The Journal of Open Source Software

MLJ: A Julia package for composable machine learning

Anthony D. Blaom[1, 2, 3], **Franz Kiraly**[3, 4], **Thibaut Lienart**[3], **Yiannis Simillides**[7], **Diego Arenas**[6], **and Sebastian J. Vollmer**[3, 5]

1 University of Auckland, New Zealand **2** New Zealand eScience Infrastructure, New Zealand **3** Alan Turing Institute, London, United Kingdom **4** University College London, United Kingdom **5** University of Warwick, United Kingdom **6** University of St Andrews, St Andrews, United Kingdom **7** Imperial College London, United Kingdom

DOI: 10.21105/joss.02704

Software

- Review
- Repository
- Archive

Editor: Yuan Tang

Reviewers:

- @degleris1
- @henrykironde

Submitted: 21 July 2020
Published: 07 November 2020

License
Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License (CC BY 4.0).

Introduction

Statistical modeling, and the building of complex modeling pipelines, is a cornerstone of modern data science. Most experienced data scientists rely on high-level open source modeling toolboxes - such as sckit-learn (Buitinck et al., 2013; Pedregosa et al., 2011) (Python); Weka (Holmes et al., 1994) (Java); mlr (Bischl et al., 2016) and caret (Kuhn, 2008) (R) - for quick blueprinting, testing, and creation of deployment-ready models. They do this by providing a common interface to atomic components, from an ever-growing model zoo, and by providing the means to incorporate these into complex work-flows. Practitioners are wanting to build increasingly sophisticated composite models, as exemplified in the strategies of top contestants in machine learning competitions such as Kaggle.

MLJ (Machine Learning in Julia) (A. Blaom, 2020b) is a toolbox written in Julia that provides a common interface and meta-algorithms for selecting, tuning, evaluating, composing and comparing machine model implementations written in Julia and other languages. More broadly, the MLJ project hopes to bring cohesion and focus to a number of emerging and existing, but previously disconnected, machine learning algorithms and tools of high quality, written in Julia. A welcome corollary of this activity will be increased cohesion and synergy within the talent-rich communities developing these tools.

In addition to other novelties outlined below, MLJ aims to provide first-in-class model composition capabilities. Guiding goals of the MLJ project have been usability, interoperability, extensibility, code transparency, and reproducibility.

Why Julia?

Nowadays, even technically competent users of scientific software will prototype solutions using a high-level language such as python, R, or MATLAB. However, to achieve satisfactory performance, such code typically wraps performance critical algorithms written in a second low-level language, such as C or FORTRAN. Through its use of an extensible, hierarchical system of abstract types, just-in-time compilation, and by replacing object-orientation with multiple dispatch, Julia solves the ubiquitous "two language problem" (Bezanson et al., 2017). With less technical programming knowledge, experts in a domain of application can get under the hood of machine learning software to broaden its applicability, and innovation can be accelerated through a dramatically reduced software development cycle.

As an example of the productivity boost provided by the single-language paradigm, we cite the DifferentialEquations.jl package (Rackauckas & Nie, 2017), which, in a few short years of development by a small team of domain experts, became the best package in its class (Rackauckas, 2018).

Another major advantage of a single-language solution is the ability to automatically differentiate (AD) functions from their code representations. The Flux.jl package (Innes, 2018), for example, already makes use of AD to allow unparalleled flexibility in neural network design.

As a new language, Julia is high-performance computing-ready, and its superlative metaprogramming features allow developers to create domain-specific syntax for user interaction.

Novelties

Composability. In line with current trends in "auto-ML", MLJ's design is largely predicated on the importance of model composability. Composite models share all the behavior of regular models, constructed using a new flexible "learning networks" syntax. Unlike the toolboxes cited above, MLJ's composition syntax is flexible enough to define stacked models, with out-of-sample predictions for the base learners, as well as more routine linear pipelines, which can include target transformations that are learned. As in mlr, hyper-parameter tuning is implemented as a model wrapper.

A unified approach to probabilistic predictions. In MLJ, probabilistic prediction is treated as a first class feature, leveraging Julia's type system. In particular, unnecessary case-distinctions, and ambiguous conventions regarding the representation of probabilities, are avoided.

Scientific types To help users focus less on data representation (e.g., `Float32`, `DataFrame`) and more on the intended *purpose* or *interpretation* of data, MLJ articulates model data requirements using *scientific types* (Anthony Blaom and collaborators, 2019), such as "continuous", "ordered factor" or "table".

Connecting models directly to arbitrary data containers. A user can connect models directly to tabular data in a manifold of in-memory and out-of-memory formats by using a universal table interface provided by the Tables.jl package (Quinn, 2020).

Finding the right model. A model registry gives the user access to model metadata without the need to actually load code defining the model implementation. This metadata includes the model's data requirements, for example, as well as a load path to enable MLJ to locate the model interface code. Users can readily match models to machine learning tasks, facilitating searches for an optimal model, a search that can be readily automated.

Tracking classes of categorical variables. Finally, with the help of scientific types and the CategoricalArrays.jl package (Bouchet-Valat, 2014), users are guided to create safe representations of categorical data, in which the complete pool of possible classes is embedded in the data representation, and classifiers preserve this information when making predictions. This avoids a pain-point familiar in frameworks that simply recast categorical data using integers: evaluating a classifier on the test target, only to find the test data includes classes not seen in the training data. Preservation of the original labels for these classes also facilitates exploratory data analysis and interpretability.

Scientific types

A scientific type is an ordinary Julia type (generally without instances) reserved for indicating how some data should be interpreted. Some of these types are shown in Figure 1.

The Journal of Open Source Software

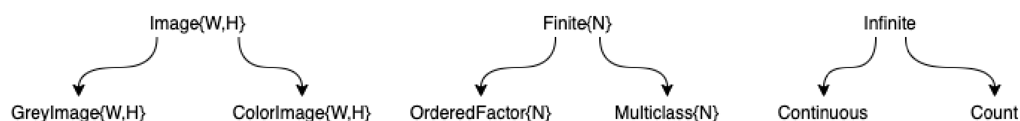

Figure 1: Part of the scientific type hierarchy.

To the scientific types, MLJ adds a specific *convention* specifying a scientific type for every Julia object. The convention is expressed through a single method `scitype`. So, for example, `scitype(x)` returns `Continuous` whenever the type of `x` is a subtype of Julia's `AbstractFloat` type, as in `scitype(3.14) == Continuous`. A tabular data structure satisfying the Tables.jl interface, will always have type `Table{K}`, where the type parameter `K` is the union of all column scientific types. A `coerce` method recasts machine types to have the desired scientific type (interpretation), and a `schema` method summarizes the machine and scientific types of tabular data.

Since scientific types are also Julia types, Julia's advanced type system means scientific types can be organized in a type hierarchy. It is straightforward to check the compatibility of data with a model's scientific requirements and methods can be dispatched on scientific type just as they would on ordinary types.

Flexible and compact work-flows for performance evaluation and tuning

To evaluate the performance of some `model` object (specifying the hyper-parameters of some supervised learning algorithm) using some specified `resampling` strategy, and measured against some battery of performance `measures`, one runs:

```
evaluate(model, X, y,
         resampling=CV(nfolds=6),
         measures=[L2HingeLoss(), BrierScore()])
```

which has (truncated) output

`measure`	`measurement`	`per_fold`
L2HingeLoss	1.4	[0.485, 1.58, 2.06, 1.09, 2.18, 1.03]
BrierScore{UnivariateFinite}	-0.702	[-0.242, -0.788, -1.03, -0.545, -1.09, -0.514]

As in mlr, hyper-parameter optimization is realized as a model wrapper, which transforms a base model into a "self-tuning" version of that model. That is, tuning is is abstractly specified before being executed. This allows tuning to be integrated into work-flows (learning networks) in multiple ways. A well-documented tuning interface (A. Blaom & collaborators, 2020) allows developers to easily extend available hyper-parameter tuning strategies.

We now give an example of syntax for wrapping a model called `forest_model` in a random search tuning strategy, using cross-validation, and optimizing the mean square loss. The `model` in this case is a composite model with an ordinary hyper-parameter called `bagging_fraction` and a *nested* hyper-parameter `atom.n_subfeatures` (where `atom` is another model). The first two lines of code define ranges for these parameters.

The Journal of Open Source Software

```
r1 = range(forest_model, :(atom.n_subfeatures), lower=1, upper=9)
r2 = range(forest_model, :bagging_fraction, lower=0.4, upper=1.0)
self_tuning_forest_model = TunedModel(model=forest_model,
                                      tuning=RandomSearch(),
                                      resampling=CV(nfolds=6),
                                      range=[r1, r2],
                                      measure=LPDistLoss(2),
                                      n=25)
```

In this random search example, default priors are assigned to each hyper-parameter, but options exist to customize these. Both resampling and tuning have options for parallelization; Julia has first class support for both distributed and multi-threaded parallelism.

A unified approach to probabilistic predictions and their evaluation

MLJ puts probabilistic models and deterministic models on equal footing. Unlike most most frameworks, a supervised model is either *probabilistic* - meaning it's `predict` method returns a distribution object - *or* it is *deterministic* - meaning it returns objects of the same scientific type as the training observations. To use a probabilistic model to make deterministic predictions one can wrap the model in a pipeline with an appropriate post-processing function, or use additional `predict_mean`, `predict_median`, `predict_mode` methods to deal with the common use-cases.

A "distribution" object returned by a probabilistic predictor is one that can be sampled (using Julia's `rand` method) and queried for properties. Where possible the object is in fact a `Distribution` object from the Distributions.jl package (Lin et al., 2020), for which an additional `pdf` method for evaluating the distribution's probability density or mass function will be implemented, in addition to `mode`, `mean` and `median` methods (allowing MLJ's fallbacks for `predict_mean`, etc, to work).

One important distribution *not* provided by Distributions.jl is a distribution for finite sample spaces with *labeled* elements (called `UnivariateFinite`) which additionally tracks all possible classes of the categorical variable it is modeling, and not just those observed in training data.

By predicting distributions, instead of raw probabilities or parameters, MLJ avoids a common pain point, namely deciding and agreeing upon a convention about how these should be represented: Should a binary classifier predict one probability or two? Are we using the standard deviation or the variance here? What's the protocol for deciding the order of (unordered) classes? How should multi-target predictions be combined?, etc.

A case-in-point concerns performance measures (metrics) for probabilistic models, such as cross-entropy and Brier loss. All built-in probabilistic measures provided by MLJ are passed a distribution in their prediction slot.

For an overview on probabilistic supervised learning we refer to (Gressmann et al., 2018).

Model interfaces

In MLJ a *model* is just a struct storing the hyper-parameters associated with some learning algorithm suggested by the struct name (e.g., `DecisionTreeClassifier`) and that is all. MLJ provides a basic *model interface*, to be implemented by new machine learning models, which is functional in style, for simplicity and maximal flexibility. In addition to a `fit` and

optional `update` method, one implements one or more operations, such as `predict`, `transform` and `inverse_transform`, acting on the learned parameters returned by `fit`.

The optional `update` method allows one to avoid unnecessary repetition of code execution (warm restart). The three main use-cases are:

- **Iterative models.** If the only change to a random forest model is an increase in the number of trees by ten, for example, then not all trees need to be retrained; only ten new trees need to be trained.
- **Data preprocessing.** Avoid overheads associated with data preprocessing, such as coercion of data into an algorithm-specific type.
- **Smart training of composite models.** When tuning a simple transformer-predictor pipeline model using a holdout set, for example, it is unnecessary to retrain the transformer if only the predictor hyper-parameters change. MLJ implements "smart" retraining of composite models like this by defining appropriate `update` methods.

In the future MLJ will add an `update_data` method to support models that can carry out on-line learning.

Presently, the general MLJ user is encouraged to interact through a *machine interface* which sits on top of the model interface. This makes some work-flows more convenient but, more significantly, introduces a syntax which is more natural in the context of model composition (see below). A *machine* is a mutable struct that binds a model to data at construction, as in `mach = machine(model, data)`, and which stores learned parameters after the user calls `fit!(mach, rows=...)`. To retrain with new hyper-parameters, the user can mutate `model` and repeat the `fit!` call.

The operations `predict`, `transform`, etc are overloaded for machines, which is how the user typically uses them, as in the call `predict(mach, Xnew)`.

Flexible model composition

Several limitations surrounding model composition are increasingly evident to users of the dominant machine learning software platforms. The basic model composition interfaces provided by the toolboxes mentioned in the Introduction all share one or more of the following shortcomings, which do not exist in MLJ:

- Composite models do not inherit all the behavior of ordinary models.
- Composition is limited to linear (non-branching) pipelines.
- Supervised components in a linear pipeline can only occur at the end of the pipeline.
- Only static (unlearned) target transformations/inverse transformations are supported.
- Hyper-parameters in homogeneous model ensembles cannot be coupled.
- Model stacking, with out-of-sample predictions for base learners, cannot be implemented.
- Hyper-parameters and/or learned parameters of component models are not easily inspected or manipulated (in tuning algorithms, for example)
- Composite models cannot implement multiple operations, for example, both a `predict` and `transform` method (as in clustering models) or both a `transform` and `inverse_transform` method.

We now sketch MLJ's composition API, referring the reader to (A. Blaom, 2020a) for technical details, and to the MLJ documentation (A. Blaom, 2020b; Lienart et al., 2020) for examples that will clarify how the composition syntax works in practice.

Note that MLJ also provides "canned" model composition for common use cases, such as non-branching pipelines and homogeneous ensembles, which are not discussed further here.

Specifying a new composite model type is in two steps, *prototyping* and *export*.

Prototyping

In prototyping the user defines a so-called *learning network*, by effectively writing down the same code she would use if composing the models "by hand". She does this using the machine syntax, with which she will already be familiar, from the basic `fit!`/`predict` work-flow for single models. There is no need for the user to provide production training data in this process. A dummy data set suffices, for the purposes of testing the learning network as it is built.

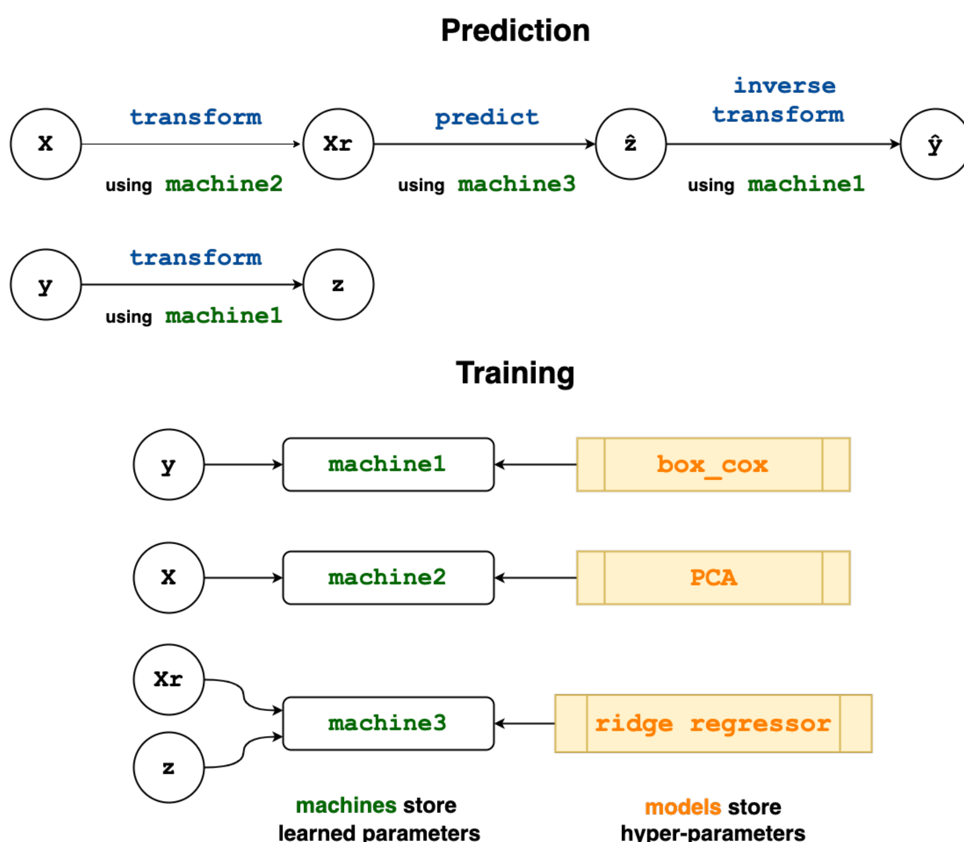

Figure 2: Specifying prediction and training flows in a simple learning network. The network shown combines a ridge regressor with a learned target transformation (Box Cox).

The upper panel of Figure Figure 2 illustrates a simple learning network in which a continuous target `y` is "normalized" using a learned Box Cox transformation, producing `z`, while PCA dimension reduction is applied to some features `X`, to obtain `Xr`. A Ridge regressor, trained using data from `Xr` and `z`, is then applied to `Xr` to make a target prediction `ẑ`. To obtain a final prediction `ŷ`, we apply the *inverse* of the Box Cox transform, learned previously, to `ẑ`.

The lower "training" panel of the figure shows the three machines which will store the parameters learned in training - the Box Cox exponent and shift (`machine1`), the PCA projection

The Journal of Open Source Software

(machine2) and the ridge model coefficients and intercept (machine3). The diagram additionally indicates where machines should look for training data, and where to access model hyper-parameters (stored in box_cox, PCA and ridge_regressor).

The only syntactic difference between composing "by hand" and building a learning network is that the training data must be wrapped in "source nodes" (which can be empty if testing is not required) and the fit! calls can be omitted, as training is now lazy. Each data "variable" in the manual work-flow is now a node of a directed acyclic graph encoding the composite model architecture. Nodes are callable, with a node call triggering lazy evaluation of the predict, transform and other operations in the network. Instead of calling fit! on every machine, a single call to fit! on a *node* triggers training of all machines needed to call that node, in appropriate order. As mentioned earlier, training such a node is "smart" in the sense that hyper-parameter changes to a model only trigger retraining of necessary machines. So, for example, there is no need to retrain the Box Cox transformer in the preceding example if only the ridge regressor hyper-parameters have changed.

The syntax, then, for specifying the learning network shown Figure 2 looks like this:

```
X = source(X_dummy)        # or just source()
y = source(y_dummy)        # or just source()

machine1 = machine(box_cox, y)
z = transform(machine1, y)

machine2 = machine(PCA, X)
Xr = transform(machine2, X)

machine3 = machine(ridge_regressor, Xr, z)
ẑ = predict(machine3, Xr)

ŷ = inverse_transform(machine1, ẑ)

fit!(ŷ)  # to test training on the dummy data
ŷ()      # to test prediction on the dummy data
```

Note that the machine syntax is a mechanism allowing for multiple nodes to point to the same learned parameters of a model, as in the learned target transformation/inverse transformation above. They also allow multiple nodes to share the same model (hyper-parameters) as in homogeneous ensembles. And different nodes can be accessed during training and "prediction" modes of operation, as in stacking.

Export

In the second step of model composition, the learning network is "exported" as a new stand-alone composite model type, with the component models appearing in the learning network becoming default values for corresponding hyper-parameters of the composite. This new type (which is unattached to any particular data) can be instantiated and used just like any other MLJ model (tuned, evaluated, etc). Under the hood, training such a model builds a learning network, so that training is "smart". Defining a new composite model type requires generating and evaluating code, but this is readily implemented using Julia's meta-programming tools, i.e., executed by the user with a simple macro call.

Future directions

There are plans to: (i) grow the number of models; (ii) enhance core functionality, particularly around hyper-parameter optimization (A. Blaom & collaborators, 2020); and (iii) broaden scope, particularly around probabilistic programming models, time series, sparse data and natural language processing. A more comprehensive road map is linked from the MLJ repository (Blaom, 2019).

Acknowledgments

We acknowledge valuable conversations with Avik Sengupta, Mike Innes, mlr author Bernd Bischl, and IQVIA's Yaqub Alwan and Gwyn Jones. Seed funding for the MLJ project has been provided by the Alan Turing Institute's Tools, Practices and Systems programme, with special thanks to Dr James Hethering, its former Programme Director, and Katrina Payne. Mathematics for Real-World Systems Centre for Doctoral Training at the University of Warwick provided funding for students exploring the Julia ML ecosystem, who created an initial proof-of-concept.

Code contributors. D. Aluthge, D. Arenas, E. Barp, C. Bieganek, A. Blaom, G. Bohner, M. K. Borregaard, D. Buchaca, V. Churavy, H. Devereux, M. Giordano, J. Hoffimann, T. Lienart, M. Nook, Z. Nugent, S. Okon, P. Oleśkiewicz, J. Samaroo, A. Shridar, Y. Simillides, A. Stechemesser, S. Vollmer

References

Anthony Blaom and collaborators. (2019). ScientificTypes.jl: An API for dispatching on the "scientific" type of data instead of the machine type. In *GitHub repository*. GitHub. https://github.com/alan-turing-institute/ScientificTypes.jl

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Rev.*, *59*(1), 65–98. https://doi.org/10.1137/141000671

Bischl, B., Lang, M., Kotthoff, L., Schiffner, J., Richter, J., Studerus, E., Casalicchio, G., & Jones, Z. M. (2016). mlr: Machine Learning in R. *Journal of Machine Learning Research*, *17*(170), 1–5. http://jmlr.org/papers/v17/15-066.html

Blaom, A. (2020a). Flexible model composition in machine learning and its implementation in MLJ. *In Preparation*.

Blaom, A. (2020b). MLJ documentation. In *GitHub pages*. GitHub. https://alan-turing-institute.github.io/MLJ.jl/dev/

Blaom, A., & collaborators. (2020). Hyperparameter optimization algorithms for use in the MLJ machine learning framework. In *GitHub repository*. GitHub. https://github.com/alan-turing-institute/MLJTuning.jl

Blaom, A. et a. (2019). MLJ: A machine learning framework for Julia. In *GitHub repository*. GitHub. https://github.com/alan-turing-institute/MLJ.jl

Bouchet-Valat, M. et a. (2014). CategoricalArrays.jl: Arrays for working with categorical data. In *GitHub repository*. GitHub. https://github.com/JuliaData/CategoricalArrays.jl

Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., Layton, R., VanderPlas, J., Joly, A., Holt, B.,

& Varoquaux, G. (2013). API design for machine learning software: experiences from the scikit-learn project. *ArXiv*, *abs/1309.0238*.

Gressmann, F., Király, F. J., Mateen, B., & Oberhauser, H. (2018). Probabilistic supervised learning. *ArXiv*, *1801.00753*.

Holmes, G., Donkin, A., & Witten, I. H. (1994). WEKA: A machine learning workbench. *Proceedings of Anziis '94 - Australian New Zealnd Intelligent Information Systems Conference*, 357–361. https://doi.org/10.1109/ANZIIS.1994.396988

Innes, M. (2018). Flux: Elegant machine learning with Julia. *Journal of Open Source Software*, *3*(25), 602. https://doi.org/10.21105/joss.00602

Kuhn, M. (2008). Building predictive models in R using the caret package. *Journal of Statistical Software, Articles*, *28*(5), 1–26. https://doi.org/10.18637/jss.v028.i05

Lienart, T., Blaom, A., & collaborators. (2020). Data science tutorials in Julia. In *GitHub pages*. GitHub. https://alan-turing-institute.github.io/DataScienceTutorials.jl/

Lin, D., White, J. M., Byrne, S., Noack, A., Besançon, M., Bates, D., Pearson, J., Arslan, A., Squire, K., Anthoff, D., Zito, J., Papamarkou, T., Schauer, M., Drugowitsch, J., Sengupta, A., Smith, B. J., Moynihan, G., Ragusa, G., Stephen, G., … Widmann, D. (2020). *JuliaStats/distributions.jl: A Julia package for probability distributions and associated functions* (Version v0.23.2) [Computer software]. Zenodo. https://doi.org/10.5281/zenodo.3730565

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, É. (2011). Scikit-learn: Machine Learning in Python. *The Journal of Machine Learning Research*, *12*, 2825–2830. https://dl.acm.org/doi/10.5555/1953048.2078195

Quinn, J. (2020). Tables.jl: An interface for tables in Julia. In *GitHub repository*. GitHub. https://github.com/JuliaData/Tables.jl

Rackauckas, C. (2018). A comparison between differential equation solver suites in matlab, r, julia, python, c, mathematica, maple, and fortran. *The Winnower*. https://doi.org/10.15200/winn.153459.98975

Rackauckas, C., & Nie, Q. (2017). DifferentialEquations.jl – A performant and feature-rich ecosystem for solving differential equations in Julia. *Journal of Open Research Software*, *5*(1), 15. https://doi.org/10.5334/jors.151