BETA Bureau d'économie théorique et appliquée

AgroParisTech
Talents d'une planète soutenable

BETA Seminar - 20 February 2024

julia and BetaML, the Machine Learning toolkit of the BETA UMR

Antonello Lobianco[1]

(1) AgroParisTech, UMR BETA, 54100 Nancy

https://bit.ly/BetaML https://bit.ly/spmlj

BetaML

BETA Bureau d'économie théorique et appliquée

AgroParisTech
Talents d'une planète soutenable

Today's presentation

(a) Context: Machine Learning in Julia

(b) The Beta Machine Learning Toolkit

- objectives, what's covered
- main algorithms and models
- API

(c) Still time for some examples ?

What is not....

- Detailed description of Julia characteristics (ask on discourse.julialang.org) or BetaML algorithms
- Discussion of a new cool ML algorithm
- Workshop on using a specific ML algorithm/workflow with BetaML (see the tutorials in the doc)

A comparative timeline of R, Python, Julia, LLVM

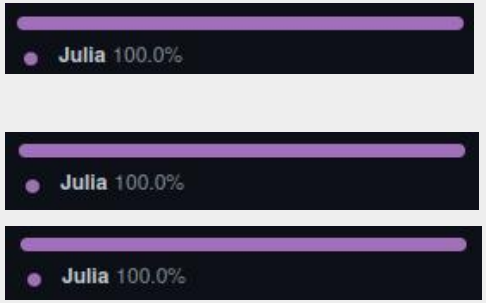
- 1991: Python announced
- 1993: R announced
- 1994: Python reaches v1.0
- 2000: R reaches v1.0
- 2003: LLVM announced (v1.0)
- 2012: Julia announced by a MIT team to solve the "two languages problem"
- 2018: Julia reaches v1.0

JuPyteR notebooks

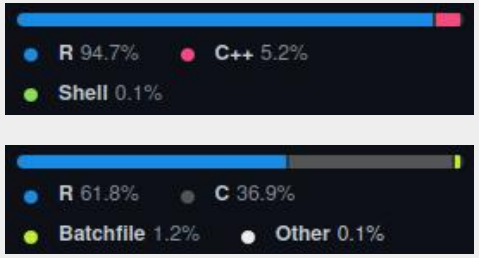
JIT compiled
- R: `compiler` library (JIT default in 3.4)
- Python: Numba, PyPy, ...
- Julia: natively, with type inference

Julia: users and developers speak the same lang

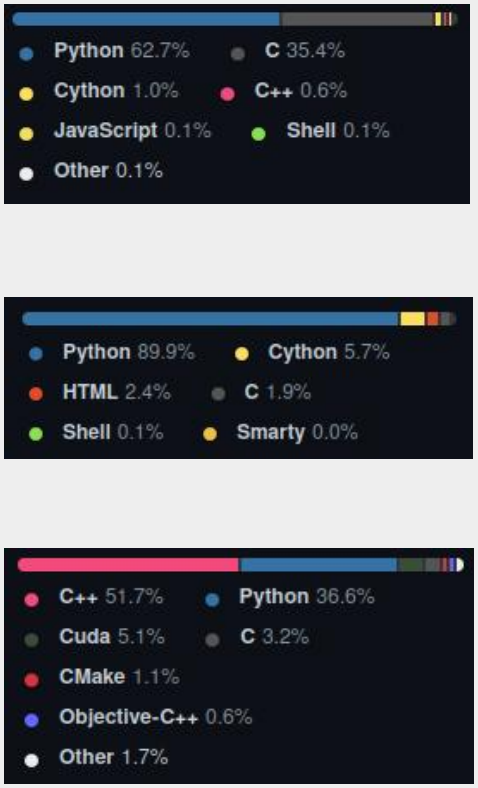
- Julia libraries
  - DataFrames:
  - Flux:
  - BetaML:



- R libraries
  - Dplyr:
  - data.table:



- Python libraries:
  - Numpy:
  - Pandas:
  - Pytorch:



Machine learning ecosystem in Julia

Category	Packages
ML toolkits/pipelines	MLJ.jl, ScikitLearn.jl, AutoMLPipeline.jl
Neural Networks	Flux.jl/Lux.it, Knet.jl
Decision Trees	DecisionTree.jl
Clustering	Clustering.jl, GaussianMixtures.jl
Missing imputation	Impute.jl

- Also wrappers for non-Julia libraries, like TensorFlow, Keras or PyTorch, but minimally maintained.
- Main organisation: packages for specific ML algorithms and packages to manage a ML Pipeline (plus packages for datasets)
- ML "pipeline" packages *less needed* than in other languages
- Still unconvenient for the "casual" ML user

BETA Bureau d'économie théorique et appliquée

AgroParisTech
Talents d'une planète soutenable

BetaML

- algorithms repository and ML workflow in a single library
- biased toward easy to use and accuracy rather than computational optimization
- consistent API to to provide an efficient and unified syntax to make machine learning accessible to non-machine learning experts, as well as to facilitate and popularize its use among several industries.

```
mod = Model([optional hyperparameters])
fit!(mod,x,[y])
ỹ   = predict(mod,[xnew])
x̃   = inverse_predict(mod,ỹ)
```

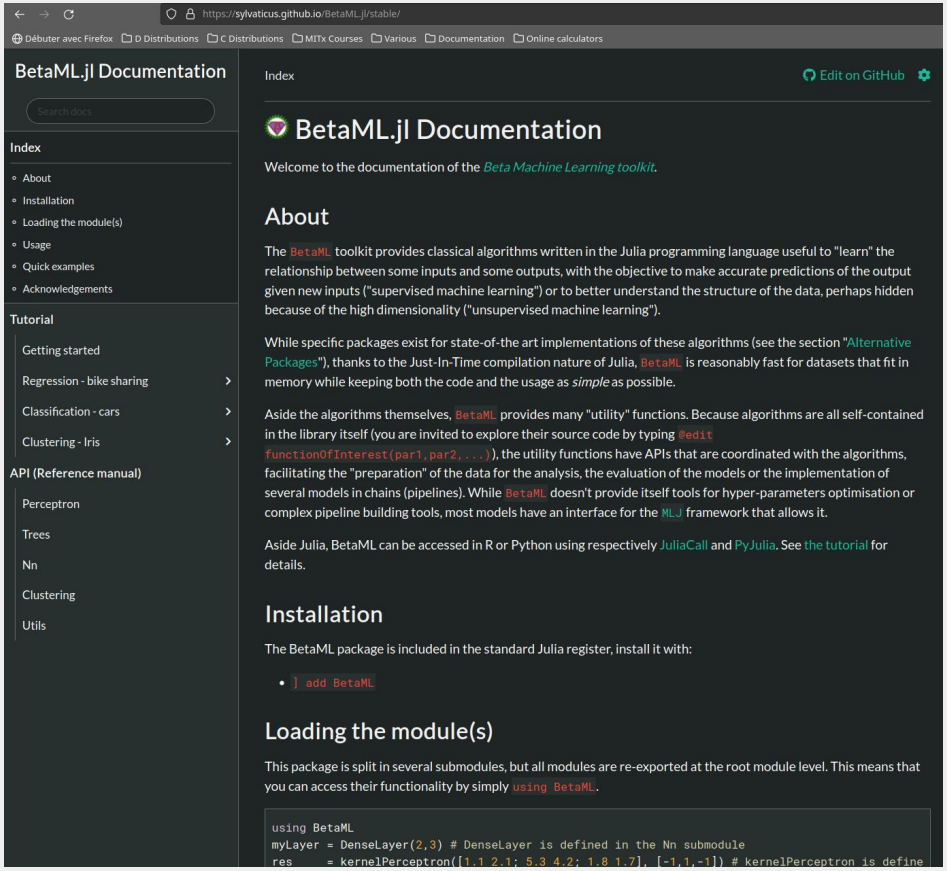
```
hyperparameters(mod)
options(mod)
parameters(mod)
info(mod)
```

ML algorithms:
- tree based
- neural networks
- mixture models / EM

ML models:
- Decision trees/ random forest
- Feedforward/ conv neural networks
- Clustering (hard/soft)
- Missing data imputers
- Dim reductions (PCA, Autoencoders)

ML workflow:
- data processing (encoding, permutations, partition, scaling, cross-validation)
- samplers (KFold, batch)
- error and distance measures (mean relative error, squared cost, crossEntropy, ConfusionMatrix, ...)

Fully tested and documented




```
help?> crossValidation
search: crossValidation

  crossValidation(f,data,sampler;dims,verbosity,returnStatistics)

  Perform crossValidation according to sampler rule by calling the function f and collecting its output

  Parameters
  ==========

  • f: The user-defined function that consume the specific train and validation data and return somehting (often the
    associated validation error). See later

  • data: A single n-dimenasional array or a vector of them (e.g. X,Y), depending on the tasks required by f.

  • sampler: An istance of a AbstractDataSampler, defining the "rules" for sampling at each iteration. [def:
    KFold(nSplits=5,nRepeats=1,shuffle=true,rng=Random.GLOBAL_RNG) ]

  • dims: The dimension over performing the crossValidation i.e. the dimension containing the observations [def: 1]

  • verbosity: The verbosity to print information during each iteration (this can also be printed in the f function)
    [def: STD]

  • returnStatistics: Wheter crossValidation should return the statistics of the output of f (mean and standard
    deviation) or the whole outputs [def: true].

  Notes
  =====

  crossValidation works by calling the function f, defined by the user, passing to it the tuple trainData, valData and rng
  and collecting the result of the function f. The specific method for which trainData, and valData are selected at each
  iteration depends on the specific sampler, whith a single 5 k-fold rule being the default.

  This approach is very flexible because the specific model to employ or the metric to use is left within the
  user-provided function. The only thing that crossValidation does is provide the model defined in the function f with the
  opportune data (and the random number generator).

  Input of the user-provided function trainData and valData are both themselves tuples. In supervised models,
  crossValidations data should be a tuple of (X,Y) and trainData and valData will be equivalent to (xtrain, ytrain) and
  (xval, yval). In unsupervised models data is a single array, but the training and validation data should still need to
  be accessed as trainData[1] and valData[1]. Output of the user-provided function The user-defined function can return
  whatever. However, if returnStatistics is left on its default true value the user-defined function must return a single
  scalar (e.g. some error measure) so that the mean and the standard deviation are returned.

  Note that crossValidation can beconveniently be employed using the do syntax, as Julia automatically rewrite
  crossValidation(data,...) trainData,valData,rng ...user defined body... end as crossValidation(f(trainData,valData,rng
  ), data,...)

  Example
  =======

julia> X = [11:19 21:29 31:39 41:49 51:59 61:69];
julia> Y = [1:9;];
julia> sampler = KFold(nSplits=3);
julia> (μ,σ) = crossValidation([X,Y],sampler) do trainData,valData,rng
                 (xtrain,ytrain) = trainData; (xval,yval) = valData
                 trainedModel    = buildForest(xtrain,ytrain,30)
                 predictions     = predict(trainedModel,xval)
                 ϵ               = meanRelError(predictions,yval,normRec=false)
                 return ϵ
               end
(0.3202242202242202, 0.04307662219315022)
```

BETA Bureau d'économie théorique et appliquée

AgroParisTech
Talents d'une planète soutenable

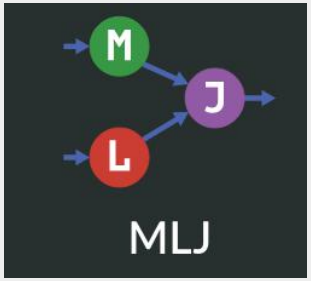
Accessible in Julia, R or Python

```
> install.packages("JuliaCall")
> library(JuliaCall)
> julia_setup(installJulia = FALSE)
> library(datasets)
> X <- as.matrix(sapply(iris[,1:4], as.numeric))
> y <- sapply(iris[,5], as.integer)
> julia_eval("using BetaML")
> yencoded <- julia_call("integerEncoder",y)
> ids      <- julia_call("shuffle",1:length(y))
> Xs       <- X[ids,]
> ys       <- yencoded[ids]
> cOut     <- julia_call("kmeans",Xs,3L)
> y_hat    <- sapply(cOut[1],as.integer)[,]
> acc      <- julia_call("accuracy",y_hat,ys)
> acc
[1] 0.8933333
```

R

```
$ python3 -m pip install --user julia
>>> from julia import BetaML
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> X = iris.data[:, :4]
>>> y = iris.target + 1 # Julia arrays start from 1
# X and y are first converted to julia
# arrays and then the returned julia arrays
# are converted back to python Numpy arrays
>>> (Xs,ys) = BetaML.shuffle([X,y])
>>> cOut    = BetaML.kmeans(Xs,3)
>>> y_hat   = cOut[0]
>>> acc     = BetaML.accuracy(y_hat,ys)
>>> acc
0.8933333333333333
```

Python



BetaML tutorials

https://blog.devgenius.

Medium Search

Random Forests in Julia

Nic Coxen · Follow

Published in Dev Genius · 6 min read · Aug 19, 2

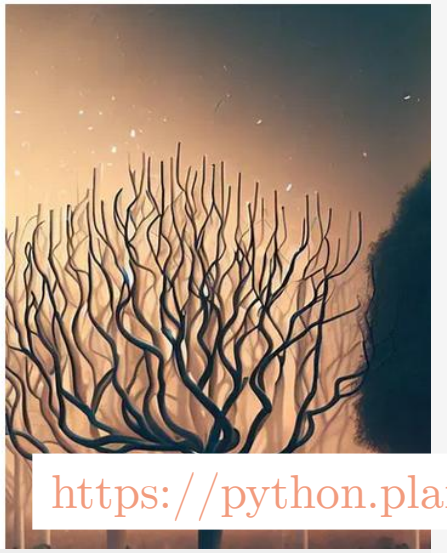
53

23

3

8

using MLJ and BetaML



julia Search...

```
using Pkg
Pkg.activate("titanic"
```

To add the packages we
`julia>` prompt, to change

```
add MLJ DataFrames Bet
```

It may take a few minutes

Tip. Next time you want t
new Julia session, you ca
lines above them.

When the `(titanic) pkg>`
versions that were install

- MLJ (machine learning
with models provided
model-generic tasks, such as hyperparameter optimization demonstrated at
Checkout these cheatsheets.
- BetaML: Provides the core decision algorithm we will be building for Titanic prediction.

Learn more about Julia package management here.

- **Loading the Model**

```
### Python
from sklearn.tree import DecisionTreeClassifier

### Julia
import BetaML
DecisionTreeClassifier = @load RandomForestClassifier pkg=BetaML
```

MLJ is called a *meta-package* because it has access to any maching learning model implemented in any Julia package provided that the package implements an `MLJModelInterface` for it (MLJ itself isn't really focused at implementing new models but it has access to over a hundred of them). For instance, here the decision tree we will be using with `MLJ` comes from the `BetaML` package.

- **Instantiating the Model**

```
# Python
model = DecisionTreeClassifier(max_depth=5, random_state=42)
# Julia
model = DecisionTreeClassifier(max_depth=5, rng=Random.Xoshiro(42))
```

https://python.plainenglish.io/from-python-to-julia-an-ultimate-guide-244fd3dc35c6

https://blog.devgenius.io/ran

BetaML

- New: small community

- No GPU support

- NN: slow convolutional layers and recurrent neural networks (RNN) not yet implemented (want to help?)

- Not optimised, not convenient with large data that doesn't fit in memory

Neural Networks

- Use manually provided derivates or auto-generated ones (using Zygote.jl)
- Provided layers: DenseLayer, DenseNoBiasLayer, VectorFunctionLayer, ScalarFunctionLayer, ReplicatorLayer, ReshaperLayer, PoolingLayer, ConvLayer, GroupedLayer.
    - other can be implemented by subclassing AbstractLayer and providing a few methods
- Provided optimization algorithms: SGD, ADAM
    - other can be implemented by subclassing OptimisationAlgorithm and providing a few methods

Example:

```
julia> using BetaML
julia> X = [1.8 2.5; 0.5 20.5; 0.6 18; 0.7 22.8; 0.4 31; 1.7 3.7];
julia> y = 2 .* X[:,1] .- X[:,2] .+ 3;
julia> layers = [DenseLayer(2,6),DenseLayer(6,6),DenseLayer(6,1)];
julia> m      = NeuralNetworkEstimator(layers=layers,opt_alg=ADAM(),epochs=3000,verbosity=LOW)
NeuralNetworkEstimator - A Feed-forward neural network (unfitted)
julia> ŷ      = fit!(m,X,y);
***
*** Training  for 3000 epochs with algorithm ADAM.
Training..       avg ϵ on (Epoch 1 Batch 1):     33.30063874270561
Training of 3000 epoch completed. Final epoch error: 34.61265465430473.
julia> hcat(y,ŷ)
6×2 Matrix{Float64}:
   4.1    4.11015
 -16.5  -16.5329
 -13.8  -13.8381
 -18.4  -18.3876
 -27.2  -27.1667
   2.7    2.70542
```

Trees

- decision trees and random forests, using the original Breiman approach
- (relatively) slow, but process everything: numerical data, ordinal data, categorical data,... missing data (!)
- require very little (if any) data processing
- both regression (continuous and int data) or classification (non-numerical data, int with "forceClassification" option)

Example:

```
julia> using BetaML
julia> X   = [1.8 missing "foo"; 0.5 20.5 "goo"; 0.6 18 "guu"; 0.7 22.8 "foo"; 0.4 31 "goo"; 1.7 3.7 "foo"];
julia> y   = ["a","b","b","b","b","a"];
julia> mod = RandomForestEstimator(n_trees=5)
RandomForestEstimator - A 5 trees Random Forest model (unfitted)
julia> ŷ   = fit!(mod,X,y) |> mode
6-element Vector{String}:
 "a"
 "b"
 "b"
 "b"
 "b"
 "a"
julia> println(mod)
RandomForestEstimator - A 5 trees Random Forest classifier (fitted on 6 records)
Dict{String, Any}("job_is_regression" => 0, "fitted_records" => 6, "avg_avg_depth" => 2.1333333333333333,
"oob_errors" => Inf, "avg_max_reached_depth" => 2.2, "xndims" => 3)
```

Some benchmarks - regression

- Bike sharing (prediction of bike sharing demand)
  - https://sylvaticus.github.io/BetaML.jl/stable/tutorials/Regression%20-%20bike%20sharing/betaml_tutorial_regression_sharingBikes.html#Neural-Networks

Model	Train rme	Test rme	Training time (ms)	Training mem (MB)
DT	0.1266	0.2223	26.5	58
RF	0.0651	0.2223	362	971
RF (DecisionTree.jl)	0.0312	0.3142	36	11
NN	0.0884	0.1761	1768	758
NN (Flux.jl)	0.0981	0.1618	1708	282

Some benchmarks - classification

- Cars country of origin (prediction of cars country of origin given cars characteristics)
  - https://sylvaticus.github.io/BetaML.jl/stable/tutorials/Classification%20-%20cars/betaml_tutorial_classification_cars.html

Model	Train acc	Test Acc	Training time (ms)	Training mem (MB)
RF	0.9969	0.8025	134	196
RF (DecisionTree.jl)	0.9969	0.7531	1.43	1.5
NN	0.895	0.765	11841	4311
NN (Flux.jl)	0.938	0.741	5665	1096

Clustering

- Hard and soft cluster algorithms: Kmeans, Kmedois (with personalisable distance metric), Gaussian Mixture Model (using Expectation/Maximisation algorithm, missing supported)
- Various "init" strategies (rand, grid, given..)

GMM

- 3 mixtures (Spherical, Diagonal and Full Gaussian), other implementable by subclassing AbstractMixture

- Various metrics to "judge" the cluster quality: BIC, AIC, silhouette

Example:

```
julia> using BetaML
julia> X = [1.1 10.1; 0.9 9.8; 10.0 1.1; 12.1 0.8; 0.8 9.8];
julia> mod = KMedoidsClusterer(n_classes=2);
julia> classes = fit!(mod,X)
5-element Vector{Int64}:
 2, 2, 1, 1, 2
julia> info(mod)
Dict{String, Any} with 3 entries:
  "fitted_records"       => 5
  "av_distance_last_fit" => 0.516375
  "xndims"               => 2
julia> s_scores = silhouette(pairwise(X),classes)
5-element Vector{Float64}:
 0.9709991332484983
 0.9829156361401824
 0.8321263686198753
 0.8527496229549888
 0.9806603535501258
```

BETA Bureau d'économie théorique et appliquée

AgroParisTech
Talents d'une planète soutenable

Some benchmarks – Clustering

Row	mName	avgAccuracy	stdAccuracy
	String	Float64	Float64
1	kMeansG	0.892	0.015
2	kMeansR	0.835	0.098
3	kMeansS	0.825	0.134
4	kMedoidsG	0.897	0.014
5	kMedoidsR	0.817	0.136
6	kMedoidsS	0.851	0.124
7	gmmSpher	0.894	0.015
8	gmmDiag	0.918	0.019
9	gmmFull	0.974	0.027
10	kMeans (Clustering.jl)	0.871	0.09
11	gmmDiag (GaussianMixtures.jl)	0.881	0.1
12	gmmFull (GaussianMixtures.jl)	0.909	0.138

Missing imputation

- Several imputers available: SimpleImputer, GaussianMixtureImputer, RandomForestImputer, GeneralImputer
- "missing" can have several interpretation (i.e. reccomendation systems, collaborative filtering)
- several imputers can provide multiple imputations and some can provide online training

Example:

```
julia> using BetaML
julia> X = [1.4 2.5 "a"; missing 20.5 "b"; 0.6 18 missing; 0.7 22.8 "b"; 0.4 missing "b"; 1.6 3.7 "a"];
julia> mod = GeneralImputer(recursive_passages=2, multiple_imputations=2, fit_function=BetaML.fit!);
julia> mX_full = fit!(mod,X);
** Processing imputation 1
** Processing imputation 2
julia> mX_full[1]
6×3 Matrix{Any}:
 1.4        2.5      "a"
 0.585667  20.5      "b"
 0.6       18        "b"
 0.7       22.8      "b"
 0.4       21.2778   "b"
 1.6        3.7      "a"
julia> mX_full[2]
6×3 Matrix{Any}:
 1.4        2.5     "a"
 0.606333  20.5     "b"
 0.6       18       "b"
 0.7       22.8     "b"
 0.4       20.486   "b"
 1.6        3.7     "a"
```

Some benchmarks – Missing Imputation

Sawnwood export quantities per country, year			
Model	seconds	l-2 norm	Mean rel. err
FeatureMeans	0.00	618	1.00
Mice_PMM	0.31	514	0.63
Mice_RF	0.54	460	0.59
BetaML_GMM	0.00	433	0.59
BetaML_RF	0.44	371	0.37
Sawnwood export prices per country,year			
Model	seconds	l-2 norm	Mean rel. err
FeatureMeans	0.00	889	0.37
Mice_PMM	0.27	761	0.32
Mice_RF	0.47	943	0.33
BetaML_GMM	0.00	824	0.34
BetaML_RF	0.46	596	0.23

Dimensionality reduction

- Linear (PCA) and non-linear (AutoEncoder)

Example:

```
julia> using BetaML
julia> xtrain        = [1 10 100; 1.1 15 120; 0.95 23 90;
0.99 17 120; 1.05 8 90; 1.1 12 95];
julia> mod           = PCAEncoder(max_unexplained_var=0.05);
julia> xtrain_reproj = fit!(mod,xtrain)
6×2 Matrix{Float64}:
 100.449    3.1783
 120.743    6.80764
  91.3551  16.8275
 120.878    8.80372
  90.3363   1.86179
  95.5965   5.51254
julia> info(mod)
Dict{String, Any} with 5 entries:
  "explained_var_by_dim" => [0.873992, 0.999989, 1.0]
  "fitted_records"       => 6
  "prop_explained_var"   => 0.999989
  "retained_dims"        => 2
  "xndims"               => 3
```

Dimensionality reduction

```
julia> using BetaML
julia> x = [0.12 0.31 0.29 3.21 0.21;
            0.22 0.61 0.58 6.43 0.42;
            0.51 1.47 1.46 16.12 0.99;
            0.35 0.93 0.91 10.04 0.71;
            0.44 1.21 1.18 13.54 0.85];
julia> m    = AutoEncoder(encoded_size=1,epochs=400, verbosity=NONE);
julia> x_reduced = fit!(m,x)
5×1 Matrix{Float64}:
  -3.5483740608901186
  -6.90396890458868
 -17.06296512222304
 -10.688936344498398
 -14.35734756603212
julia> x̂ = inverse_predict(m,x_reduced)
5×5 Matrix{Float64}:
 0.0982406  0.110294  0.264047   3.35501  0.327228
 0.205628   0.470884  0.558655   6.51042  0.487416
 0.529785   1.56431   1.45762   16.067    0.971123
 0.3264     0.878264  0.893584  10.0709   0.667632
 0.443453   1.2731    1.2182    13.5218   0.842298

julia> info(m)["rme"]
0.020858783340281222

julia> hcat(x,x̂)
5×10 Matrix{Float64}:
 0.12  0.31  0.29   3.21  0.21  0.0982406  0.110294  0.264047   3.35501  0.327228
 0.22  0.61  0.58   6.43  0.42  0.205628   0.470884  0.558655   6.51042  0.487416
 0.51  1.47  1.46  16.12  0.99  0.529785   1.56431   1.45762   16.067    0.971123
 0.35  0.93  0.91  10.04  0.71  0.3264     0.878264  0.893584  10.0709   0.667632
 0.44  1.21  1.18  13.54  0.85  0.443453   1.2731    1.2182    13.5218   0.842298
```

Still time for some examples ?

If not $\rightarrow$ as exercises in my course *Introduction to Scientific Programming and Machine Learning in Julia*:

https://sylvaticus.github.io/SPMLJ/stable/

Contacts

Antonello Lobianco
BETA - Bureau d'Economie Théorique et Appliquée, AgroParisTech Nancy

Lobianco, A., (2021). BetaML: The Beta Machine Learning Toolkit, a self-contained repository of Machine Learning algorithms in Julia. Journal of Open Source Software, 6(60), 2849, https://doi.org/10.21105/joss.02849


https://www6.nancy.inra.fr/lef_eng/Member/Antonello-LOBIANCO

antonello.lobianco@agroparistech.fr

https://github.com/sylvaticus/BetaML.jl

merci ;-)