discretizer based on the fuzzy entropy. [10] Partitioning of continuous attributes is a crucial aspect in the generation of fuzzy decision trees (FDTs) and, therefore, should be performed carefully. An interesting study proposed has investigated 111 different approaches for generating fuzzy partitions and has analyzed how these approaches can influence the accuracy and the complexity (in terms of number of nodes) of the generated FDTs. Among them, fuzzy partitioning based on fuzzy entropy (FPFE) has proved to be very effective. The SK-MOEFS library comes equipped with an FPFE for generating strong triangular fuzzy partitions. The original algorithm was implemented in order to be able to work with big data as well [10].

The proposed FPFE is a recursive supervised method, which generates candidate fuzzy partitions and evaluates these partitions employing the fuzzy entropy. The algorithm selects the candidate fuzzy partition that minimizes the fuzzy entropy and then splits the continuous attribute domain into two subsets. Similar to the entropy minimization method proposed by Fayyad and Irani, the process is repeated for each generated subset until a stopping condition is met. The candidate fuzzy partitions are generated for each value of the attribute in the training set: the values are sorted in increasing order.

¹³<https://github.com/singularitti/BisectPy.jl>

Also in this case, the candidate splits are obtained by considering the unique elements in the sorted features array. The number of bins is an additional parameter in this case and it is taken into account when computing the candidate splits. For each feature a histogram is built using the binary histogram function `simpleHist(self::FuzzyMDLFilter, e::Float64, fIndex::Int64)`. Finally, *fuzzy entropy* is used to compute the best split points:



(a) Example of fuzzy partition defined on $x_{3,f}$ (b) Example of application of the recursive procedure to the fuzzy partition defined on $x_{3,f}$

Fig. (b) shows an example of application of the recursive procedure to the fuzzy partition shown in Fig. (a). We can observe that the partitioning of both I_f^1 and I_f^2 generates three fuzzy sets in both $[l_f, x_{i,f}^0]$ and $(x_{i,f}^0, u_f]$. Actually, the two fuzzy sets, which have the core in $x_{i,f}^0$, are fused for generating a unique fuzzy set. Thus, the resulting partition is a strong partition with five fuzzy sets. This fusion can be applied at each level of the recursion. The final result is a strong fuzzy partition $P_f = \{A_{f,1}, \dots, A_{f,T_f}\}$ on U_f , where $A_{f,j}$, with $j = 1, \dots, T_f$, is the j -th triangular fuzzy set. The procedure adopted for the fuzzy partition generation is simple, although computationally quite heavy. Furthermore, it generates strong fuzzy partitions, which are widely assumed to have a high interpretability. Finally, it allows performing an attribute selection because it may lead to the elimination of attributes, speeding up the FDT learning process.

As an example, this type of discretization applied to the `iris` dataset will produce the following cut points:

- `SepalLength` real $[4.3, 7.9]$: $[4.3, 5.7, 7.9]$;
- `SepalWidth` real $[2.0, 4.4]$: $[]$;
- `PetalLength` real $[1.0, 6.9]$: $[1.0, 1.9, 4.0, 5.0, 6.9]$;
- `PetalWidth` real $[0.1, 2.5]$: $[0.1, 0.6, 1.3, 1.8, 2.5]$.

No normalization was used in the previous example.

As far as it concerns the source code:

- logging functionalities are used to debug which features are accepted, which ones are rejected;

1.10 fuzzysets

Fuzzy Rule-Based Systems (FRBSs) are a category of models strongly oriented towards explainability. FRBSs are highly interpretable and transparent because of the linguistic definitions of fuzzy rules and fuzzy sets, which represent the knowledge base of these models. Moreover, the simplicity of the reasoning method, adopted for providing a decision based on input facts, ensures also a high explainability level of FRBSs.

A Fuzzy Rule-Based System (FRBS) is characterized by two main components, namely the Knowledge Base (KB) and the fuzzy inference engine. The KB is composed by a set of linguistic rules and by a set of parameters which describe the fuzzy sets on which the rules are defined. The fuzzy inference engine is in charge of generating a prediction, given a new input pattern, based on the content of the KB.

Let $X = \{X_1, \dots, X_F\}$ be the set of input attributes and X_{F+1} be the output attribute. Let U_f , with $f = 1, \dots, F+1$, be the universe of the f -th attribute X_f . Let $P_f = \{A_{f,1}, \dots, A_f, T_f\}$ be a fuzzy partition of T_f fuzzy sets on attribute X_f . Finally, we define the training set $\{(x_1, x_{F+1,1}), \dots, (x_N, x_{F+1,N})\}$ as a collection of N input-output pairs, with $x_t = [x_{t,1}, \dots, x_{t,F}] \in \mathbb{R}, t = 1, \dots, N$. In regression problems, X_{F+1} is a continuous attribute and, therefore, $\forall t \in [0 \dots N], x_{F+1,t} \in \mathbb{R}$. With the aim of estimating the output value corresponding to a given input vector, we can adopt a Fuzzy Rule-Based Regressor (FRBR) with a rule base (RB) composed of M linguistic fuzzy rules expressed as:

$$R_m : \text{IF } X_1 \text{ is } A_{1,j_m,1} \text{ AND } \dots \text{ AND } X_f \text{ is } A_{f,j_m,f} \text{ AND} \\ \dots \text{ AND } X_F \text{ is } A_{F,j_m,F} \text{ THEN } X_{F+1} \text{ is } A_{F+1,j_m,F+1}$$

In classification problems, X_{F+1} is categorical and $x_{F+1,t} \in C$, where $C = \{C_1, \dots, C_K\}$ is the set of K possible classes. With the aim of determining the class of a given input vector, we can adopt a Fuzzy Rule-Based Classifier (FRBC) with an RB composed of M rules expressed as:

$$R_m : \text{IF } X_1 \text{ is } A_{1,j_m,1} \text{ AND } \dots \text{ AND } X_f \text{ is } A_{f,j_m,f} \text{ AND} \\ \dots \text{ AND } X_F \text{ is } A_{F,j_m,F} \text{ THEN } X_{F+1} \text{ is } C_{j_m} \text{ with } RW_m$$

where C_{j_m} is the class label associated with the m^{th} rule, and RW_m is the rule weight, i.e., a certainty degree of the classification in the class C_{j_m} for a pattern belonging to the fuzzy subspace delimited by the antecedent of the rule R_m .

In this subdirectory, the implementation for different types of Fuzzy Sets is provided.

1.10.1 FuzzySet.jl

This file implements the Abstract Base Class `FuzzySet.py`. This provides the definitions for the methods that must be implemented by all fuzzy set implementations:

1. `membershipDegree(self::FuzzySet, x)`: returns the membership degree of input `x` to the fuzzy set `self`;
2. `isInSupport(self::FuzzySet, x)`: returns `true` if the given input `x` is in the support of the given fuzzy set `self`, `false` otherwise;
3. `isFirstOfPartition(self::FuzzySet)`: returns `true` if the given fuzzy set is the first one in an array of fuzzy sets (*lower boundary* = $-\infty$), `false` otherwise;
4. `isLastOfPartition(self::FuzzySet)`: returns `true` if the given fuzzy set is the last one in an array of fuzzy sets (*upper boundary* = ∞), `false` otherwise;

In Julia there is no way of enforcing the implementation of such methods. Effectively, this means asking Julia to do static checks. This is not how the language was designed¹⁴.

1.10.2 UniverseFuzzySet.jl

This file implements the Universe of Discourse (`UniverseFuzzySet.py`). Formally, the definition of a fuzzy set requires two basic components: a universe of discourse or domain and a function, called the membership function, which defines the "degree" to which a particular element of the domain belongs or not belongs to the set.

1.10.3 SingletonFuzzySet.jl

This file implements a fuzzy singleton (`SingletonFuzzySet.py`). A fuzzy singleton is a fuzzy set which support is a single point in universe of discourse.

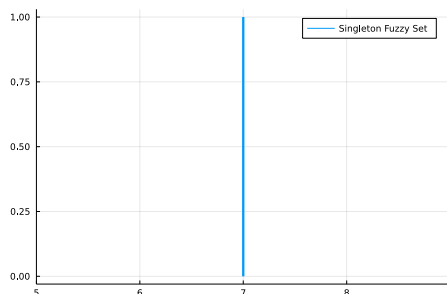


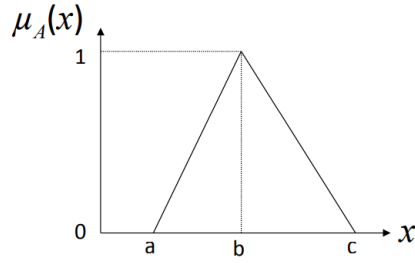
Figure 4: Fuzzy Singleton plot in Julia.

¹⁴<https://discourse.julialang.org/t/forcing-users-to-implements-all-methods-in-an-abstract-interface/23687>

1.10.4 TriangularFuzzySet.jl

This file implements a triangular fuzzy set (`TriangularFuzzySet.py`). A triangular membership function is specified by three parameters a, b, c , which represent the x coordinates of the three vertices of the membership function $\mu(A)$:

- a : lower bound, which membership degree is 0;
- b : center, which membership degree is 1;
- c : upper bound, which membership degree is 0.



(a) Triangular Fuzzy Set

$$\mu_A(x) = \begin{cases} 0 & \text{if } x \leq a \\ \frac{x-a}{b-a} & \text{if } a \leq x \leq b \\ \frac{c-x}{c-b} & \text{if } b \leq x \leq c \\ 0 & \text{if } x \geq c \end{cases}$$

(b) Triangular Fuzzy Set Membership Function

In Julia, one can be instantiated as follows:

```
1 fuzzy_triangular = createTriangularFuzzySet([5.5, 6.3, 8.4])
2 println(fuzzy_triangular)
3 plotTriangularFuzzySet(fuzzy_triangular)
```

Listing 18: Julia implementation of `numpy.isscalar`.

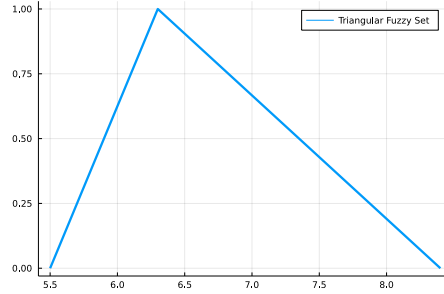
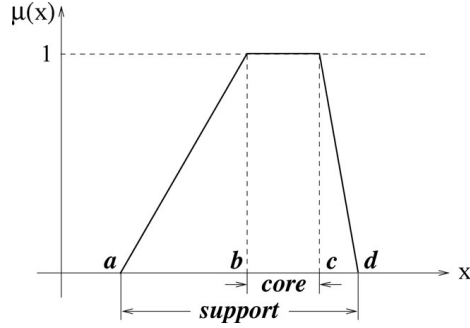


Figure 5: Fuzzy Triangular set plot in Julia.

1.10.5 TrapezoidalFuzzySet.jl

This file implements a trapezoidal fuzzy set (`TrapezoidalFuzzySet.py`). A trapezoidal membership function is specified by four parameters a, b, c, d , which represent the x coordinates of the four vertices of the membership function $\mu(A)$:

- a : lower bound, which membership degree is 0;
- b : left plateau, which membership degree is 1;
- c : right plateau, which membership degree is 1;
- d : upper bound, which membership degree is 0.



(a) Trapezoidal Fuzzy Set

$$\mu_A(x) = \begin{cases} 0 & \text{if } x \leq a \\ \frac{x-a}{b-a} & \text{if } a \leq x \leq b \\ 1 & \text{if } b \leq x \leq c \\ \frac{d-x}{d-c} & \text{if } c \leq x \leq d \\ 0 & \text{if } d \leq x \end{cases}$$

(b) Trapezoidal Fuzzy Set Membership Function

In Julia, one can be instantiated as follows:

```
1 fuzzy_trapezoidal = createTrapezoidalFuzzySet([1.7, 2.8, 3.1, 4.9])
2 println(fuzzy_trapezoidal)
3 plotTrapezoidalFuzzySet(fuzzy_trapezoidal)
```

Listing 19: Julia implementation of `numpy.isscalar`.

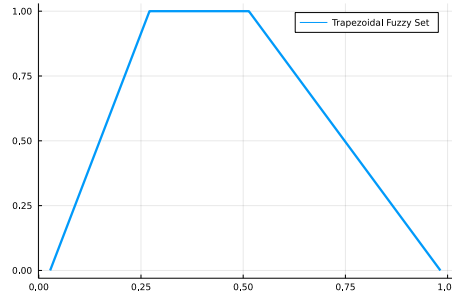


Figure 6: Fuzzy Trapezoidal set plot in Julia.

1.11 Strong triangular fuzzy partitions

In the current version of SK-MOEFES, **strong triangular fuzzy partitions** was adopted. As shown in the figure below, each partition is made up of triangular fuzzy sets $A_{f,j}$, whose membership function can be represented by the tuples

$(a_{f,j}, b_{f,j}, c_{f,j})$, where $a_{f,j}$ and $c_{f,j}$ correspond to the left and right extremes of the support of $A_{f,j}$, and $b_{f,j}$ to its core.

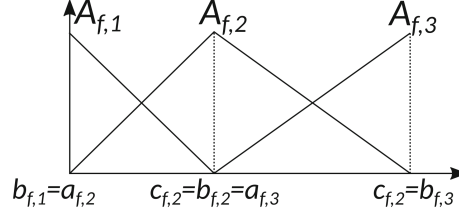


Figure 7: An example of a strong triangular fuzzy partition with three fuzzy sets.

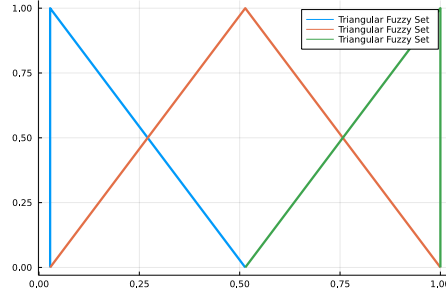


Figure 8: Strong triangular fuzzy partition in Julia.

1.12 moel.jl

We now start getting deeper in the design of the SK-MOEFS library. SK-MOEFS extends the functionalities of Scikit-Learn, a popular Open Source tool for predictive data analysis. It's Julia counter part was used, `ScikitLearn.jl` implements the popular `scikit-learn` interface and algorithms in Julia. It supports both models from the Julia ecosystem and those of the `scikit-learn` Python library (via `PyCall.jl`). Similarly to Scikit-Learn, SK-MOEFS allows also to adopt the generated models for making predictions and evaluating the models in terms of different metrics. However, since SK-MOEFS creates a collection of different FRBSs, data structures and methods were appropriately designed for handling more than one model. Indeed, classically, Scikit-Learn algorithms allow the user to define, train, evaluate, and use just one model.

To design and implement SK-MOEFS, the official Scikit-Learn guidelines for developers were followed:

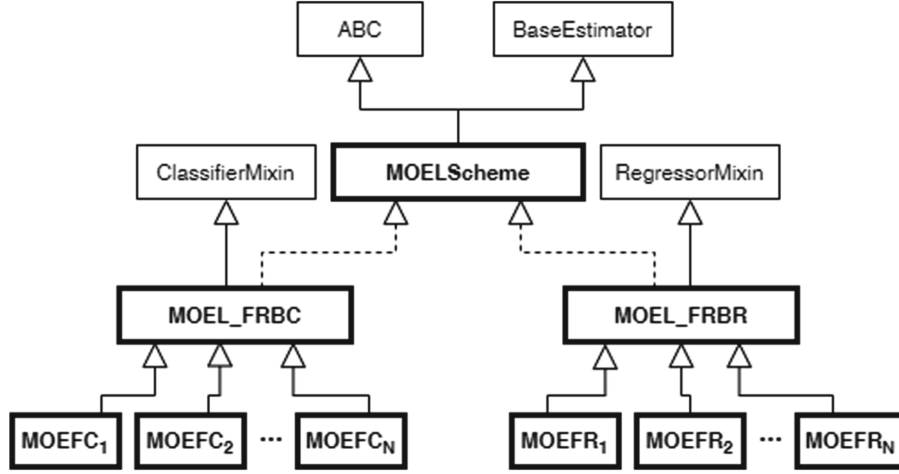


Figure 9: UML class diagram describing the class hierarchy of SK-MOEFs.

A MOELScheme represents a general multiobjective evolutionary learning scheme for generating a set of FRBSs characterized by different trade-offs between accuracy and explainability. We recall that the chromosome coding and the mating operators depend on the selected learning scheme. As regards the fitness functions, the accuracy measure depends on the type of problems to be approached (classification or regression), and the explainability measure can be defined in several ways, as discussed in the previous section. Since the aim was of providing a general scheme for approaching both classification and regression problems by using MOEFs, two abstract classes are derived from the MOELScheme one, namely MOEL FRBC and MOEL FRBR. They define, respectively, the MOEL scheme for Fuzzy Rule-based Classifiers (FRBCs) and the one for Fuzzy Rule-based Regressors (FRBRs). The former includes methods from the ClassifierMixin class and the latter from RegressorMixin class.

Of course, in Julia, there is no such thing as deriving a class. Everything is implemented by means of mutable structs and by adding types to methods.

```

1 mutable struct MOELScheme
2 end

```

Listing 20: Snippet from moel.jl.

With the following methods defined on:

- `fit(self::MOELScheme, X::MatrixFloat64, y::ArrayInt64)`: estimates the model parameters, namely the RB and the fuzzy partitions, exploiting the provided training set; in the beginning, the method initializes a MOEL scheme according to a specific learning strategy and to the type of problem to be handled, namely classification or regression; then, a MOEA

is in charge of carrying out the learning process, which stops when a specific condition is reached (for example, when the algorithm reaches the maximum number of fitness function evaluations); finally, it returns an approximated Pareto front of FRBSs, which are sorted by an ascending order per accuracy; the first model, labeled as the FIRST solution, is the one characterized by the highest accuracy and by the lowest explainability; on the contrary, the model with the highest explainability but the lowest accuracy is marked as the LAST solution; finally, the MEDIAN model is the middle ground between the two; indeed, its accuracy is the median among the solutions;

- `cross_val_score(self::MOELScheme, X::MatrixFloat64, y::ArrayInt64, num_fold::Int64)`: measures the performance of the model using K-Fold cross validation; the cross val score which usually returns an array of k scores, one for each fold, was redefined; here, the method returns a $k \times 6$ matrix, where each row contains the accuracy and the explainability, calculated on the test set, of the FIRST, MEDIAN, and LAST solutions;
- `show_pareto(self::MOELScheme)`: extracts and plots the values of accuracy and explainability; it returns a plot of the approximated Pareto front, both on the training and the test sets;
- `show_model(self::MOELScheme, position)`: given the position of a model in the Pareto front, this method shows the set of fuzzy linguistic rules and the fuzzy partitions associated with each linguistic attribute; the predefined model of choice is, as always, the FIRST solution;
- `__getitem__(self::MOELScheme, position)`: returns the model for which the position in the Pareto front is given.

```
1 mutable struct MOEL_FRBC
2     classifiers::Array{Any}
3 end
```

Listing 21: Snippet from `moel.jl`.

With the following methods defined on:

- `predict(self::MOEL_FRBC, X::MatrixFloat64, position::String="first")`: in charge of predicting the class labels associated with a new set of input patterns; it returns a vector of estimated labels; since the MOEL scheme generates multiple models, the method takes as input also an index for selecting the model into the Pareto front; by default, the function adopts the most accurate model (FIRST) for making predictions;
- `score(self::MOEL_FRBC, X::MatrixFloat64, y::ArrayInt64, sample_weight=nothing)`: generates the values of the accuracy and explainability measures for the selected model.

```

1 mutable struct MOEL_FRBR
2     regressors::Array{Any}
3 end

```

Listing 22: Snippet from `moel.jl`.

With the following methods defined on:

- `predict(self::MOEL_FRBR, X::MatrixFloat64, position::String="first")`: in charge of predicting the values associated with a new set of input patterns; it returns a vector of estimated values; since the MOEL scheme generates multiple models, the method takes as input also an index for selecting the model into the Pareto front; by default, the function adopts the most accurate model (FIRST) for making predictions;
- `score(self::MOEL_FRBR, X::MatrixFloat64, y::ArrayInt64, sample_weight=nothing)`: generates the values of the accuracy and explainability measures for the selected model.

1.13 moea.jl

Once the multi-objective evolutionary learning scheme is chosen, a multiobjective evolutionary algorithm is needed in order to be able to simultaneously optimize both objective functions. This file provides the implementation of different MOEA extended starting from the Python `platypus` package. In Julia, the `PyCall` package must be used. As an example of porting, consider the following:

```

1 using PyCall
2 platypus = pyimport("platypus")
3
4 mutable struct NSGAIIS
5     """
6     Extended version of NSGA3 algorithm with added support for
7     snapshots.
8     """
9     snapshots::Array{Any}
10    algorithm::PyObject
11 end
12 NSGAIIS() = NSGAIIS([], platypus.algorithms.NSGAIII)
13
14 function initialize(self::NSGAIIS)
15     self.snapshots = []
16     algorithm = platypus.algorithms.NSGAIII()
17     algorithm.initialize()
18
19     return self
20 end
21
22 function iterate(self::NSGAIIS)
23     if (self.algorithm.nfe % 100) == 0
24         print("Fitness evaluations " * self.algorithm.nfe)

```



```

25     end
26     if length(self.snapshots) < length(milestones) && self.
algorithm.nfe >= milestones[length(self.snapshots)]
27         print("new milestone at " * string(self.algorithm.nfe))
28         append!(self.snapshots, self.algorithm.population)
29     end
30     self.algorithm.iterate()
31 end
32
33 function createNSGAIIS()
34     return initialize(NSGAIIS())
35 end

```

Listing 23: NSGA3 Implementation in Julia by means of the Python platypus package.

SK-MOEFS comes equipped with an example of an MOEL scheme implementation: PAES-RCS-FDT [4] for classification problems (indeed, an actual implementation of a MOEL_FRBC). The implemented algorithm adopts the rule and condition selection (RCS) learning scheme [1] for classification problems. The multi-objective evolutionary learning scheme is based on the $(2 + 2)$ M-PAES, which is an MOEA successfully employed in the context of MOEFSs during the last years. The algorithm concurrently optimizes two objectives: the first objective considers the TRL as explainability measure; the second objective takes into account the accuracy, assessed in terms of classification rate.

```

1 mutable struct MPAES2_2
2     """
3     M-PEAS(2+2) algorithm implementation.
4     """
5     algorithm::PyObject
6     variator::RCSVariator
7     dominance::PyObject
8     archive::PyObject
9 end
10
11 MPAES2_2() = MPAES2_2(platypus.algorithms.AbstractGeneticAlgorithm,
12                       RCSVariator(),
13                       platypus.ParetoDominance,
14                       platypus.AdaptiveGridArchive)
15
16 function __init__(self::MPAES2_2, problem::RCSPProblem, variator::
17     RCSVariator=nothing, capacity::Int64=32, divisions::Int64=8,
18     generator::MOEAGenerator=MOEAGenerator())
19     """
20     :param problem: the problem object which is responsible of
21     creating and evaluating a solution.
22     :param variator: the genetic operator which can comprises
23     elements of cross-over and mutation
24     :param capacity: the maximum number of solutions that can be
25     held by the archive throughout
26     the evolution
27     :param divisions: the number of divisions for the grid within
28     the archive. This value
29     influences how a new solution will replace an existing one

```

```

24     :param generator: a MOEA generator to initialize the population
25     /archive
26     """
27     self.algorithm.__init__(problem, population_size=2, generator=
28     generator)
29     self.variator = variator
30     self.dominance = platypus.ParetoDominance()
31     self.archive = platypus.AdaptiveGridArchive(capacity, problem.
32     nobjs, divisions)
33     # from time to time we take snapshots of the archive for
34     debugging purposes
35     self.snapshots = []
36 end
37 function createMPAES2_2(problem::RCSPProblem, variator::RCSVariator=
38     nothing, capacity::Int64=32, divisions::Int64=8, generator::
39     MOEAGenerator=MOEAGenerator())
40     return __init__(MPAES2_2(), problem, variator, capacity,
41     divisions, generator)
42 end

```

Listing 24: MPAES2.2 implementation in Julia by means of the Python platypus package.

1.14 fmdt.jl

In the learning scheme, an initial set of candidate rules must be generated through a heuristic or provided by an expert. In our implementation, the set of candidate rules is generated exploiting the fuzzy multi-way decision trees (FMDT) [10]: a distributed FDT learning scheme shaped according to the MapReduce programming model for generating both binary and multiway FDTs from big data. The scheme relies on a novel distributed fuzzy discretizer that generates a strong fuzzy partition for each continuous attribute based on fuzzy information entropy. The fuzzy partitions are, therefore, used as an input to the FDT learning algorithm, which employs fuzzy information gain for selecting the attributes at the decision nodes.

Each path from the root to a leaf node translates into a rule. Before learning the FMDT, we need to define an initial strong fuzzy partition for each attribute. The adopted FMDT algorithm embeds a discretization algorithm that is in charge of generating such partitions.

```

1 Feature: 4 - [a=-Inf, b=0.0, c=0.25] : [1.0, 0.0, 0.0]
2 Feature: 4 - [a=0.0, b=0.25, c=0.5]
3 |   Feature: 3 - [a=-Inf, b=0.0, c=0.25] : [1.0, 0.0, 0.0]
4 |   Feature: 3 - [a=0.0, b=0.25, c=0.5]
5 | |   Feature: 2 - [a=-Inf, b=0.0833, c=0.2917] : [0.0978, 0.9022, 0.0]
6 | |   Feature: 2 - [a=0.0833, b=0.2917, c=0.5] : [0.2344, 0.7656, 0.0]
7 | |   Feature: 2 - [a=0.2917, b=0.5, c=0.7083] : [1.0, 0.0, 0.0]

```

```

8 | | Feature: 2 - [a=0.5, b=0.7083, c=0.9167] : [1.0, 0.0, 0.0]
9 | | Feature: 2 - [a=0.7083, b=0.9167, c=Inf] : [1.0, 0.0, 0.0]
10 | | Feature: 3 - [a=0.25, b=0.5, c=0.75] : [0.0, 1.0, 0.0]
11 | | Feature: 3 - [a=0.5, b=0.75, c=1.0] : [0.0, 1.0, 0.0]
12 | | Feature: 3 - [a=0.75, b=1.0, c=Inf] : [0.0, 0.0, 0.0]
13 | Feature: 4 - [a=0.25, b=0.5, c=0.75]
14 | | Feature: 3 - [a=-Inf, b=0.0, c=0.25] : [0.0, 0.0, 0.0]
15 | | Feature: 3 - [a=0.0, b=0.25, c=0.5] : [0.0, 1.0, 0.0]
16 | | Feature: 3 - [a=0.25, b=0.5, c=0.75]
17 | | | Feature: 1 - [a=-Inf, b=0.0278, c=0.2708] : [0.0, 1.0, 0.0]
18 | | | Feature: 1 - [a=0.0278, b=0.2708, c=0.5139] : [0.0, 0.9912, 0.0088]
19 | | | Feature: 1 - [a=0.2708, b=0.5139, c=0.7569]
20 | | | | Feature: 1 - [a=-Inf, b=0.0278, c=0.2708] : [0.0, 0.0, 0.0]
21 | | | | Feature: 1 - [a=0.0278, b=0.2708, c=0.5139] : [0.0, 0.9773, 0.0227]
22 | | | | Feature: 1 - [a=0.2708, b=0.5139, c=0.7569] : [0.0, 0.914, 0.086]
23 | | | | Feature: 1 - [a=0.5139, b=0.7569, c=1.0] : [0.0, 0.9644, 0.0356]
24 | | | | Feature: 1 - [a=0.7569, b=1.0, c=Inf] : [0.0, 0.0, 0.0]
25 | | | | Feature: 1 - [a=0.5139, b=0.7569, c=1.0] : [0.0, 0.9806, 0.0194]
26 | | | | Feature: 1 - [a=0.7569, b=1.0, c=Inf] : [0.0, 0.0, 0.0]
27 | | Feature: 3 - [a=0.5, b=0.75, c=1.0]
28 | | | Feature: 2 - [a=-Inf, b=0.0833, c=0.2917] : [0.0, 0.4305, 0.5695]
29 | | | Feature: 2 - [a=0.0833, b=0.2917, c=0.5]
30 | | | | Feature: 1 - [a=-Inf, b=0.0278, c=0.2708] : [0.0, 0.0, 0.0]
31 | | | | Feature: 1 - [a=0.0278, b=0.2708, c=0.5139] : [0.0, 0.9114, 0.0886]
32 | | | | Feature: 1 - [a=0.2708, b=0.5139, c=0.7569] : [0.0, 0.5806, 0.4194]
33 | | | | Feature: 1 - [a=0.5139, b=0.7569, c=1.0] : [0.0, 0.7109, 0.2891]
34 | | | | Feature: 1 - [a=0.7569, b=1.0, c=Inf] : [0.0, 0.0, 1.0]
35 | | | Feature: 2 - [a=0.2917, b=0.5, c=0.7083]
36 | | | | Feature: 1 - [a=-Inf, b=0.0278, c=0.2708] : [0.0, 0.0, 0.0]
37 | | | | Feature: 1 - [a=0.0278, b=0.2708, c=0.5139] : [0.0, 0.907, 0.093]
38 | | | | Feature: 1 - [a=0.2708, b=0.5139, c=0.7569] : [0.0, 0.7816, 0.2184]
39 | | | | Feature: 1 - [a=0.5139, b=0.7569, c=1.0] : [0.0, 0.7367, 0.2633]
40 | | | | Feature: 1 - [a=0.7569, b=1.0, c=Inf] : [0.0, 0.0, 1.0]
41 | | | Feature: 2 - [a=0.5, b=0.7083, c=0.9167] : [0.0, 1.0, 0.0]
42 | | | Feature: 2 - [a=0.7083, b=0.9167, c=Inf] : [0.0, 0.0, 0.0]
43 | | Feature: 3 - [a=0.75, b=1.0, c=Inf] : [0.0, 0.0, 1.0]
44 | Feature: 4 - [a=0.5, b=0.75, c=1.0]
45 | | Feature: 3 - [a=-Inf, b=0.0, c=0.25] : [0.0, 0.0, 0.0]
46 | | Feature: 3 - [a=0.0, b=0.25, c=0.5] : [0.0, 1.0, 0.0]
47 | | Feature: 3 - [a=0.25, b=0.5, c=0.75]
48 | | | Feature: 2 - [a=-Inf, b=0.0833, c=0.2917] : [0.0, 0.4549, 0.5451]
49 | | | Feature: 2 - [a=0.0833, b=0.2917, c=0.5]
50 | | | | Feature: 1 - [a=-Inf, b=0.0278, c=0.2708] : [0.0, 1.0, 0.0]
51 | | | | Feature: 1 - [a=0.0278, b=0.2708, c=0.5139] : [0.0, 0.4906, 0.5094]
52 | | | | Feature: 1 - [a=0.2708, b=0.5139, c=0.7569] : [0.0, 0.2278, 0.7722]
53 | | | | Feature: 1 - [a=0.5139, b=0.7569, c=1.0] : [0.0, 0.6238, 0.3762]
54 | | | | Feature: 1 - [a=0.7569, b=1.0, c=Inf] : [0.0, 0.0, 0.0]
55 | | | Feature: 2 - [a=0.2917, b=0.5, c=0.7083]
56 | | | | Feature: 1 - [a=-Inf, b=0.0278, c=0.2708] : [0.0, 0.0, 0.0]
57 | | | | Feature: 1 - [a=0.0278, b=0.2708, c=0.5139] : [0.0, 0.7121, 0.2879]
58 | | | | Feature: 1 - [a=0.2708, b=0.5139, c=0.7569] : [0.0, 0.5752, 0.4248]
59 | | | | Feature: 1 - [a=0.5139, b=0.7569, c=1.0] : [0.0, 0.7958, 0.2042]
60 | | | | Feature: 1 - [a=0.7569, b=1.0, c=Inf] : [0.0, 0.0, 0.0]
61 | | | Feature: 2 - [a=0.5, b=0.7083, c=0.9167] : [0.0, 0.956, 0.044]
62 | | | Feature: 2 - [a=0.7083, b=0.9167, c=Inf] : [0.0, 0.0, 0.0]

```

Listing 25: Fuzzy Multi-way Decision Tree.

During the evolutionary process, the most relevant rules and their conditions are selected. Moreover, each triangular strong fuzzy partition P_f is concurrently tuned, by adapting the positions of the cores $b_{f,j}$.

1.15 PAES-RCS-FDT

In this Section, we describe the actual implementation of an MOEL scheme for classification problems in SK-MOEFS, namely PAES-RCS-FDT [1].

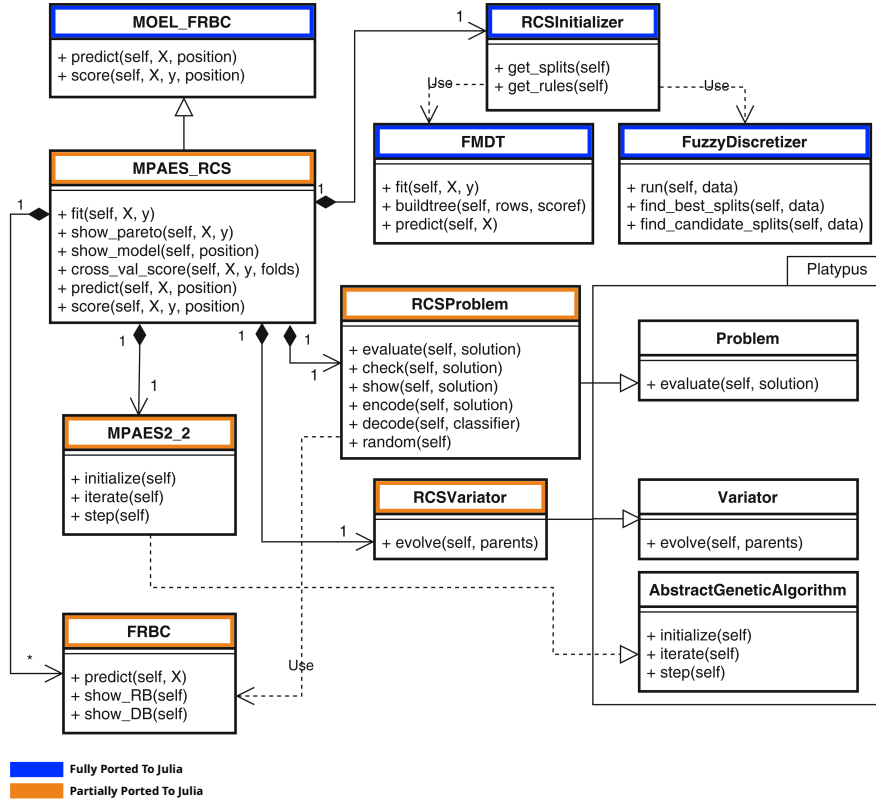


Figure 10: UML class diagram of PAES-RCS-FDT in SK-MOEFS.

In the UML class diagram, the classes fully ported to Julia are highlighted in blue while the ones only partially ported to Julia are highlighted in orange.

First of all, the MPAES_RCS is derived from the class MOEL_FRBC, which is in charge of handling the rule and condition selection multi-objective learning scheme, by means of $(2 + 2)$ M-PAES algorithm:

```
1 mutable struct MPAES_RCS
```

```

2      learning_scheme::MOEL_FRBC
3      M::Int64
4      Amin::Int64
5      capacity::Int64
6      divisions::Int64
7      variator::RCSVariator
8      initializer::RCSInitializer
9      objectives::Array{String}
10     moea_type::String
11     problem::RCSPProblem
12 end
13
14 MPAES_RCS() = MPAES_RCS(MOEL_FRBC(), 0, 0, 0, 0, RCSVariator(),
15     RCSInitializer(), [], "", RCSPProblem())
16 function __init__(self::MPAES_RCS, M::Int64=100, Amin::Int64=1,
17     capacity::Int64=64, divisions::Int64=8, variator::RCSVariator=
18     createRCSVariator(), initializer::RCSInitializer=
19     createRCSInitializer(), objectives=["accuracy", "trl"],
20     moea_type::String="mpaes22")
21     self.learning_scheme = __init__(MOEL_FRBC(), [])
22     self.M = M
23     self.Amin = Amin
24     self.capacity = capacity
25     self.divisions = divisions
26     self.objectives = objectives
27     self.variator = variator
28     self.initializer = initializer
29     self.moea_type = moea_type
30
31     return self
32 end
33
34 function _initialize(self::MPAES_RCS, X::Matrix{Float64}, y::Array{
35     Int64})
36     fit_tree(self.initializer, X, y)
37     J = get_rules(self.initializer)
38     splits = get_splits(self.initializer)
39
40     # Find initial set of rules and fuzzy partitions
41     # MOEFS problem
42     self.problem = CreateRCSPProblem(self.Amin, self.M, J, splits,
43     self.objectives)
44     set_training_set(self.problem, X, y)
45 end

```

Listing 26: MPAES_RCS implementation in Julia.

This class needs the `RCSPProblem`, which is a class derived from the `Problem` class of the Package *Platypus*, a Python framework for multiobjective evolutionary optimization. It defines operations on a possible solution (a chromosome) such as its encoding, feasibility checks, evaluation of objectives, and generation of a random solution:

```

1 mutable struct RCSPProblem
2     Amin::Int64
3     M::Int64

```

```

4     Mmax::Int64
5     Mmin::Int64
6     F::Int64
7     TFmax::Int64
8     J::Matrix{Int64}
9     initialBfs::Array{Array{Float64}}
10    G::Array{Int64}
11    objectives::Array{String}
12    crb_l::Int64
13    gran_l::Int64
14    cdb_l::Int64
15    problem::PyObject
16    types::Array{Any}
17    directions::Array{Int64}
18    train_x::Matrix{Float64}
19    train_y::Array{Int64}
20 end
21
22 RCSPProblem() = RCSPProblem(0, 0, 0, 0, 0, 0, Array{Int64, 2}(undef,
    1, 1), [], [], [], 0, 0, 0, platypus.Problem, [], [], Array{
    Float64, 2}(undef, 1, 1), [])
23
24 function __init__(self::RCSPProblem, Amin::Int64, M::Int64, J::
    Matrix{Int64}, splits::Array{Array{Float64}}, objectives::Array
    {String}, TFmax::Int64=7)
25     self.Amin = Amin
26     self.M = M
27     self.Mmax = size(J)[1]
28     self.Mmin = length(unique(J[:, end]))
29     self.M = clamp(self.M, self.Mmin, self.Mmax)
30     self.F = size(J)[2] - 1
31     self.TFmax = TFmax
32
33     self.J = J
34     self.initialBfs = splits
35     self.G = [length(split) for split in splits]
36
37     self.objectives = objectives
38     self.crb_l = 2 * self.M
39     self.gran_l = self.F
40     self.cdb_l = self.gran_l * self.TFmax
41
42     # Creation of the problem and assignment of the types
43     # CRB: rule part
44     # Granularities: number of fuzzy sets
45     # CDB: database part
46     self.problem = platypus.Problem(self.crb_l + self.gran_l + self
    .cdb_l, length(objectives))
47     self.types = []
48     for _ in range(1, self.M)
49         push!(self.types, platypus.Real(0, self.M))
50         push!(self.types, platypus.Binary(self.F))
51     end
52     [push!(self.types, platypus.Real(2, self.TFmax)) for _ in range
    (1, self.F)]
53     [push!(self.types, platypus.Real(0, 1)) for _ in range(1, self
    .cdb_l)]

```

```

54
55     # Maximize objectives by minimizing opposite
56     self.directions = [platypus.Problem.MINIMIZE for _ in range(1,
57                          length(objectives))]
58
59     return self
60 end
61 function set_training_set(self::RCSProblem, train_x::Matrix{Float64}
62     }, train_y::Array{Int64})
63     self.train_x = train_x
64     self.train_y = train_y
65 end
66 function CreateRCSProblem(Amin::Int64, M::Int64, J::Matrix{Int64},
67     splits::Array{Array{Float64}},
68     objectives::Array{String}, TFmax::Int64=7)
69     return __init__(RCSProblem(), Amin, M, J, splits, objectives,
70                     TFmax)
71 end

```

Listing 27: RCSProblem implementation in Julia.

Moreover, the MPAES_RCS class adopts an RCSVariator, a particular implementation of the Platypus Variator, which includes all the mating operators, namely mutation and crossover, appropriately defined for acting on the chromosome and generating offsprings:

```

1 mutable struct RCSVariator
2     variator::PyObject
3     p_crb::Float64
4     p_cdb::Float64
5     alpha::Float64
6     p_mrb::Float64
7     p_mrb_add::Float64
8     p_mrb_del::Float64
9     p_mrb_flip::Float64
10    p_mdb::Float64
11    p_cond::Float64
12 end
13
14 RCSVariator() = RCSVariator(platypus.Variator, 0.0, 0.0, 0.0, 0.0,
15     0.0, 0.0, 0.0, 0.0, 0.0)
16
17 function __init__(self::RCSVariator, p_crb::Float64=0.2, p_cdb::
18     Float64=0.5, alpha::Float64=0.5, p_mrb::Float64=0.1, p_mrb_add
19     ::Float64=0.5, p_mrb_del::Float64=0.2, p_mrb_flip::Float64=0.7,
20     p_mdb::Float64=0.2, p_cond::Float64=0.5)
21     self.variator = platypus.Variator(2)
22     self.p_crb = p_crb
23     self.p_cdb = p_cdb
24     self.alpha = alpha
25     self.p_mrb = p_mrb
26     self.p_mrb_add = p_mrb_add
27     self.p_mrb_del = p_mrb_del
28     self.p_mrb_flip = p_mrb_flip
29     self.p_mdb = p_mdb

```

```

26     self.p_cond = p_cond
27
28     return self
29 end

```

Listing 28: RCSVariator implementation in Julia.

Two additional classes are adopted as compositions by the MPAES_RCS class, namely the MPAES2_2 and the RCSInitializer. Specifically, the MPAES2_2 class extends the AbstractGeneticAlgorithm class of Platypus and implements (2 + 2)M-PAES:

```

1 mutable struct MPAES2_2
2     algorithm::PyObject
3     variator::RCSVariator
4     dominance::PyObject
5     archive::PyObject
6 end
7
8 MPAES2_2() = MPAES2_2(platypus.algorithms.AbstractGeneticAlgorithm,
9     RCSVariator(), platypus.ParetoDominance, platypus.
10    AdaptiveGridArchive)
11
12 function __init__(self::MPAES2_2, problem::RCSProblem, variator::
13    RCSVariator=nothing, capacity::Int64=32, divisions::Int64=8,
14    generator::MOEAGenerator=MOEAGenerator())
15     self.algorithm.__init__(problem, population_size=2, generator=
16     generator)
17
18     self.variator = variator
19     self.dominance = platypus.ParetoDominance()
20     self.archive = platypus.AdaptiveGridArchive(capacity, problem.
21     nobjs, divisions)
22
23     # from time to time we take snapshots of the archive for
24     debugging purposes
25     self.snapshots = []
26 end

```

Listing 29: MPAES2_2 implementation in Julia.

Finally, the RCSInitializer implements the methods for the definition of the initial strong fuzzy partitions and for generating the initial set of rules

```

1 mutable struct RCSInitializer
2     discretizer::FuzzyDiscretization
3     tree::FMDT
4     fTree::FMDT
5     splits::Array{Array{Float64}}
6     rules::Matrix{Int64}
7 end
8
9 RCSInitializer() = RCSInitializer(FuzzyDiscretization(), FMDT(),
10    FMDT(), [], Array{Int64, 2}(undef, 1, 1))
11
12 function __init__(self::RCSInitializer, discretizer::
13    FuzzyDiscretization=createFuzzyDiscretizer("uniform", 5), tree
14    ::FMDT=createFMDT())

```



```

12     self.discretizer = discretizer
13     self.tree = tree
14     self.splits = []
15
16     return self
17 end
18
19 function fit_tree(self::RCSInitializer, X::Matrix{Float64}, y::
    Vector{Int64})
20     continuous = ones(Bool, size(X)[2])
21     cPoints = runFuzzyDiscretizer(self.discretizer, X, continuous)
22     self.fTree = fitFMDTTree(self.tree, X, y, continuous, cPoints)
23     self.rules = _csv_ruleMine(self.fTree.tree, size(X)[2], [])
24     self.rules[:, end] .= 1
25     self.splits = self.fTree.cPoints
26 end
27
28 function get_splits(self::RCSInitializer)
29     return self.splits
30 end
31
32 function get_rules(self::RCSInitializer)
33     return self.rules
34 end
35
36 function createRCSInitializer(discretizer::FuzzyDiscretization=
    createFuzzyDiscretizer("uniform", 5), tree::FMDT=createFMDT())
37     return __init__(RCSInitializer(), discretizer, tree)
38 end

```

Listing 30: RCSInitializer implementation in Julia.

To this aim, this class uses the fuzzy discretizer (implemented by `FuzzyDiscretizer.jl`) and the Multi-way Fuzzy Decision Tree (implemented by `fmdt.jl`), respectively.

1.16 example.jl

This provides the Julia implementation for `example.py`. It contains three SK-MOEFS usage examples:

- `example1()`
- `example2(seed::Int64)`
- `example3(dataset::String, algorithm::String, seed::Int64, nEvals::Int64=50000, store::Bool=false)`

The usage of the rule and condition selection multiobjective learning scheme, by means of $(2 + 2)$ M-PAES algorithm, is demonstrated on different datasets. Finally, the pareto front, the pareto archive, the rule base of the first, median and last solution and the associated strong fuzzy partitions are plotted.

In this Section, we show some examples of usage of SK-MOEFS in Julia. Specifically, we show how to generate a set of FRBCs, characterized by different

trade-offs between accuracy and explainability, using the PAES-RCS-FDT implementation. To this aim, we have selected the **NewThyroid** dataset¹⁵: the classification task associated with this dataset is to detect if a given patient is normal (Class 1), suffers from hyperthyroidism (Class 2) or hypothyroidism (Class 3).

Since the porting of the entire code base was not terminated, the `PyCall.jl` package was used in order to access the missing code:

```

1 include("skmoefs/toolbox.jl")
2 include("skmoefs/discretization/discretizer_base.jl")
3 using PyCall
4 using Random
5 using ScikitLearn.CrossValidation: train_test_split
6
7 # add path for importing local skmoefs package
8 pushfirst!(PyVector(pyimport("sys")["path"]), "python/")
9
10 # import python modules
11 skmoefs_py_discretization = pyimport("skmoefs.discretization.
    discretizer_base")
12 skmoefs_py_toolbox = pyimport("skmoefs.toolbox")
13 skmoefs_py_rcs = pyimport("skmoefs.rcs")
14
15 function example2(seed::Int64)
16     set_rng_seed(seed)
17
18     X, y, attributes, inputs, outputs = load_dataset("newthyroid")
19     X_n, y_n = normalize(X, y, attributes)
20     Xtr, Xte, ytr, yte = train_test_split(X_n, y_n, test_size=0.3)
21
22     mpaes_rcs = skmoefs_py_toolbox.MPAES_RCS(capacity=32, variator=
    skmoefs_py_rcs.RCSVariator(), initializer=skmoefs_py_rcs.
    RCSInitializer())
23     mpaes_rcs.fit(Xtr, ytr, max_evals=30000)
24
25     mpaes_rcs.show_pareto()
26     mpaes_rcs.show_pareto(Xte, yte)
27     mpaes_rcs.show_pareto_archives()
28     mpaes_rcs.show_pareto_archives(Xte, yte)
29     mpaes_rcs.show_model("median", inputs=inputs, outputs=outputs)
30 end

```

Listing 31: Julia Porting.

In code Listing 31, we show an example of usage, in which we first load a dataset from a file and then we divide it into training and test sets. Second, we instantiate an `MPAES-RCS` object passing to its constructor the `RCSVariator` and an `RCSInitializer`. The latter will partition each input attributes into a predefined number of fuzzy sets. Afterward, we call the `fit` method, which returns the fitted model having now a list of the FRBCs characterized by different trade-offs between accuracy and explainability. Then, we call the method for showing the Pareto front approximation (in Fig. 11), both on the training and test sets.

¹⁵<https://sci2s.ugr.es/keel/dataset.php?cod=66>

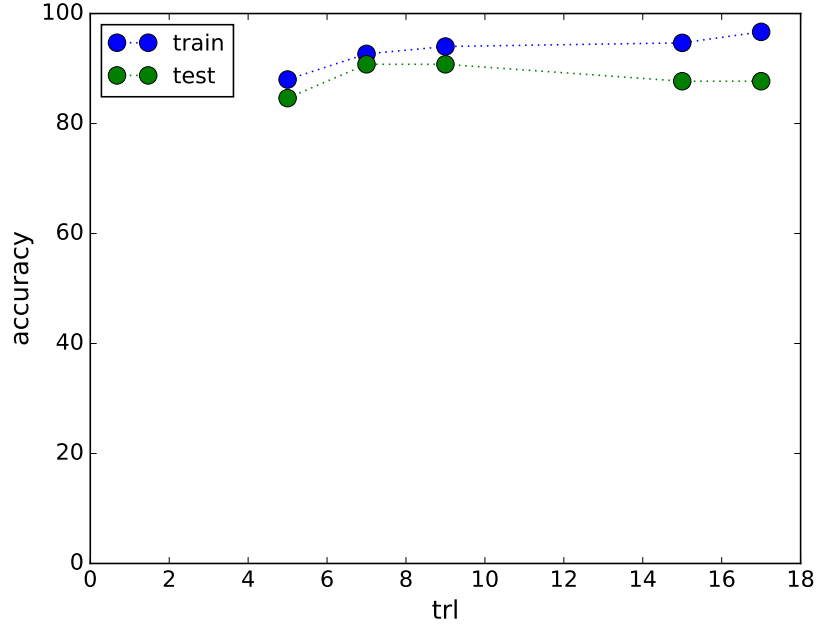


Figure 11: Pareto Front approximation both on the training and test sets.

The algorithm concurrently optimizes two objectives: the first objective considers the TRL as explainability measure; the second objective takes into account the accuracy, assessed in terms of classification rate.

All the multiobjective evolutionary algorithms extended starting from the **Platypus** Python package were extended by adding **snapshots**: from time to time a snapshot of the archive is taken for debugging purposes. Predefined milestones are used for taking such snapshots:

```
1 milestones = [500, 1000, 2000, 5000, 10000, 20000, 30000, 40000,
               50000, 75000, 100000]
```

Listing 32: Milestones definition in `moea.jl`.

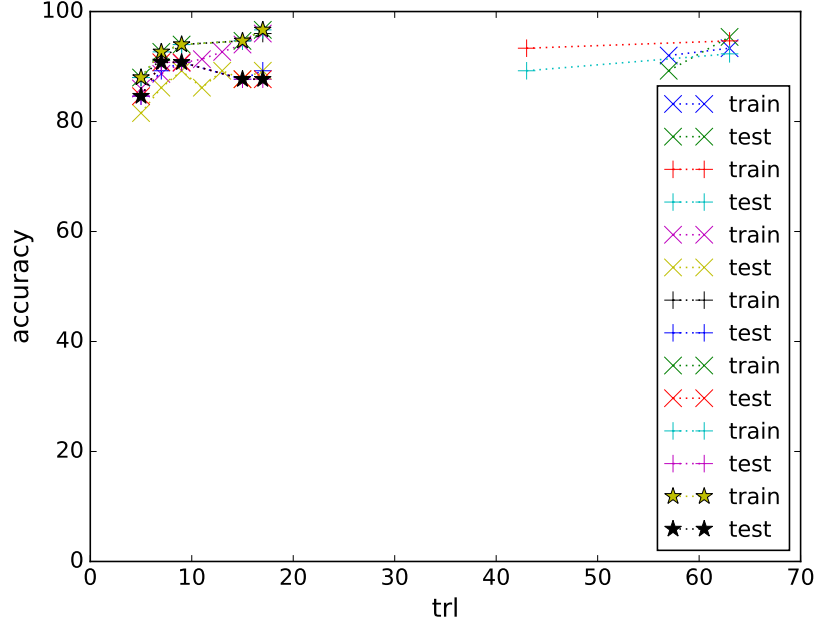


Figure 12: Pareto Archives snapshots.

Finally, we show the RB and the fuzzy partitions of the MEDIAN solution in the Pareto Front approximation.

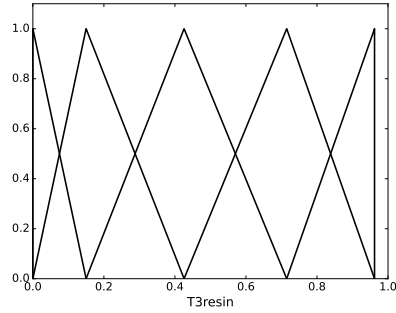
```

1 RULE BASE
2 1: IF Thyroxin is VL THEN Class is 3
3 2: IF T3resin is M AND Thyroxin is L THEN Class is 1
4 3: IF T3resin is H AND Thyroxin is M THEN Class is 1
5 4: IF T3resin is M AND Thyroxin is H THEN Class is 2
6 5: IF T3resin is L AND Thyroxin is H THEN Class is 2

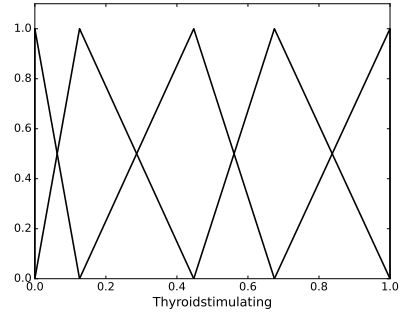
```

Listing 33: Rule Base of MEDIAN solution.

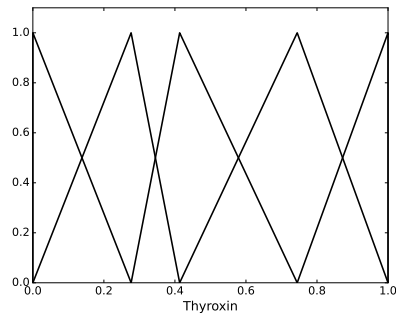
In this example, we labeled the five fuzzy sets of each partition as Very Low (VL), Low (L), Medium (M), High (H), and Very High (VH). As we can see, the set of linguistic rules allows the user to understand the motivation of a decision: simply speaking, based on the levels of each input attribute describing a new patient, a specific class is associated with him/her. As regards the fuzzy partitions, it is worth noting, especially for the last two, that they moved from the initial uniform shape. However, they are still strong fuzzy partitions, thus ensuring a good integrity level, in terms of order, distinguishability, coverage and normality.



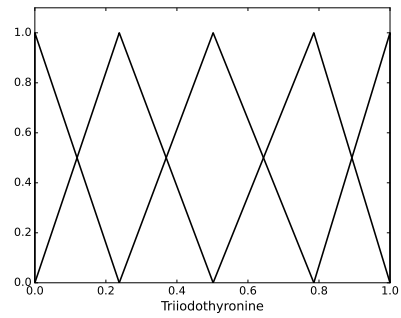
(a) T3resin



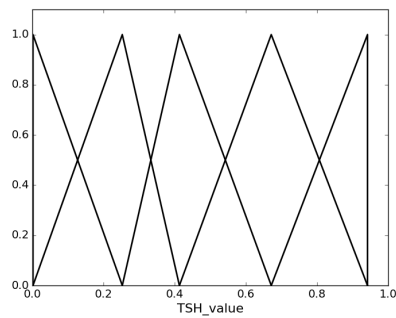
(b) Thyroidstimulating



(c) Thyroxin



(d) Triiodothyronine



(e) TSH_value

1.17 test.jl

In order to be able to run the full demonstrative examples found in `example.jl`, the entire code base had to be ported to Julia. Writing code without debugging

is not for everyone, even more when it has to be done in a completely new language. The `test.jl` contains a series of tests to verify the implemented functionalities one by one. As an example, consider the following:

```
1 fuzzy_mdldf_discretizer = createFuzzyMDLDiscretizer(3, X_n, y, [true
  , true, true, true])
2 fuzzy_mdldf_splits = runFuzzyMDLDiscretizer(fuzzy_mdldf_discretizer)
3 println("Fuzzy MDLF Discretizer:\n" * string(fuzzy_mdldf_splits))
```

Listing 34: FuzzyMDLFilter test in Julia.

```
1 fuzzy_mdldf_discretizer = FuzzyMDLFilter(3, X_n, y_n, [True, True,
  True, True])
2 fuzzy_mdldf_splits = fuzzy_mdldf_discretizer.run()
3 print("Fuzzy MDLF Discretizer:\n" + str(fuzzy_mdldf_splits))
```

Listing 35: FuzzyMDLFilter test in Python.

2 Execution Time Benchmark

Among the reasons that motivated the porting of the original Python implementation to Julia, execution time is for sure to be taken into account. This is why a benchmark was performed in order to have an estimate of the execution time.

The benchmark was performed taking into account `test.py` and `test.jl` which contain the same exact source code the corresponding languages.

In order to measure the execution time of `test.py` the command-line benchmarking tool `hyperfine` command was used¹⁶:

```
$hyperfine --warmup 3 -r 100 'python python/test.py'
Benchmark 1: python python/test.py
  Time (mean ± σ):      2.519 s ± 0.103 s    [User: 2.779 s, System: 0.657 s]
  Range (min ... max):  2.482 s ... 3.250 s    100 runs
```

Figure 13: Python test script execution time benchmark using `hyperfine`.

An additional confirmation run was performed using the Unix bash `time` command:

```
1 $ time python python/test.py
2
3 real    0m2.396s
4 user    0m2.643s
```

¹⁶This might sound as not the most appropriate way of doing it, but all the alternatives provided by Python packages such as `timeit`, `cProfile`, `pycallgraph` require a much greater coding effort.

```
5 sys 0m0.675s
```

Listing 36: Python test script execution time benchmark using `time`.

As far as it concerns benchmarking the execution time of `test.jl`, the `BenchmarkTools.jl`¹⁷ package was used:

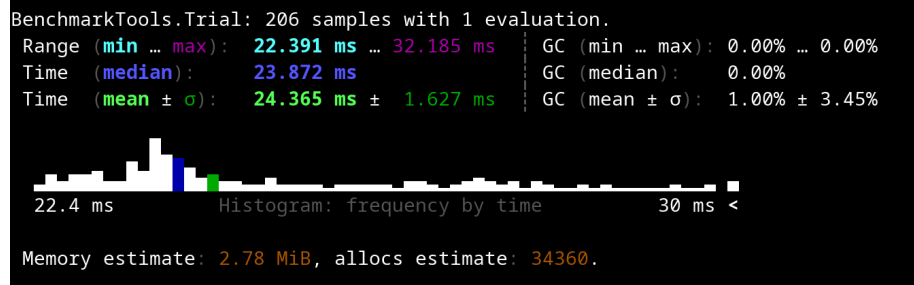


Figure 14: Julia test script execution time benchmark using `BenchmarkTools.jl`.

An additional confirmation run was performed using the Unix bash `@time` macro:

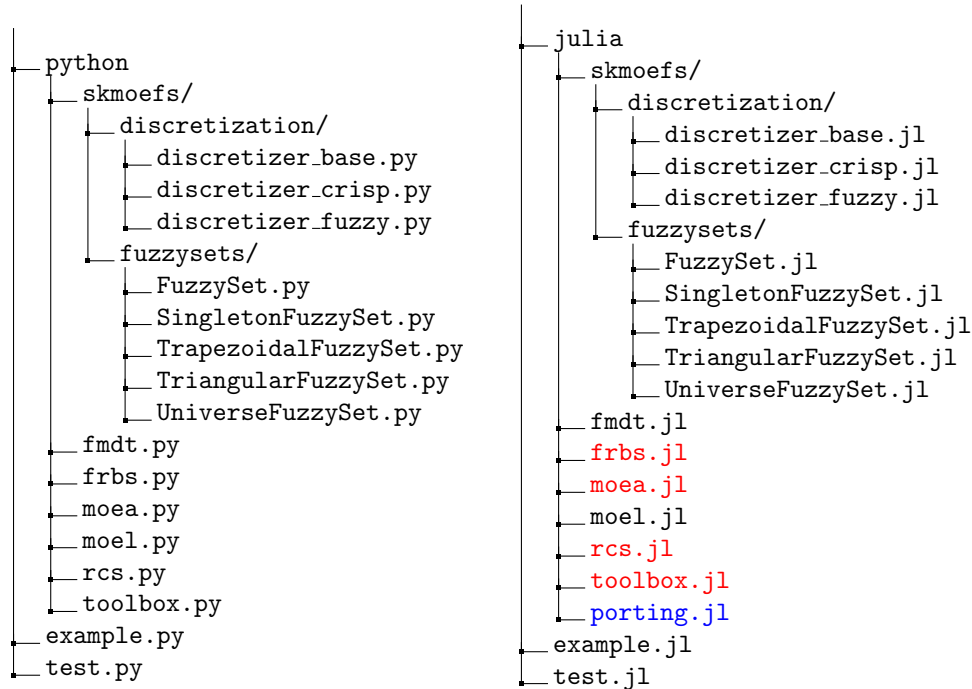
```
1 julia> include("julia/test.py")
2
3 0.579320 seconds (676.12 k allocations: 37.472 MiB, 2.92% gc time,
  94.90% compilation time)
```

Listing 37: Julia test script execution time benchmark using `@time`.

3 Conclusions

Comparing the directory listings of the Python and Julia implementations provides a clear picture of the results that were obtained at the end of this work:

¹⁷<https://github.com/JuliaCI/BenchmarkTools.jl>



In the Julia directory listing, items in **red** represent source code files for which the porting was not fully completed due to dependencies with Python packages. In such cases an initial implementation by means of the PyCall package is still provided. Items in **blue** represent source code files not present in the original Python implementation.

4 Future work

Possible future works may focus on

- replace `ScikitLearn.jl` with a machine-learning framework specially-designed for Julia: MLJ (Machine Learning in Julia), by the Alan Turing Institute¹⁸, is a toolbox written in Julia providing a common interface and meta-algorithms for selecting, tuning, evaluating, composing and comparing over 180 machine learning models written in Julia and other languages;
- finish `rcs.jl` implementation: only the `RCSInitializer` class implementation was completely ported to Julia in `rcs.jl`; this is due to the fact that the two remaining classes `RCSVariator` and `RCSProblem`, both partially implemented in Julia, depend on the Python Platypus package;
- finish `moea.jl` implementation: the code for the multiobjective evolutionary algorithms implemented in `moea.py` heavily depends on the Python

¹⁸<https://www.turing.ac.uk/>

Platypus package; however, finishing the porting of this file by means of the PyCall package will allow to complete and test the `rcs.jl` source code;

- finish `frbs.jl` implementation: almost 90% of the code was ported to Julia, however there is no proper way of testing it properly because of the lack of a complete implementation of `rcs.jl`;
- integrate existing PL-NSGA-II Julia implementation;

References

- [1] Michela Antonelli, Pietro Ducange, and Francesco Marcelloni. A fast and efficient multi-objective evolutionary learning scheme for fuzzy rule-based classifiers. *Inf. Sci.*, 283:36–54, nov 2014.
- [2] Eva Armengol and Àngel García-Cerdàña. Refining discretizations of continuous-valued attributes. In Vicenç Torra, Yasuo Narukawa, Beatriz López, and Mateu Villaret, editors, *Modeling Decisions for Artificial Intelligence*, pages 258–269, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- [3] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.
- [4] Pietro Ducange, Giuseppe Mannarà, and Francesco Marcelloni. Multi-objective evolutionary granular rule-based classifiers: An experimental comparison. In *2017 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, pages 1–6, 2017.
- [5] Gionatan Gallo, Vincenzo Ferrari, Francesco Marcelloni, and Pietro Ducange. Sk-moefs: A library in python for designing accurate and explainable fuzzy models. In *Information Processing and Management of Uncertainty in Knowledge-Based Systems*, pages 68–81, Cham, 2020. Springer International Publishing.
- [6] R. Huijzer J. Storopoli and L. Alonso. *Julia Data Science*. Independently published, 2021.
- [7] Perkel JM. Julia: come for the syntax, stay for the speed. *Nature*, 2019.
- [8] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, nov 2011.

- [9] Armando Segatori, Francesco Marcelloni, and Witold Pedrycz. On distributed fuzzy decision trees for big data. *Trans. Fuz Sys.*, 26(1):174–192, feb 2018.
- [10] Armando Segatori, Francesco Marcelloni, and Witold Pedrycz. On distributed fuzzy decision trees for big data. *IEEE Transactions on Fuzzy Systems*, 26(1):174–192, 2018.