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 4, 2022

0 Introduction

The project focused on porting the Python implementation of SK-MOEFS [5] 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 (acronym of SciKit-Multi Objective Evolution- ary Fuzzy System), 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 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 [4] 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.

[1] https://julialang.org/

The creators of Julia explained why they created Julia in a 2012 blogpost[2]. 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.

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:

```
└── skmoefs/
    ├── discretization/
    │   ├── __init__.py
    │   ├── discretizer_base.py
    │   ├── discretizer_crisp.py
    │   └── discretizer_fuzzy.py
    └── fuzzysets/
        ├── FuzzySet.py
        ├── __init__.py
        ├── SingletonFuzzySet.py
        ├── TrapezoidalFuzzySet.py
        ├── TriangularFuzzySet.py
        └── UniverseFuzzySet.py
├── __init__.py
├── example.py
└── fmdt.py
```

[2] https://julialang.org/blog/2012/02/why-we-created-julia/

```
├── frbs.py
├── moea.py
├── moel.py
├── rcs.py
├── toolbox.py
└── test.py
```

A fork was created starting from the original implementation[3] and can be found in this repository: `https://github.com/rambodrahmani/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:

```
@relation iris
@attribute SepalLength real [4.3, 7.9]
@attribute SepalWidth real [2.0, 4.4]
@attribute PetalLength real [1.0, 6.9]
@attribute PetalWidth real [0.1, 2.5]
@attribute Class {0, 1, 2}
@inputs SepalLength, SepalWidth, PetalLength, PetalWidth
@outputs Class
@data
5.1, 3.5, 1.4, 0.2, 0
4.9, 3.0, 1.4, 0.2, 0
4.6, 3.1, 1.5, 0.2, 0
```

Listing 1: 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::MatrixFloat64`: representing the input data samples;
- `y::ArrayInt64`: representing the class labels
- `attributes::ArrayArrayFloat64`: range of values in the format [min, max] for each feature;

[3] `https://github.com/GionatanG/skmoefs`

- `inputs::ArrayString`: names of the input features;
- `outputs::ArrayString`: names of the output features;

1.2 Classes

Julia does not have classes in the object-oriented sense[4]. Instead we define new types and then define methods on those types. The closest analogy to a Python class is a `mutable struct`.
Actually, there is no need to mimic python class, because it is 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:

```python
class PyClass:
    def __init__(self):
        self.a = 1.0
        self.b = 2.0

    def method1(self):
        return self.a + self.b

mim = PyClass()
mim.method1()
```

Listing 2: Python Class.

```julia
mutable struct Pyclass
    a::Float
    b::Float
end

PyClass() = PyClass(0.0, 0.0)

function __init__(self::PyClass)
    self.a = 1.0
    self.b = 2.0
end

function method1(self::PyClass)
    return self.a + self.b
end

mim = __init__(PyClass())
method1(mim)
```

Listing 3: 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

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

In Julia you have

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

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

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

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.

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

Listing 4: 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 possible[5]:

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

Listing 5: 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...

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

Listing 6: 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]:

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

Listing 7: Snippet from `discretizer_crisp.jl`.

[5] `https://docs.julialang.org/en/v1/base/io-network/#Text-I/O`
[6] `https://docs.julialang.org/en/v1/stdlib/Logging/`

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 them[7].
The following example demonstrates how `PyCall` allows to use the `Platypus` Python package directly in Julia[8]:

```julia
using PyCall
platypus = pyimport("platypus")

mutable struct NSGAIIS
    """
    Extended version of NSGA2 algorithm with added support for snapshots.
    """
    snapshots::Array{Any}
    algorithm::PyObject
end

NSGAIIS() = NSGAIIS([], platypus.algorithms.NSGAII)

function initialize(self::NSGAIIS)
    self.snapshots = []
    algorithm = platypus.algorithms.NSGAII()
    algorithm.initialize()

    return self
end

function iterate(self::NSGAIIS)
    if (self.algorithm.nfe % 100) == 0
        println("Fitness evaluations " * self.algorithm.nfe)
    end
    if length(self.snapshots) < length(milestones) &&
        self.algorithm.nfe >= milestones[length(self.snapshots)]
        print("new milestone at " * string(self.algorithm.nfe))
        append!(self.snapshots, self.algorithm.archive)
    end
    self.algorithm.iterate()
end
```

Listing 8: 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]:

```julia
using PyCall

```

[7] https://github.com/JuliaPy/PyCall.jl

[8] Platypus is a framework for evolutionary computing in Python with a focus on multiobjective evolutionary algorithms (MOEAs). https://platypus.readthedocs.io/en/latest/

[9] https://github.com/JuliaPy/PyCall.jl/issues/48

```
# add path for importing local skmoefs package
pushfirst!(PyVector(pyimport("sys")["path"]), "python/")

# import python modules
skmoefs_py_toolbox = pyimport("skmoefs.toolbox")
skmoefs_py_rcs = pyimport("skmoefs.rcs")

function test1(seed::Int64)
    set_rng_seed(seed)

    X, y, attributes, inputs, outputs = load_dataset("newthyroid")
    X_n, y_n = normalize(X, y, attributes)
    Xtr, Xte, ytr, yte = train_test_split(X_n, y_n, test_size=0.3)

    my_moefs = skmoefs_py_toolbox.MPAES_RCS(capacity=32, variator=
    skmoefs_py_rcs.RCSVariator(), initializer=skmoefs_py_rcs.
    RCSInitializer())
    my_moefs.fit(Xtr, ytr, max_evals=1000)

    my_moefs.show_pareto()
    my_moefs.show_pareto(Xte, yte)
    my_moefs.show_model("median", inputs=inputs, outputs=outputs)
end
```

Listing 9: Snippet from `example.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 relaying 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 object[10].

```
function is_object_present(name)
    """
    Check if file exists on the filesystem.
    """
    return isfile(name * ".obj")
end

function store_object(filename::String, name::String, object::Any)
    """
    Save object as Julia Data format (JLD).
    """
    save(filename * ".jld", name, object)
```

[10] `https://github.com/JuliaIO/JLD.jl`

```
end

function load_object(filename::String)
    """
    Load JLD file to object.
    """
    return load(filename * ".jld")
end
```

Listing 10: Snippet from `toolbox.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`).

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 1[11]:

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

Listing 11: 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 following[12]:

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

Listing 12: 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 [2] 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 inter- vals. 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 [3].
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.

```
function entropy(self::CrispMDLFilter, counts::Array{Float64},
    totalCount)
    if totalCount == 0
        return 0
    end

    numClasses = length(counts)
    impurity = 0.0
    classIndex = 1
    while (classIndex <= numClasses)
        classCount = counts[classIndex]
```

```
        if classCount != 0
            freq = classCount / totalCount
            if freq != 0
                impurity -= freq * log2(freq)
            end
        end
        classIndex += 1
    end
    return impurity
end
```

Listing 13: $Info(D)$ 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 $Gain(A)$;

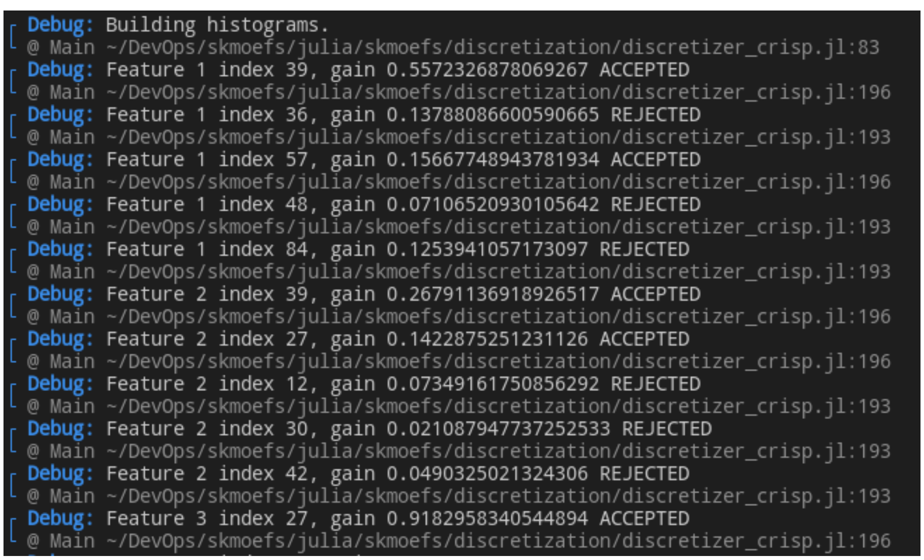


Figure 1: Crisp MDL Discretizer debug logs.

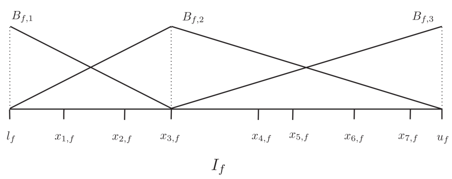
- the `BisectPy.jl`[13] Julia package, which implements Python `bisect` module, was used;
  - since Julia's array index starts from 1 but Python starts from 0, the returned index of either `bisect_left` or `bisect_right` is always their Python's correspondence plus 1; also, the behavior of Python's `a[:i]` 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`

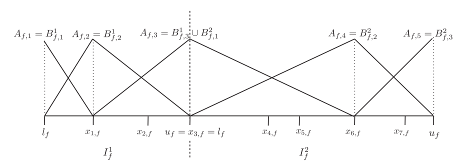
This implements the fuzzy MDL discretizer available in the SK-MOEFS library (`discretizer_fuzzy.py`). A strong fuzzy partition is determined on each continuous attribute by using a novel discretizer based on the fuzzy entropy. [10] Partitioning of continuous attributes is a crucial aspect in the generation 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 equiped 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 $[l_f, x^0_{i,f}]$ and $(x^0_{i,f}, u_f]$. Actually, the two fuzzy sets, which have the core in $x^0_{i,f}$, 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 $P_f = \{A_{f,1}, \cdots, A_{f,T_f}\}$ on U_f, where $A_{f,j}$, with $j = 1, \cdots, T_f$, 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 $X = \{X_1, \cdots, X_F\}$ be the set of input attributes and X_{F+1} be the output attribute. Let U_f, with $f = 1, ..., F+1$, be the universe of the f-th attribute X_f. Let $P_f = \{A_{f,1}, \cdots, A_f, T_f\}$ be a fuzzy partition of T_f fuzzy sets on attribute X_f. Finally, we define the training set $\{(x_1, x_{F+1,1}), \cdots, (x_N, x_{F+1,N})\}$ as a collection of N input-output pairs, with $x_t = [x_{t,1}, \cdots, x_{t,F}] \in \mathbb{R}, t = 1, \cdots, N$. In regression problems, X_{F+1} is a continuous attribute and, therefore, $\forall t \in [0 \cdots N], x_{F+1,t} \in \mathbb{R}$. 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:

$$\begin{aligned} R_m \ : \ &\textbf{IF } X_1 \textbf{ is } A_{1,j_m,1} \textbf{ AND } \cdots \textbf{ AND } X_f \textbf{ is } A_{f,j_m,f} \textbf{ AND} \\ &\cdots \textbf{ AND } X_F \textbf{ is } A_{F,j_m,F} \textbf{ THEN } X_{F+1} \textbf{ is } A_{F+1,j_m,F+1} \end{aligned}$$

In classification problems, X_{F+1} is categorical and $x_{F+1,t} \in C$, where $C = \{C_1, \cdots, C_K\}$ 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:

$$\begin{aligned} R_m \ : \ &\textbf{IF } X_1 \textbf{ is } A_{1,j_m,1} \textbf{ AND } \cdots \textbf{ AND } X_f \textbf{ is } A_{f,j_m,f} \textbf{ AND} \\ &\cdots \textbf{ AND } X_F \textbf{ is } A_{F,j_m,F} \textbf{ THEN } X_{F+1} \textbf{ is } C_{j_m} \textbf{ with } RW_m \end{aligned}$$

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_{jm} 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 ($upper\ boundary = \infty$), `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 designed[14].

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.

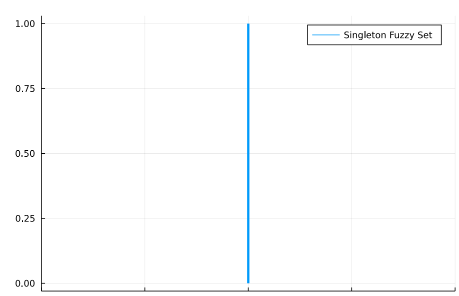


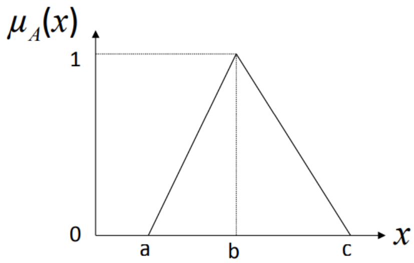
Figure 2: Fuzzy Singleton plot in Julia.

[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

$$\mu_A(x) = \begin{cases} 0 & \text{if } x \le a \\ \dfrac{x-a}{b-a} & \text{if } a \le x \le b \\ \dfrac{c-x}{c-b} & \text{if } b \le x \le c \\ 0 & \text{if } x \ge c \end{cases}$$

(b) Triangular Fuzzy Set Membership Function

In Julia, one can be instantiated as follows:

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

Listing 14: Julia implementation of `numpy.isscalar`.

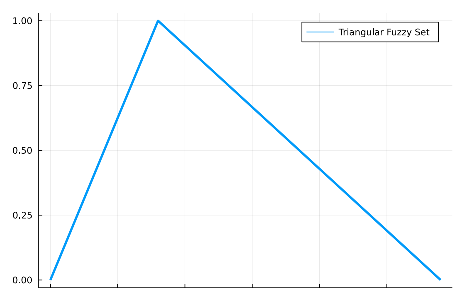
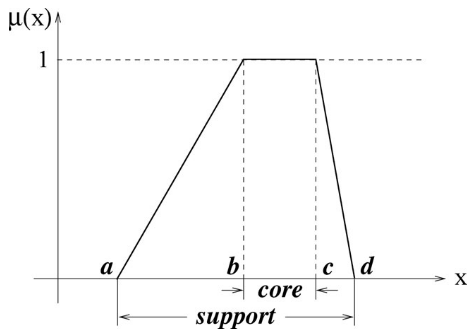


Figure 3: Fuzzy Triangular set plot in Julia.

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

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

(b) Trapezoidal Fuzzy Set Membership Function

In Julia, one can be instantiated as follows:

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

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

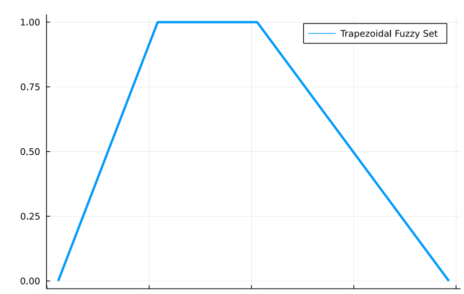


Figure 4: Fuzzy Trapezoidal set plot in Julia.

1.11 Strong triangular fuzzy partitions

In the current version of SK-MOEFS, **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.

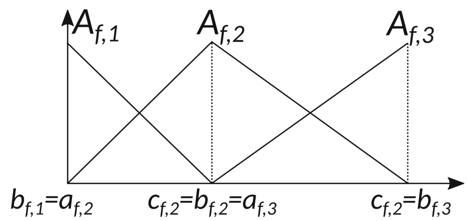


Figure 5: An example of a strong triangular fuzzy partition with three fuzzy sets.

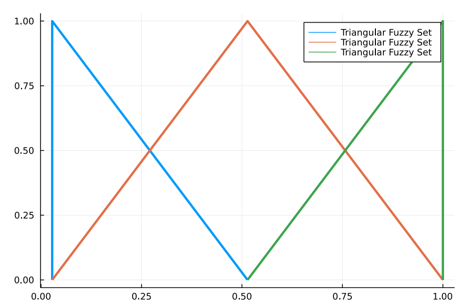


Figure 6: Strong triangular fuzzy partition in Julia.

1.12 `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.

```
Feature: 4 - [a=-Inf, b=0.0, c=0.25] : [1.0, 0.0, 0.0]
Feature: 4 - [a=0.0, b=0.25, c=0.5]
|    Feature: 3 - [a=-Inf, b=0.0, c=0.25] : [1.0, 0.0, 0.0]
|    Feature: 3 - [a=0.0, b=0.25, c=0.5]
|    |    Feature: 2 - [a=-Inf, b=0.0833, c=0.2917] : [0.0978, 0.9022, 0.0]
|    |    Feature: 2 - [a=0.0833, b=0.2917, c=0.5] : [0.2344, 0.7656, 0.0]
|    |    Feature: 2 - [a=0.2917, b=0.5, c=0.7083] : [1.0, 0.0, 0.0]
|    |    Feature: 2 - [a=0.5, b=0.7083, c=0.9167] : [1.0, 0.0, 0.0]
|    |    Feature: 2 - [a=0.7083, b=0.9167, c=Inf] : [1.0, 0.0, 0.0]
|    Feature: 3 - [a=0.25, b=0.5, c=0.75] : [0.0, 1.0, 0.0]
|    Feature: 3 - [a=0.5, b=0.75, c=1.0] : [0.0, 1.0, 0.0]
|    Feature: 3 - [a=0.75, b=1.0, c=Inf] : [0.0, 0.0, 0.0]
Feature: 4 - [a=0.25, b=0.5, c=0.75]
|    Feature: 3 - [a=-Inf, b=0.0, c=0.25] : [0.0, 0.0, 0.0]
|    Feature: 3 - [a=0.0, b=0.25, c=0.5] : [0.0, 1.0, 0.0]
|    Feature: 3 - [a=0.25, b=0.5, c=0.75]
|    |    Feature: 1 - [a=-Inf, b=0.0278, c=0.2708] : [0.0, 1.0, 0.0]
|    |    Feature: 1 - [a=0.0278, b=0.2708, c=0.5139] : [0.0, 0.9912, 0.0088]
|    |    Feature: 1 - [a=0.2708, b=0.5139, c=0.7569]
|    |    |    Feature: 1 - [a=-Inf, b=0.0278, c=0.2708] : [0.0, 0.0, 0.0]
|    |    |    Feature: 1 - [a=0.0278, b=0.2708, c=0.5139] : [0.0, 0.9773, 0.0227]
|    |    |    Feature: 1 - [a=0.2708, b=0.5139, c=0.7569] : [0.0, 0.914, 0.086]
|    |    |    Feature: 1 - [a=0.5139, b=0.7569, c=1.0] : [0.0, 0.9644, 0.0356]
|    |    |    Feature: 1 - [a=0.7569, b=1.0, c=Inf] : [0.0, 0.0, 0.0]
|    |    Feature: 1 - [a=0.5139, b=0.7569, c=1.0] : [0.0, 0.9806, 0.0194]
|    |    Feature: 1 - [a=0.7569, b=1.0, c=Inf] : [0.0, 0.0, 0.0]
|    Feature: 3 - [a=0.5, b=0.75, c=1.0]
|    |    Feature: 2 - [a=-Inf, b=0.0833, c=0.2917] : [0.0, 0.4305, 0.5695]
|    |    Feature: 2 - [a=0.0833, b=0.2917, c=0.5]
|    |    |    Feature: 1 - [a=-Inf, b=0.0278, c=0.2708] : [0.0, 0.0, 0.0]
|    |    |    Feature: 1 - [a=0.0278, b=0.2708, c=0.5139] : [0.0, 0.9114, 0.0886]
|    |    |    Feature: 1 - [a=0.2708, b=0.5139, c=0.7569] : [0.0, 0.5806, 0.4194]
|    |    |    Feature: 1 - [a=0.5139, b=0.7569, c=1.0] : [0.0, 0.7109, 0.2891]
|    |    |    Feature: 1 - [a=0.7569, b=1.0, c=Inf] : [0.0, 0.0, 1.0]
|    |    Feature: 2 - [a=0.2917, b=0.5, c=0.7083]
|    |    |    Feature: 1 - [a=-Inf, b=0.0278, c=0.2708] : [0.0, 0.0, 0.0]
|    |    |    Feature: 1 - [a=0.0278, b=0.2708, c=0.5139] : [0.0, 0.907, 0.093]
|    |    |    Feature: 1 - [a=0.2708, b=0.5139, c=0.7569] : [0.0, 0.7816, 0.2184]
|    |    |    Feature: 1 - [a=0.5139, b=0.7569, c=1.0] : [0.0, 0.7367, 0.2633]
|    |    |    Feature: 1 - [a=0.7569, b=1.0, c=Inf] : [0.0, 0.0, 1.0]
|    |    Feature: 2 - [a=0.5, b=0.7083, c=0.9167] : [0.0, 1.0, 0.0]
|    |    Feature: 2 - [a=0.7083, b=0.9167, c=Inf] : [0.0, 0.0, 0.0]
|    Feature: 3 - [a=0.75, b=1.0, c=Inf] : [0.0, 0.0, 1.0]
Feature: 4 - [a=0.5, b=0.75, c=1.0]
|    Feature: 3 - [a=-Inf, b=0.0, c=0.25] : [0.0, 0.0, 0.0]
|    Feature: 3 - [a=0.0, b=0.25, c=0.5] : [0.0, 1.0, 0.0]
|    Feature: 3 - [a=0.25, b=0.5, c=0.75]
|    |    Feature: 2 - [a=-Inf, b=0.0833, c=0.2917] : [0.0, 0.4549, 0.5451]
|    |    Feature: 2 - [a=0.0833, b=0.2917, c=0.5]
|    |    |    Feature: 1 - [a=-Inf, b=0.0278, c=0.2708] : [0.0, 1.0, 0.0]
|    |    |    Feature: 1 - [a=0.0278, b=0.2708, c=0.5139] : [0.0, 0.4906, 0.5094]
|    |    |    Feature: 1 - [a=0.2708, b=0.5139, c=0.7569] : [0.0, 0.2278, 0.7722]
|    |    |    Feature: 1 - [a=0.5139, b=0.7569, c=1.0] : [0.0, 0.6238, 0.3762]
|    |    |    Feature: 1 - [a=0.7569, b=1.0, c=Inf] : [0.0, 0.0, 0.0]
|    |    Feature: 2 - [a=0.2917, b=0.5, c=0.7083]
|    |    |    Feature: 1 - [a=-Inf, b=0.0278, c=0.2708] : [0.0, 0.0, 0.0]
|    |    |    Feature: 1 - [a=0.0278, b=0.2708, c=0.5139] : [0.0, 0.7121, 0.2879]
```

```
|    |    |    Feature: 1 - [a=0.2708, b=0.5139, c=0.7569] : [0.0, 0.5752, 0.4248]
|    |    |    Feature: 1 - [a=0.5139, b=0.7569, c=1.0] : [0.0, 0.7958, 0.2042]
|    |    |    Feature: 1 - [a=0.7569, b=1.0, c=Inf] : [0.0, 0.0, 0.0]
|    |    Feature: 2 - [a=0.5, b=0.7083, c=0.9167] : [0.0, 0.956, 0.044]
|    |    Feature: 2 - [a=0.7083, b=0.9167, c=Inf] : [0.0, 0.0, 0.0]
|    Feature: 3 - [a=0.5, b=0.75, c=1.0]
|    |    Feature: 1 - [a=-Inf, b=0.0278, c=0.2708] : [0.0, 0.0, 0.0]
|    |    Feature: 1 - [a=0.0278, b=0.2708, c=0.5139] : [0.0, 0.241, 0.759]
|    |    Feature: 1 - [a=0.2708, b=0.5139, c=0.7569] : [0.0, 0.1731, 0.8269]
|    |    Feature: 1 - [a=0.5139, b=0.7569, c=1.0]
|    |    |    Feature: 1 - [a=-Inf, b=0.0278, c=0.2708] : [0.0, 0.0, 0.0]
|    |    |    Feature: 1 - [a=0.0278, b=0.2708, c=0.5139] : [0.0, 0.0, 0.0]
|    |    |    Feature: 1 - [a=0.2708, b=0.5139, c=0.7569] : [0.0, 0.1803, 0.8197]
|    |    |    Feature: 1 - [a=0.5139, b=0.7569, c=1.0] : [0.0, 0.1533, 0.8467]
|    |    |    Feature: 1 - [a=0.7569, b=1.0, c=Inf] : [0.0, 0.0, 1.0]
|    |    Feature: 1 - [a=0.7569, b=1.0, c=Inf] : [0.0, 0.0, 1.0]
|    Feature: 3 - [a=0.75, b=1.0, c=Inf] : [0.0, 0.0, 1.0]
Feature: 4 - [a=0.75, b=1.0, c=Inf] : [0.0, 0.0, 1.0]
```

Listing 16: Fuzzy Multi-way Decision Tree.

1.13 `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 partwas 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:

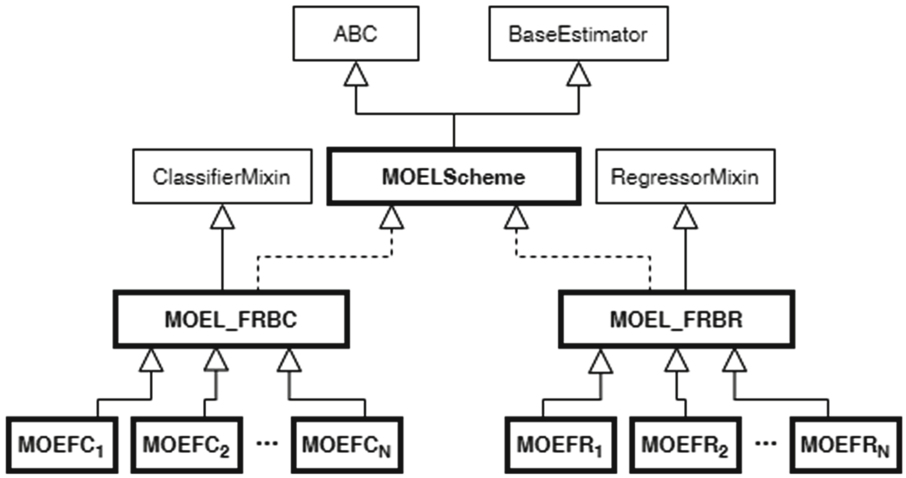


Figure 7: UML class diagram describing the class hierarchy of SK-MOEFS.

A MOELScheme represents a general multi-objective 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 MOEFSs, 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 Classifier-Mixin 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.**

1.14 `moea.jl`

1.15 `example.jl`

1.16 `frbs.jl`

1.17 `rcs.jl`

1.18 `toolbox.jl`

1.19 `test.jl`

2 Julia Precompilation

3 Conclusions

4 Future work

Possible future works may focus on

- use a machine-learning framework specially-designed for Julia: `MLJ` (Machine Learning in Julia), by the Alan Turing Institute[15], is a toolbox written in Julia providing a common interface and meta-algorithms for selecting, tuning, evaluating, composing and comparing over 160 machine learning models written in Julia and other languages;

References

[1] Michela Antonelli, Pietro Ducange, Beatrice Lazzerini, and Marcelloni Francesco. Multi-objective evolutionary design of granular rule-based classifiers. 2016.

[2] 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.

[3] Eva Armengol and Àngel García-Cerdaña. 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.

[4] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

[15] `https://www.turing.ac.uk/`

[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.