The Journal of Open Source Software

BetaML: The Beta Machine Learning Toolkit, a self-contained repository of Machine Learning algorithms in Julia

Antonello Lobianco*[1, 2, 3, 4, 5, 6]

1 Université de Lorraine **2** Université de Strasbourg **3** AgroParisTech **4** CNRS **5** INRAE **6** BETA

DOI: 10.21105/joss.02512

Software

- Review
- Repository
- Archive

Editor: Pending Editor

Submitted: 23 July 2020
Published: 23 July 2020

Summary

A serie of *machine learning* algorithms has been implemented and bundled in a single package for the Julia language. Currently, algorithms are available in the areas of classification (perceptron, kernel perceptron, pegasos), neural networks (feed-forward) and clustering (kmeans, kmenoids, EM, missing values attribution). Development of these algorithms started following the theoretical notes of the MOOC class "Machine Learning with Python: from Linear Models to Deep Learning" from MITx/edX.

This paper presents the motivations and the general approach of the package and gives an overview of its organisation. We refer the reader to the package documentation for instructions on how to use the various algorithms provided or to the MOOC notes available on GitHub (Lobianco, 2020) for their mathematical backgrounds.

Motivations and objectives

`BetaML` provides one of the simplest way to run ML algorithms in Julia. While many packages already implement specific ML algorithms in Julia, these are fragmented across different packages and often value performances more than usability. For example the popular Deep Learning library Flux (Mike Innes, 2018), while extremely performant and flexible, adopts some designing choices that for a beginner could appear odd, for example avoiding the neural network object from the training process, or requiring all parameters to be explicitly defined. In `BetaML` we made the choice to allow the user to experiment with the hyperparameters of the algorithms learning them one step at the time. Hence for most functions we provide reasonable default parameters that can be overridden when needed. For example, modelling, training and collecting predictions from a feed-forward artificial neural network with one hidden layer can be as simple as:

```
using BetaML.Nn
mynn   = buildNetwork([DenseLayer(nIn,nHidden),
                       DenseLayer(nHidden,nOut)],
                       squaredCost)
train!(mynn,xtrain,ytrain)
ytrain_est = predict(mynn,xtrain)
ytest_est  = predict(mynn,xtest)
```

*Corresponding author.

While much better results can be obtained (in general) by scaling the variables and/or tuning their activation functions, the training parameters or the optimisation algorithm, this code snippet already runs the model using common practices like random mini-batches.

Still BetaML offers a fair level of flexibility. As we didn't aim for heavy optimisation, we were able to keep the API (Application Programming Interface) both beginner-friendly and flexible. For example, one can implement its own neural network layer, optimisation algorithm, or mixture model, define the initial conditions in almost all the stochastic algorithms or specify its own distance metric in the clustering algorithms. To help beginners, many parameters and functions have pretty longer but more explicit names than usual. For example the Dense layer is a `DenseLayer`, the RBF kernel is `radialKernel`, etc.

A few packages try to provide a common framework of the ML ecosystem of Julia packages. Notably MLJ.jl (Blaom, Kiraly, Lienart, & Vollmer, 2019) builds up on existing ML specialised packages. While avoiding the problem of "reinventing the wheel", the wrapping level unintentionally introduces some complications for the end-user, like the need to load the models and learn MLJ-specific concepts as *model* or *machine*.

We chose instead to bundle the main ML algorithms directly within the package. This offers a complementary approach that we feel is more beginner-friendly.

We believe that the BetaML flexibility and simplicity, together with the efficiency and usability of a Just in Time compiled language like Julia and the convenience to have several ML algorithms and data-science utilities all in the same package, will support the needs of that community of students and researchers that, contrary to industrial practitioners or computer science specialists, don't necessarily need to work with very large datasets that don't fit in memory or algorithms that require distributed computation.

Package organisation

The BetaML toolkit is currently composed of 4 modules: `Utils` provides common data-science utility functions to be used in the other modules, `Perceptron` supplies linear and non-linear classifiers based on the classical Perceptron algorithm, `Nn` allows implementing and training artificial neural networks, and `Clustering` includes several clustering algorithms and missing value attribution / collaborative filtering algorithms based on clustering.

`Perceptron`, `Nn` and `Clustering` all import and re-export the `Utils` function, so the final users normally doesn't need to deal with `Utils`, but just with the module of interest.

The `Utils` module

The `Utils` module is intended to provide functionalities that are either: (a) used in other modules but are not strictly part of that specific module's logic (for example activation functions would be most likely used in neural networks, but could be of more general usage); (b) general methods that are used alongside the ML algorithms implemented in the other modules, e.g. to improve their predictions capabilities or (c) general methods to assess the goodness of fits of ML algorithms.

Concerning the fist category `Utils` provides "classical" activation functions (and their respective derivatives) like `relu`, `sigmoid`, `softmax`, but also more recent implementations like elu (Clevert, Unterthiner, & Hochreiter, 2015), celu (Barron, 2017), plu (Nicolae, 2018), softplus (Glorot, Bordes, & Bengio, 2011) and mish (Misra, 2019). Kernel functions (`radialKernel` - aka "KBF", `polynomialKernel`), distance metrics (`l1_distance` - aka "Manhattan", `l2_distance`, `l2²_distance`, `cosine_distance`), and functions typically used to improve numerical stability (`lse`) are also provided with the intention to be available in the different ML algorithms.

Often ML algorithms work better if the data is normalised or dimensions are reduced to those explaining the greatest extent of data variability. This is the purpose of the functions `scale` and `pca` respectively. `scale` scales the data to $\mu = 0$ and $\sigma = 1$, optionally skipping dimensions that don't need to be normalised (like categorical ones). The related function `getScaleFactors` saves the scaling factors so that inverse scaling (typically for the predictions of the ML algorithm) can be applied. `pca` performs Principal Component Analysis, where the user can specify the wanted dimensions or the maximum approximation error that he is willing to accept either *ex-ante* or *ex-post*, after having analysed the distribution of the explained variance by number of dimensions. Other "general support" functions provided are `oneHotEncoder` and `batch`.

Concerning the last category, several functions are provided to assess the goodness of fit of a single datapoint or of the whole dataset, whether the output of the ML algorithm is in R^n or categorical. Notably, `accuracy` provides categorical accuracy given a probabilistic prediction (as PMF) of a datapoint. finally, the Bayesian Information Criterion `bic` and Akaike Information Criterion `aic` functions can be used for regularisation.

The `Perceptron` module

It provides the classical Perceptron linear classifier, a *kernelised* version of it and "Pegasos" (Shalev-Shwartz, Singer, Srebro, & Cotter, 2011), a gradient-descent based implementation.

The basic Perceptron classifier is implemented in the `perceptron` function, where the user can provide the initial weights and retrieve both the final and the average parameters of the classifier. In `kernelPerceptron` the user can either pass one of the kernel implemented in `Utils` or implement its own kernel function. `pegasos` performs the classification using a basic stochastic descent method[1]. Finally `predict` predicts the binary label given the feature vector and the linear coefficients or the error distribution as obtained by the kernel Perceptron algorithm.

The `Nn` module

Artificial neural networks can be implemented using the functions provided by the `Nn` module. Currently only feed-forward networks for regression or classification tasks are fully provided, but more complex layers (convolutional, pooling, recursive,...) can be eventually defined and implemented directly by the user. The instantiation of the layers required by the network can be done indeed either using one of the layer provided (`DenseLayer`, `DenseNoBiasLayer` or `VectorFunctionLayer`, the latter one being a parameterless layer whose activation function, like `softMax`, is applied to the ensemble of the neurons rather than individually on each of them) or by creating a user-defined layer by subclassing the `Layer` type and implementing the functions `forward`, `backward`, `getParams`, `getGradient`, `setParams` and `size`.

While in the provided layers the computation of the derivatives for `backward` and `getParams` is coded manually[2], for complex user-defined layers the two functions can benefit of automatic differentiation packages like `Zygote`(Michael Innes, 2018), eventually wrapped in the function `autoJacobian` defined in `Utils`.

Once the layers are defined, the neural network is modelled by setting the layers in an array, giving the network a cost function (default to) and a name. The `show` function can be employ to print the structure of the network.

[1] We plan to generalise the Pegasos algorithm to use the optimisation algorithms implemented for neural networks.

[2] For the derivatives of the activation function the user can (a) provide one of the derivative functions defined in `Utils`, (b) implement it by himself, or (c) just leave the library use automatic differentiation (using Zygote) to compute it.

The training of the model is done with the highly parametrisable `train!` function. In a similar way than for the definition of the layers, one can use for training one of the "standard" optimisation algorithms provided (`SGD` and `ADAM`, Kingma & Ba (2014)), either using their default values or by fine-tuning their parameters, or by defining the optimisation algorithm by subclassing the `OptimisationAlgorithm` class and implementing the `singleUpdate!` and eventually `initOptAlg!` methods. Note that the `singleUpdate!` function provides the algorithm with quite a large set of information from the training process, allowing a wide class of optimisation algorithms to be implemented.

`Clustering` module

Both the classical `kmeans` and `kmedoids` algorithms are provided (with the difference being that the clusters "representatives" can be in any R^n point in `kmeans`, while are restricted to be one of the data point in `kmedoids`), where different measure metrics can be provided (either those defined in `Utils` or user-provided ones) as well as different initialisation strategies (`random`, `grid`, `shuffle` - randomly within the available points, `given`).

Alongside these "hard clustering" algorithms, the `Clustering` module provides `em`, an implementation of the Expectation-Maximisation algorithm to estimate a generative mixture model, with variance-free and variance-constrained Gaussian mixtures already provided (and again, one can write his own mixture by subclassing `Mixture` and implementing `initMixtures!`, `lpdf`, `updateParameters!` and `npar`) and with `kmeans` or `grid` initialisation strategy supported.

Notably the `em` algorithm would accept an input data whose values are missing in one or all dimensions (and in the former case learning would use only the available dimensions).

This, together with the probabilistic assignment nature of the em algorithm, allows it to be used as base for missing values assignment or even collaborative filtering/recommandation systems in the `predictMissing` function.

Acknowledgements

This work was supported by: (i) the French National Research Agency through the Laboratory of Excellence ARBRE, part of the "Investissements d'Avenir" Program (ANR 11 – LABX-0002-01), and the ORACLE project (ANR-10-CEPL-011); (ii) a grant overseen by Office National des Forêts through the Forêts pour Demain International Teaching and Research Chair.

References

Barron, J. T. (2017). Continuously differentiable exponential linear units. Retrieved from https://arxiv.org/abs/1704.07483https://arxiv.org/abs/1704.07483

Blaom, A., Kiraly, F., Lienart, T., & Vollmer, S. (2019). *MLJ, a machine learning framework for julia*. Zenodo. doi:10.5281/zenodo.3541505

Clevert, D.-A., Unterthiner, T., & Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (elus). Retrieved from https://arxiv.org/abs/1511.07289v5

Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep sparse rectifier neural networks. *Proceedings of Machine Learning Research*, *15*, 315–323. Retrieved from http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf

Innes, M. (2018). Flux: Elegant machine learning with julia. *Journal of Open Source Software*. doi:10.21105/joss.00602

Innes, M. (2018). Don't unroll adjoint: Differentiating ssa-form programs. Retrieved from https://arxiv.org/abs/1810.07951

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. Retrieved from https://arxiv.org/abs/1412.6980

Lobianco, A. (2020). Student's notes (2020 run) of the mitx 6.86x course. GitHub repository. Retrieved from https://github.com/sylvaticus/MITx_6.86x

Misra, D. (2019). Mish: A self regularized non-monotonic neural activation function. Retrieved from https://arxiv.org/abs/1908.08681

Nicolae, A. (2018). PLU: The piecewise linear unit activation function. Retrieved from http://arxiv.org/abs/1809.09534

Shalev-Shwartz, S., Singer, Y., Srebro, N., & Cotter, A. (2011). Pegasos: Primal estimated sub-gradient solver for svm. *Mathematical Programming*, *127*(1), 3–30. doi:10.1007/s10107-010-0420-4