MLJ: A JULIA PACKAGE FOR COMPOSABLE MACHINE LEARNING

Anthony D. Blaom
University of Auckland, Alan Turing Institute, New Zealand eScience Infrastructure
anthony.blaom@gmail.com

Franz Kiraly
University College London, Alan Turing Institute
fkiraly@turing.ac.uk

Thibaut Lienart
Alan Turing Institute
tlienart@turing.ac.uk

Yiannis Simillides
Imperial College London
ysimillides@imperial.ac.uk

Diego Arenas
University of St Andrews
da60@st-andrews.ac.uk

Sebastian J. Vollmer
Alan Turing Institute, University of Warwick
svollmer@turing.ac.uk

ABSTRACT

MLJ (Machine Learing in Julia) is an open source software package providing a common interface for interacting with machine learning models written in Julia and other languages. It provides tools and meta-algorithms for selecting, tuning, evaluating, composing and comparing those models, with a focus on flexible model composition. In this design overview we detail chief novelties of the framework, together with the clear benefits of Julia over the dominant multi-language alternatives.

Keywords machine learning toolbox · hyper-parameter optimization · model composition · Julia language · pipelines

1 Introduction

Statistical modelling, and the building of complex modelling pipelines, is a cornerstone of modern data science. Most experienced data scientists rely on high-level open source modelling toolboxes - such as sckit-learn [1]; [2] (Python); Weka [3] (Java); mlr [4] and caret [5] (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 able 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) [6] 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.

1.1 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

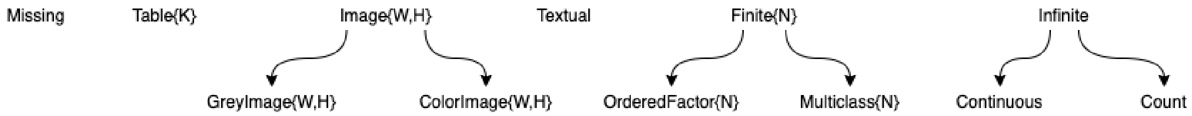

Figure 1: Part of the scientific type hierarchy.

dispatch, Julia solves the ubiquitous "two language problem" [7]. 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 [8], which, in a few short years of development by a small team of domain experts, became the best package in its class [9].

Another major advantange of a single-language solution is the ability to automatically differentiate (AD) functions from their code representations. The Flux.jl package [10], 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.

1.2 Novelties

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 behaviour 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.

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

A user can connect models directly to tabular data in a manifold of in-memory and out-of-memory formats, and usability is enhanced through the introduction of "*scientific types*" which allow the user to focus on the intended purpose of data ("continous", "ordered factor", etc) rather than particular machine type representations.

Finally, with the help of scientific types and the CategoricalArrays.jl package [11], 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 environments that simply recast categorical data using integers (e.g., scikit-learn): 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 anaysis and interpretability.

2 Scientific types

To help users focus less on data representation (e.g., `Float32`, `CategoricalValue{Char,UInt8}` or `DataFrame`) and more on intended *interpretation* (such as, "continuous", "ordered factor" and "table") MLJ articulates model data requirements, as well as data pre-processing tasks, using *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 (provided by an external package for re-use elsewhere) are shown in Figure 1.

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`. A `coerce` method recasts machine types to have the desired scientific type (interpretation), and a `schema` method summarizes the machine and scientfic 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.

3 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 (framed in terms of scientific types), the names and types of hyper-parameters, a brief document string, the url for the providing package, open source license and a load path to enable MLJ to locate the model interface code.

Such information allows users to match models to machine learning tasks, facilitating searches for an optimal model. For example, to find all supervised models making probabilistic predictions, compatible with input data X and target y, one defines a filter

```
In[1]:  using MLJ, CategoricalArrays
        scitype(1.2)

Out[1]: Continuous

In[2]:  table = (age=[20, 13, 48], attempts=[10, 5, 6], query=["why", "what", "whom"])
        schema(table)

Out[2]: -----------------------------------
        | _.names  | _.types | _.scitypes |
        -----------------------------------
        | age      | Int64   | Count      |
        | attempts | Int64   | Count      |
        | query    | String  | Textual    |
        -----------------------------------

        _.nrows = 3

In[3]:  fixed_table = coerce(column_table, :age=>Continuous, :query=>Multiclass)
        schema(fixed_table)

Out[3]: --------------------------------------------------------
        | _.names  | _.types                   | _.scitypes    |
        --------------------------------------------------------
        | age      | Float64                   | Continuous    |
        | attempts | Int64                     | Count         |
        | query    | CategoricalString{UInt32} | Multiclass{3} |
        --------------------------------------------------------

        _.nrows = 3
```

and lists the models with `models(task)`.

4 Flexible and compact work-flows for performance evaluation and tuning

Evaluating 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`, looks like this:

```
task(model) = matching(model, X, y) && model.prediction_type ==
:probabilistic models(task)
```

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 [12] 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.

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

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.

5 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 *probablistic* - 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 [13], for which an additional `pdf` method for evaluating the distribution's probability density or mass function will be implemented, and 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 labeled data (called `UnivariateFinite`) which additionally tracks all possible classes of the categorical variable it is modelling, and not just those observed in training data.

By predicting distributions, instead of raw probablities 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 probablisitic measures provided by MLJ are passed a distribution in their prediction slot.

For an overview on probabilistic supervised learning we refer to [14].

6 Model interfaces

MLJ provides a basic `fit/update/predict` interface to be implemented by new supervised models. For unsupervised models `predict` is replaced with `transform` and an optional `inverse_transform` method. These methods operate on *models* which are mutable structs storing hyper-parameters, and nothing else. This *model interface* is purely functional for maximum flexibility. Presently the general MLJ user is encouraged to interact through a *machine interface* sitting on top. See more on this below.

6.1 The model interface

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. In the low-level, functional-style, model interface *learned parameters* are not stored, only passed around. Learned parameters *are* stored in machines (which additionally *point* to the hyperparameters stored in a model); see below. The separation of hyper-parameters and learned parameters is essential to flexible model composition.

For supervised models the fit method has this signature:

```
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)
```

where `X` is training input and `y` the training target. The method outputs a triple, typically denoted `(fitresult, cache, report)`.

The `fitresult` stores the learned parameters, which must include everything needed by `predict` to make predictions, apart from `model` and new input data:

```
fit(model, verbosity, X, y)
```

The purpose of `cache` is to pass on "state" not included in the `fitresult` to an `update` method that the model implementer may optionally overload:

```
predict(model, fitresult, X)
```

This method is to be called instead of `fit` (and passed the `fitresult` and `cache` returned by the `fit` call) when retraining using identical data. (The data `X`, `y`, are included for implementer convenience.) It provides an opportunity for the model implementer to avoid unnecessary repetition of code execution. 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. If a "self-tuning" model has been fit (i.e., tuned) using 70 iterations of Tree Parzen optimization, then adding 20 more iterations should build on the existing surrogate objective function, not ignore the existing tuning history.
- **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 unecessary 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 cary out on-line learning.

6.2 The machine interface

The general MLJ user trains models through its machine interface. This makes some work-flows more convenient, but more significantly, introduces a syntax closely aligned with that for model composition (see below).

A *machine* is a mutable struct that binds a model to data at construction

```
update(model, verbosity, fitresult, cache, X, y)
```

When the user calls `fit!(mach, rows=...)` the `fitresult`, `cache` and `report` variables generated by lower-level calls to `fit` or `update`, are stored or updated in the machine struct, `mach`, with the training being optionally restricted to the specified rows of data. To retrain with new hyper-parameters, the user simply mutates `model` and repeats the `fit!` call.

Syntax for predicting using a machine is `predict(mach, Xnew)`.

7 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).

We now sketch MLJ's composition API, referring the reader to [15] for technical details, and to the MLJ documentation [16]; [17] 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*.

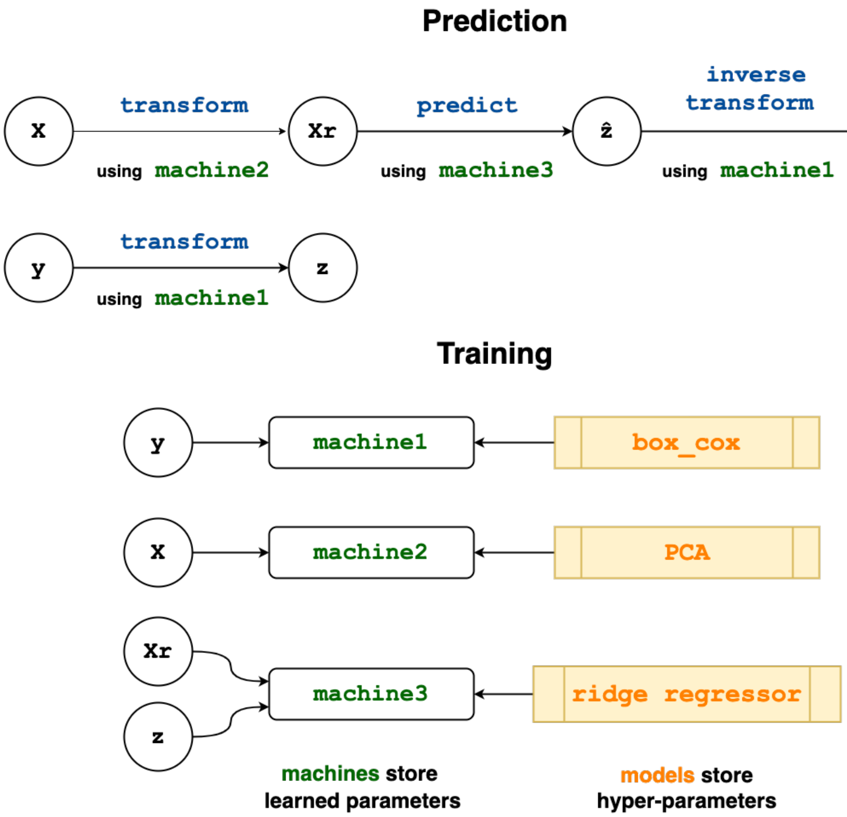

Figure 2: Specifying prediction and training flows in a simple learning network. The network shown combines a ridge regressor with a learned target transformation (Box Cox).

7.1 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.

The upper panel side of 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 (`machine2`) and the ridge model coefficients and intercept (`machine3`). The diagram additionally indicates where machines should look for training data, and where to accesses 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). Each data "variable" in the manual workflow becomes instead 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 in Figure 2 looks like this:

```
    mach = machine(model, X, y)
```

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.

7.2 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 (whose values are themselves models). 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.

8 Future directions

Here is a selection of future work planned or in progress:

- **Supporting more models**. Proofs of concept already exist for interfacing pure-Julia deep learning and probabilistic programming models.
- **Enhancing core functionality**. Add more tuning strategies, in particular, Bayesian methods and AD-powered gradient descent.
- **Broadening Scope**. Adding resampling strategies and tools for dealing with time series data, and for dealing with sparse data relevant in natural language processing.
- **Scalability**. Add DAG scheduling for learning network training

A more comprehensive road map is linked from the MLJ repository [6].

Acknowledgements

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, Practicies 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 intial 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

[1] 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. *The Journal of Machine Learning Research*, 12:2825–2830, 2011. URL https://dl.acm.org/citation.cfm?id=2078195.

[2] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jacob VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. Api design for machine learning software: experiences from the scikit-learn project. *ArXiv*, abs/1309.0238, 2013.

[3] G. Holmes, A. Donkin, and I.H. Witten. WEKA: a machine learning workbench. In *Proceedings of ANZIIS '94 - Australian New Zealnd Intelligent Information Systems Conference*, pages 357–361. IEEE, 1994. ISBN 0-7803-2404-8. doi: 10.1109/ANZIIS.1994.396988. URL http://ieeexplore.ieee.org/document/396988/.

[4] Bernd Bischl, Michel Lang, Lars Kotthoff, Julia Schiffner, Jakob Richter, Erich Studerus, Giuseppe Casalicchio, and Zachary M. Jones. mlr: Machine learning in R. *Journal of Machine Learning Research*, 17(170):1–5, 2016. URL http://jmlr.org/papers/v17/15-066.html.

[5] Max Kuhn. Building predictive models in R using the caret package. *Journal of Statistical Software, Articles*, 28 (5):1–26, 2008. ISSN 1548-7660. doi: 10.18637/jss.v028.i05. URL https://www.jstatsoft.org/v028/i05.

[6] Anthony et al. Blaom. MLJ: A machine learning framework for Julia, 2019. URL https://github.com/alan-turing-institute/MLJ.jl.

[7] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: a fresh approach to numerical computing. *SIAM Rev.*, 59(1):65–98, 2017. ISSN 0036-1445. doi: 10.1137/141000671. URL https://doi.org/10.1137/141000671.

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

[9] C. Rackauckas. A comparison between differential equation solver suites in MATLAB, R, Julia, Python, C, Mathematica, Maple, and Fortran, 2017. URL https://www.stochasticlifestyle.com/comparison-differential-equation-solver-suites-matlab-r-julia-python-c-fortran/.

[10] Mike Innes. Flux: Elegant machine learning with julia. *Journal of Open Source Software*, 3(25):602, 2018. doi: 10.21105/joss.00602. URL https://doi.org/10.21105/joss.00602.

[11] Milan et al. Bouchet-Valat. CategoricalArrays.jl: Arrays for working with categorical data, 2014. URL https://github.com/JuliaData/CategoricalArrays.jl.

[12] Anthony Blaom and collaborators. Hyperparameter optimization algorithms for use in the MLJ machine learning framework, 2020. URL https://alan-turing-institute.github.io/MLJTuning/.

[13] Dahua Lin, John Myles White, Simon Byrne, Andreas Noack, Mathieu Besançon, Douglas Bates, John Pearson, Alex Arslan, Kevin Squire, David Anthoff, John Zito, Theodore Papamarkou, Moritz Schauer, Jan Drugowitsch, Avik Sengupta, Brian J Smith, Glenn Moynihan, Giuseppe Ragusa, Gord Stephen, Christoph Dann, Mike J Innes, Michael, Martin O'Leary, Tamas K. Papp, Jiahao Chen, Iain Dunning, Gustavo Lacerda, Richard Reeve, Kai Xu, and David Widmann. Distributions.jl: A Julia package for probability distributions and associated functions, March 2020. URL https://doi.org/10.5281/zenodo.3730565. doi: 10.5281/zenodo.3730565.

[14] Frithjof Gressmann, Franz J. Király, Bilal Mateen, and Harald Oberhauser. Probabilistic supervised learning. *ArXiv*, 1801.00753, 2018.

[15] Anthony Blaom. Flexible model composition in machine learning and its implementation in MLJ. In preparation, 2020.

[16] Anthony Blaom. MLJ documentation, 2020. URL https://alan-turing-institute.github.io/MLJ.jl/dev/.

[17] Thibaut Lienart, Anthony Blaom, and collaborators. Data Science Tutorials in Julia, 2020. URL https://alan-turing-institute.github.io/DataScienceTutorials.jl/.