Machine Learning in Julia

Edoardo Barp
Centre for Complexity Science
University of Warwick
Coventry, United Kingdom
e.g.barp@warwick.ac.uk

Harvey Devereux
Centre for Complexity Science
University of Warwick
Coventry, United Kingdom
h.devereux@warwick.ac.uk

Mohammad Noorbakhsh
Centre for Complexity Science
University of Warwick
Coventry, United Kingdom
m.noorbakhsh@warwick.ac.uk

Annika Stechemesser
Centre for Complexity Science
University of Warwick
Coventry, United Kingdom
a.stechemesser@warwick.ac.uk

Abstract

julia is an open-source programming language for machine learning, data science and scientific computing combining high speed and an easy-to-use syntax. To improve its performance and user experience while solving machine learning tasks, Julia Computing paired up with the Mathematics of Real World Systems Doctoral Training Centre. The aim of the project was to analyse the current state of the machine learning ecosystem in Julia, to facilitate the usage of packages by providing instructions and explanations, and to fill part of the gaps in the ecosystem that were identified in the analysis. The analysis was done in the form of a strategic package review testing many of the published ML packages in Julia in the very fragmented ecosystem. It showed a lack of an overall framework, a substandard performance of decision trees and random forests and missing implementations of clustering metrics and outlier detection. We were able to fill in these gaps by the building the packages `MLJ.jl`, `MondrianForest.jl` and making contributions to the `MLmetrics.jl` package.

1 Introduction

Since its launch in 2012, the programming language Julia [9] has gained growing attention with more than 1,800,000 downloads as of January 2018 [28]. As the first high-level, high-performance, open-source, multiple-dispatch based programming language it offers an array of interesting features to the scientific computing community.

Julia uses Just-In-Time (JIT) compilation, meaning that part of the code is compiled whilst being executed, if this is beneficial, positioning Julia in the middle between a fully interpreted and a fully compiled language. Its performance is very convincing, sometimes comparable to that of `C` programs. This combination of speed and high-level syntax makes Julia suitable for solving the two-language problem, which is that most programming languages are either convenient to write in or have a high performance, but rarely both. Therefore, the most common solution is to write computationally intensive code in a high performance language (`C`, `C++`), and then "wrap" around it using languages which have an easier syntax (`Python`, `R`, `Matlab`), needing two different languages to solve a problem. This is feasible for projects having a lot of resources but not for smaller projects requiring similar performances but not being able to afford building these wrapping mechanisms. Moreover the two language approach is prone to human error and highly inflexible.

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.

Whilst the Julia ecosystem is growing relentlessly, with a 60% increase in the number of Github Stars for Julia and its community packages from 2017 to 2018 [28], there are some challenges that come with an open-source system. Often, a wide variety of packages of differing functionality exists for solving a problem and many of these will be missing some core functionalities.

This project was done in cooperation with Julia Computing to review the status and fill gaps in the machine learning community in Julia, an area in which it is possible to have a strong impact. Many companies highly appreciate the advantages Julia offers and have started using Julia for solving real world problems often involving machine learning tasks. For example, engineers in Queensland use machine learning in Julia to identify points on the electrical grid that have failed or are at risk of failure [22] and deep learning in Julia is used to diagnose diabetic retinopathy [25]. The aim of our project was to have a positive impact on the machine learning community by following a twofold approach.

Initially, a package review of core machine learning packages was done during which we raised issues where bugs were identified and flagged packages which were substantially broken. We grouped the identified issues into two categories. Many packages had good functionality, but were missing instructions and explanations that would allow members of the community to use them properly. We solved these *educational issues* by writing a large amount of Jupyter notebooks for several important packages including documentation, examples and details about the mathematical background of the problems and algorithms. In *section 2* we present the results of the package review, explain our work flow and discuss the main issues found in the ecosystem. We then proceed in *sections 3 and 4* to explain the most notable educational improvements.

Other packages have proper instructions but are lacking important *functionalities*. We contributed to the `MLMetrics.jl` package [5] by implementing clustering metrics. Details about the contributions can be found in *Section 5*.

We aim to set a new standard for decision trees in Julia with the `MondrianForest.jl` package after identifying gaps in the implementations of decision trees and random forests. Random forests are to this day one of the most popular machine learning methods. Citations of core articles in the field like "Random Forests" by Leo Breiman [13] have been constantly increasing reaching a total of over 35000 citations so far (see figure 12, Appendix D). An improvement in this area could have a meaningful impact on the Julia community. Details about the implementation of Mondrian forests can be found in *section 6*.

One of the most challenging problems with the Julia machine learning ecosystem is the lack of a common syntax as each package has an individual interface which is not necessarily compatible with other packages. The package `MLJ.jl` is designed to provide this much needed common syntax for all core machine learning packages. Additionally, it attempts to fill the lack of composite trainers. Packages that frame multiple machine learning methods and make it easy for the user to build and access models have proven to be vital in other programming languages, such as `scikit-learn` [35] in Python, which was cited 10275 times (figure 12). Insights in the implementation of `MLJ` are given in *section 7*.

In all our implementations we made sure to follow a few "philosophical principles" that make code especially efficient in Julia. Julia is a multiple dispatch based language. A dispatch is the choice of which method to apply if multiple functions are called. Julia makes this choice based on the number of arguments given and the types of these arguments. The usage of multiple dispatch programming to get very clean, understandable code is shown in (*section 6*). Furthermore Julia's compiler specialises code for argument type, meaning core computations of an algorithm should be split into sub-functions. Examples of this are given in listings 3 and 4 (*section 6*). Another specialty in Julia is that it is sensible to de-vectorise expressions. Loops in Julia are, unlike in Python or Matlab, as fast as in C. Therefore it is more efficient to apply mathematical operations to every element of a vector, instead of to the vector directly. An example for de-vectorisation is given in *section 5*.

2 Package Review

The aim of the package review was to assess the current state of the Julia machine learning library, in order to identify issues and missing functionalities. For this purpose, a list of packages relating to

important machine learning models was compiled. Each package was first tested against a series of summary questions. Finally, important packages were further analysed and documented.

2.1 Reviewing Process Methodology

When checking a package, a consistent work flow was followed. First, we made a short summary of the package to clarify which problem the package aims to solve and what expectations it should fulfill. Then a list of core yes/no questions was checked for a quick assessment of the package. This helped to highlight important issues immediately such that packages that were fundamentally broken didn't need to be analysed further. These short summaries were then collected and sorted by category, to make it easy to compare packages with just a look. A snippet of that file, showing the questions, is shown in figure 1.

	SparseRegression.jl	MultivariateStats.jl	OnlineStats.jl
Package works	yes	yes	yes
Deprecations warnings	No	No	No
Compatible with JuliaDB	If transformed into matrix	If transformed into matrix	If transformed into matrix
Documentation	lacking	very good	very good
Simplicity	good	good	High

Figure 1: Example of summary for the linear regression packages.

In the next step we wrote modest documentation if the package lacked it, and offered examples. We then clustered the packages and composed more theoretical documents, explaining the different models and how to use them. We compared the libraries and benchmarked packages with similar functionality against each other and against the related `Python` and `R` packages. These "theory notebooks" make it easy for users to find the best package for the problem they want to solve saving the frustration of having to learn a packages syntax first to then discover that it doesn't solve the task.

2.2 Results and Discussion / Conclusions

The package review clearly showed how fragmented the Julia machine learning ecosystem is. The sheer amount of available packages, many of which being incomplete, lacking or simply deprecated can make for a frustrating user experience. The results of our package review provide guidance on which packages to use. All results of the review can be found in our `Github` repository (https://github.com/dominusmi/warwick-rsg/tree/master/Scouting).

General ease-of-use is a very important factor to attract and maintain the attention of new users, but available packages of very good functionality often weren't simple to use and had little documentation. In these cases we attempted to write sizable and complete instructions in our analysis to encourage users to still use otherwise perfect libraries. An example for this is the package `LearningStrategies.jl` [4] which will be further discussed in *section 3*.

A special focus was put on neural networks as this type of machine learning model becomes increasingly more popular and important. The two main packages, `Knet` [21] and `MLFlux` [3] are compared in detail in *section 4*. The examples provided allow a simpler acclimatisation for users coming from different languages.

We identified multiple gaps in the ecosystem, either because all available packages for a particular problem were broken or because the Julia implementation couldn't compete with other languages in the benchmarks. The latter was the case for the decision tree implementations in Julia which is further discussed in *section 6*. Similarly, gaps were found in the package `MLMetrics.jl` [5], which consists of multiple metrics for various machine learning problems. The lack of clustering metrics was solved by a contribution to the package which is discussed in *section 5*.

Finally, packages often use different conventions, for instance some packages would use *learn* and others *train*, some *predict* and others *evaluate*, and so on. The package `LearningStrategies.jl` [4] introduces a generic framework for any learning algorithm and will certainly be used more

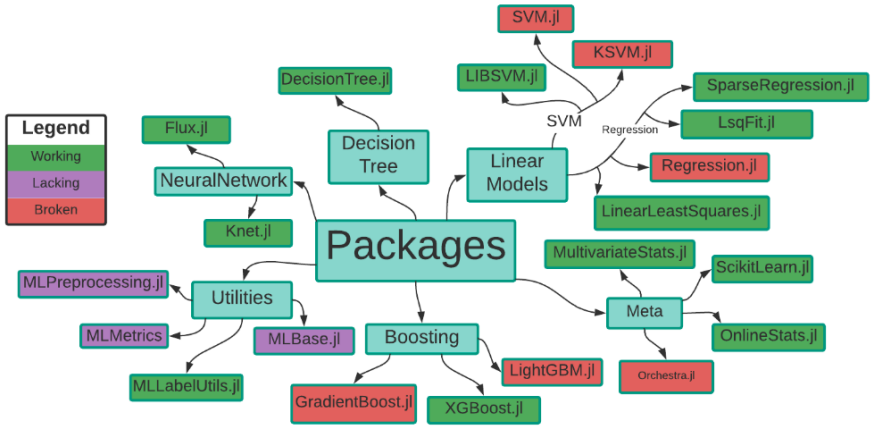

Figure 2: Visual simplified summary of the package review, clustered by package type.

frequently when having proper documentation (see *section 3*). The implementation of the interface package `MLJ.jl` (*section 7*) finally allows to train multiple models without having to worry about the syntax of each package.

3 Structural Inconsistency: Learning Strategies

3.1 Summary and Aims

Often users want to implement new optimisation algorithms or learning strategies for models for which packages already exist. But especially in Juila these packages often use different conventions and structure and lack proper documentation such that users will end up building their own package instead of contributing to existing ones. This causes a lot of unnecessary work, since building a new package also includes building side utilities, such as process logging, data loading, various convergence checks and so on.

`LearningStrategies.jl` [4] is a Julia package which provides a framework allowing users to implement new algorithms and models, in an attempt to solve both the problems mentioned. At the forefront, it formalises the structure for learning packages, forcing the use of a common syntax and structure. It further implements a modular structure for strategies, which can be used to implement complete learning strategies or to simply provide some side features that can easily be re-used.

This package is currently not used throughout the community. Partly responsible is certainly the lack of documentation with only a basic introduction and few examples given. This is an even bigger obstacle because of the highly abstract nature of the package, meaning that it needs to be understood fully to be used effectively. Therefore, to make the package more user-friendly we provided a structured and in-depth documentation, together with multiple examples for several different types of algorithms. Given the lacking documentation, all the precise mechanisms of the package had to be investigated and inferred directly from the source code. These are briefly discussed in the next section.

3.2 Explanation of the Structure

The main pseudo-code around which the whole framework is based (see Listing 1) is divided into multiple functions, each with its specific task, which are described below:

1. **Setup:** The function used to initialise the model and the strategy. The training data can be passed if necessary.
2. **For loop:** Main loop over the training data. The behaviour of the loop will depend on how the data is passed. If the programmer wants the same data to be used for all iterations of

the learning process, they could for instance make sure the data is `repeated`[1]. Similarly, K-fold and other sampling methods can easily be implemented.

3. **Update:** The main learning function. The user can implement several strategies for the same model. For instance, gradient finding could be done using the *Newton-Raphson* method, or the *Euler* method, and so on.
4. **Hook:** Allows to *hook in* the learning process. Mostly used for logging, verbose, and other activities which do not interfere with the learning.
5. **Finished:** Allows to do convergence checks. For instance, check whether the loss is smaller than some ϵ.
6. **Cleanup:** Allows to implement clean up, for instance for memory allocations, if required.

Listing 1: `LearningStrategies` framework pseudo-code

```
function learn!(model, strategy, data)
    setup!(strategy, model [,data] )
    for (i, item) in enumerate(data)
        update!(model, strategy, i, item)
        hook(strategy, model, [,data, i] )
        finished(strategy, model [,data, i])
    end
    cleanup!(strategy, model)
    return model
end
```

3.3 Explanation of Strategies

It is important to note that there are two types of strategies, learning strategies and meta-strategies. The difference between the two is conceptual. A learning strategy is expected to overload the `update` method, that is to implement a learning algorithm, whereas meta-strategies can be thought of as helper strategies. These could simply be logging strategies or doing convergence checks and therefore will usually only overload `hook` or `finished`.

Finally, it is important to note that the purpose of the framework is for users to have to do the least amount of work to implement their model, since they can re-use all the meta-strategies available to do trivial tasks such as logging progress, tracking, and so on. For instance, a popular combination of strategies is

```
learn!(model, strategy(
    Optimiser(),
    ConvergedTo(loss, ϵ, goal, max_iter),
    Log(model, x→[x.x,x.y])
))
```

where `Optimiser` is our main learning strategy, and the others are meta-strategies. `ConvergedTo` checks that `|loss(model)-goal|` $< \epsilon$ and additionally forces termination after `max_iter` iterations; `Log`, logs the return of the abstract function $x \rightarrow [x.x, x.y]$[2].

3.4 Discussion

This package is very well built, allowing for several types of models, from optimisers to brute-force learning. It introduces a consistent framework, making it very simple to understand new strategies and models and to further modify these according to specific needs. Therefore, `LearningStrategies.jl` solves the recurring consistency problem that was observed several times throughout the reviewing. As such, this package could greatly benefit the community if it became more popular.

[1] Act of creating an iterator over the whole data, useful when an algorithms needs the whole dataset at each iteration. Note: this does not copy the memory, it simply points to the same section at each iteration.

[2] Abstract functions are functions, usually very simple, for which assigning a name is not necessary. In this case, the abstract function extracts the internal variables x and y

The main obstacle to using the package was the lack of explanation and examples. This is now solved as our documentation was proposed to the authors, who greatly appreciated it. Our contribution will be linked at the very top of the package `Github`[3] page.

4 Deep Learning in Julia

Deep-learning has become a prominent topic in the field of machine learning, therefore we put a special focus on it while reviewing the Julia machine learning libraries.

Deep learning packages can be divided between `static` and `dynamic` approaches [27]. `Dynamic` packages are high-level APIs which interface with other back-end and low-level libraries, such as `TensorFlow` [6], enabling users to build complex deep learning models with just a few lines of code, such as `Keras` [17] in Python. They are designed to provide a high-level of abstraction and make the whole process of building a model quick, easy and user-friendly. On the other hand, `static` packages are low-level and provide more flexibility for the users and allow them to customise and optimise deep learning models with more freedom. These two approaches are presented in Julia, and the trade-off between simplicity and efficiency shown.

4.1 Comparison between Flux and KNet

There are two major deep learning packages in Julia, `Flux` [26] and `Knet` [21].

`Knet` is a low-level, deep-learning framework written entirely in Julia and following a static approach. It relies on a package called `AutoGrad.jl` [20], an automatic differentiation package for Julia, closely related to the popular homonymous `Python` package.

Being low-level, everything about the model needs to be specified, except for the gradient calculations used in the weight updates. This allows complete freedom in the neural network's architecture, choice of activation and training rule. For instance, the weight matrix can be initialised in any way, allowing the creation of non-fully connected layers. The trade-off is that every weight matrix needs to be manually initialised, although some helper methods exist.

Additionally, the training function needs to be explicitely defined. The trade-off is again present in this step, since in the most simple case, the user is still required to create a loop over all the weights. On the other hand, this also allows the implementation of entirely different types of networks, such as recurrent neural networks, where part of the input is reused on the same layer, and therefore requires uncommon training mechanisms.

`Flux` softens the distinction between `static` and `dynamic` approaches. It offers a similar low-level approach as that of `Knet`, but additionally has a high-level interface, through which neural networks can be used similarly as in `Keras` in Python. Therefore, `Flux` allows the users to easily switch back and forth between two approaches. It is important to note that `Knet` is complete and fully working, whereas `Flux` is still being developed, and some important features, such as convolutional neural networks, are still missing.

[3] https://github.com/JuliaML/LearningStrategies.jl

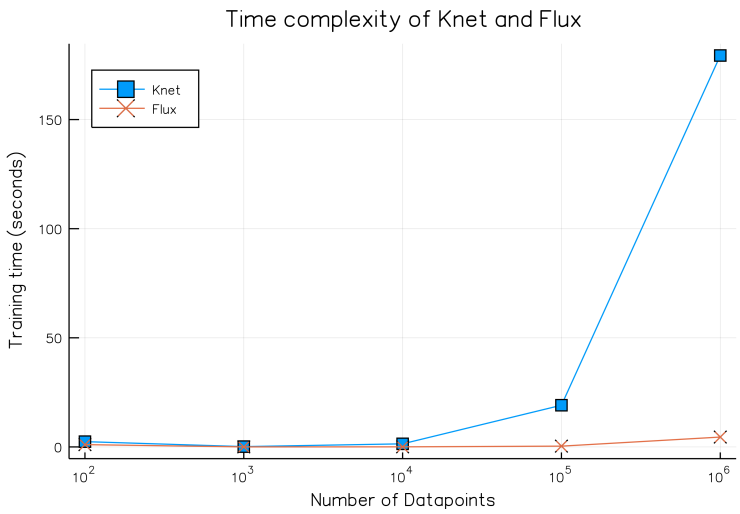

Figure 3: Comparison between the complexity of Knet and the complexity of Flux

5 Clustering in Julia

5.1 Summary

During the package review, we noticed that while several packages exist for supervised learning utilities this is not the case for unsupervised learning, counting a library of few, unorganised packages. The main purpose of this section of the project was to contribute to filling this gap by implementing some of the core unsupervised learning functionalities, focusing on clustering metrics and outlier detection algorithms.

`MLMetrics.jl` [5] provides several commonly used metrics for evaluating errors. The package attempts to mimic Pythons `Sklearn.metrics` [35] as closely as possible. `MLMetrics.jl` is a perfect candidate among the metrics packages to extend since it is popular so the added functionality will help the whole community. In the subsequent sections, the theoretical background is introduced and the implementations are discussed.

5.2 Theoretical Background

5.2.1 Measures for Comparing Clusterings

This section introduces very briefly the theoretical background of `Clustering Metrics` based on the work of Vinh et al. (2010) [32].

Let S be a set of N data items, and let U and V be the partitionings resulting from two distinct clustering methods. Both these partitions are made of mutually exclusive subsets $\{U_1, U_2, ..., U_R\}$ and $\{V_1, V_2, ..., V_C\}$, where R and C are the number of clusters found by the respective methods. Note that $\bigcup_{i=1}^{R} U_i = S$ and $\bigcup_{i=1}^{C} V_i = S$.

The information on how these two clusterings overlap can be summarised in a $R \times C$ contingency table illustrated in Table 1, where n_{ij} is the number of common items between U_i and V_j, and where a_i and b_i are the marginal distributions.

U\V	V_1	V_2	$\cdots$	V_C	Sums
U_1	n_{11}	n_{12}	$\cdots$	n_{1C}	a_1
U_2	n_{21}	n_{22}	$\cdots$	n_{2C}	a_2
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
U_R	n_{R1}	n_{R2}	$\cdots$	n_{RC}	a_R
Sums	b_1	b_2	$\cdots$	b_C	$\sum_{ij} n_{ij} = N$

Table 1: Example of a contingency table [32]

Using the marginal and joint distribution of data items in the contingency table, we can define entropy, joint entropy, conditional entropy and mutual information of the two clusterings U and V as (Cover and Thomas, 1991) [18]:

$$H(U) = -\sum_{i=1}^{R} \frac{a_i}{N} log \frac{a_i}{N}$$

$$H(U,V) = -\sum_{i=1}^{R}\sum_{j=1}^{C} \frac{n_{ij}}{N} \log \frac{n_{ij}}{N}$$

$$H(U|V) = -\sum_{i=1}^{R}\sum_{j=1}^{C} \frac{n_{ij}}{N} \log \frac{\frac{n_{ij}}{N}}{\frac{b_j}{N}}$$

$$I(U,V) = \sum_{i=1}^{R}\sum_{j=1}^{C} \frac{n_{ij}}{N} \log \frac{\frac{n_{ij}}{N}}{\frac{a_i b_j}{N^2}}$$

There are several desirable properties for a clustering metric, as discussed by Vinh et al.[32]. One such property is that the metric can be normalised. This property is not satisfied by the mutual information, since its domain is $[0, \min\{H(U), H(V)\}]$. Therefore, a lot of variants have been proposed which do satisfy a property, as shown in Table 2.

Name	Expression	Range	Related Source
NMI_{joint}	$\frac{I(U,V)}{H(U,V)}$	[0,1]	Yao(2003) [43]
NMI_{max}	$\frac{I(U,V)}{max\{H(U),H(V)\}}$	[0,1]	Kvalseth (1987) [29]
NMI_{sum}	$\frac{2I(U,V)}{H(U)+H(V)}$	[0,1]	Kvalseth (1987) [29]
NMI_{sqrt}	$\frac{I(U,V)}{\sqrt{H(U)H(V)}}$	[0,1]	Strehl and Ghosh (2002) [40]
NMI_{min}	$\frac{I(U,V)}{min\{H(U),H(V)\}}$	[0,1]	Liu et al.(2008) [44]

Table 2: List of variants of the mutual information metrics which satisfy the normalisation property [32]

Similarly, *constant baseline* is another desirable property which is satisfied if the expected value of a metric for pairs of independent clustering is a constant, preferably 0. The mutual information does not have this property, therefore we introduce the adjusted mutual information, a variant proposed by Vinh et al (2010) [32]. It is defined as

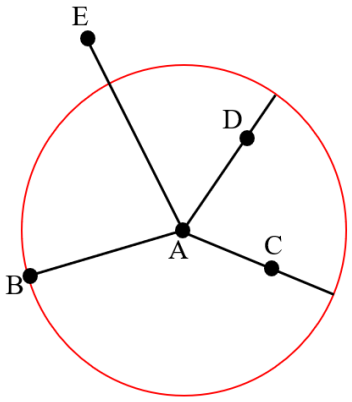

Figure 4: Objects A,B,C and D have the same reachability distance (k=3). Since E is not a k-nearest neighbor, its reachability distance from A is equal to the actual distance, $d(E, A)$.

$$AMI_{max}(U,V) = \frac{NMI_{max}(U,V) - E\{NMI_{max}(U,V)\}}{1 - E\{NMI_{max}(U,V)\}} = \frac{I(U,V) - E\{I(U,V)\}}{max\{H(U),H(V)\} - E\{I(U,V)\}}$$

where $E(I_{U,V})$ (Vinh et al., 2009) [42] is defined as

$$E(I_{U,V}) = \sum_{i=1}^{R}\sum_{j=1}^{C}\sum_{n_{ij}=\max(a_i+b_j-N,0)}^{\min(a_i,b_j)} \frac{n_{ij}}{N}\log(\frac{N.n_{ij}}{a_i b_j})\frac{a_i!b_j!(N-a_i)!(N-b_j)!}{N!n_{ij}!(a_i-n_{ij})!(b_j-n_{ij})!(N-a_i-b_j n_{ij})!}$$

5.2.2 Local Outliers Factor

The Local Outlier Factor (LOF) is a measure to quantify the belief that a point is an outlier. It was proposed by Breunig et al. [15]. The intuition behind LOF is that an outlier has a very different surrounding density than its neighbours.

The following definitions are required:

Definition 5.1 *Let K be a natural number, The reachability distance of object p with respect to object o is defined as reach-dist$_k(p,o) = \max\{$k-distance$(o), d(p,o)\}$ where d(p,o) is a predefined generic distance metric, and the k-distance of p, is the distance such that for at least k objects o', t d(p,o') $\leq$ d(p,o).*

Definition 5.2 *The local reachability density of p is defined as*

$$ird(p) = \Big(\frac{\sum_{o\in N_k(p)} \textit{reach-dist}(p,o)}{|N_k(p)|}\Big)^{-1}$$

Definition 5.3 *The Local Outlier Factor of p is defined as*

$$LOF_k(p) = \frac{\sum_{o\in N_k(p)} \frac{ird(o)}{ird(p)}}{|N_k(p)|}$$

5.3 Implementation

5.3.1 Clustering Metrics

The contingency tables required for all of the aforementioned metrics can be built using the package `FreqTables.jl` [12]. Based on that, all metrics mentioned in section 5.2 were implemented. Variants are made available through the use of a `mode` parameter which can be passed.

It is evident that the various metrics make a large use of sums. To optimise the implementation for Julia the sums are not applied to vectors but to each element of the vectors individually. Loops are especially fast in Julia, therefore this devectorisation increases the efficiency. An example for this coding style is given in code snippet listing 2.

Listing 2: Mutual Information Score code

```
function MI(target::AbstractVector, output::AbstractVector)
    result = 0
    table = ContingencyTable(target,output)
    # devectorisation
    for i =1:table.R
        for j = 1:table.C
            intersection_number = table.n[i,j]
            true_number = table.a[j]
            pred_number = table.b[j]
            N = table.N
            temp = (intersection_number / N) * log(N* intersection_number / (
                true_number*pred_number))
            if !isnan(temp)
                result += temp
            end
        end
    end
        result
end
```

5.3.2 Local Outliers Factor

In order to implement anomaly detection with Local Outliers Factor (LOF), the following functions were implemented:

- `kneighbors`: returns the k Nearest Neighbors of X, using the package `NearestNeighbors.jl` [16].
- `fit_predict`: fits the model to the training set X, and returns a value of 1 for normal points, and -1 for outliers, according to the *LOF* score and the contamination parameter.
- `fit`: fits the model using X as training data and returns the *LOF* score and the threshold value according to contamination parameter.
- `local_reachability_density`: returns the local reachability density of a sample which is the inverse of the average reachability distance of its k-nearest neighbors.

An example of anomaly detection applied to a two-dimensional an artificial dataset is shown in figure 5.3.2.

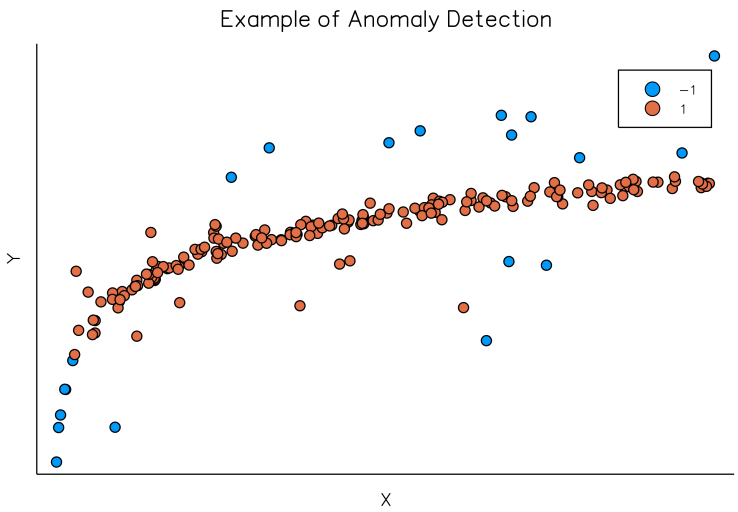

Figure 5: Example output of the anomaly detection. Outliers are labeled -1.

6 Mondrian Forests

6.1 Motivation

Decision tree and random forest algorithms are widely used in machine learning tasks and core methods in every machine learning framework. Each node in the tree represents a test applied to a feature of the data which splits the data into partitions. This partitioning leads to a structure as shown in figure 6 [24]. In practice decision tree algorithms need to specify how the test at each node is chosen and when to stop splitting data with more tests. The method by which the test (i.e the feature tested) at each node is chosen depends on the algorithm used. Popular examples are the ID3 algorithm (Iterative Dichotomiser 3) [36] or the CART (Classification and Regression Trees) algorithm [14].

The random forest algorithm is an ensemble model consisting of a set of decision tree models, each with a set of random parameters all drawn from the same distribution, and each having a unit vote as to the final classification/regression output. The parameters can be a random selection of features or probabilities for a random split selection at each node (so the best is not always chosen). The benefit of introducing this randomness is that it produces largely uncorrelated trees. It can be shown that an upper bound on the generalisation error of a random forest is proportional to the average correlation of the trees.

Our package review showed that the implementations of decision trees and random forests in Julia have room for improvement. The most complete decision tree package `DecisionTree.jl` [38] implements the CART algorithm [14] for decision tree classification and regression models along with a random forest implementation based on these. Even though the functionality is reasonable, problems arise when benchmarking the training time against Python's `Scikit-Learn` [35] implementation, as shown in Figure 7. As a language being particularly used for its speed, this weak performance of a standard method is significant.

The only implementation of an online random forest we found is part of the package `OnlineStats.jl` [19], where "FastForests" [39] are implemented. During our package review we identified an error in the implementation of this algorithm and came up with a fix for it. We raised this as an issue on the `Github` page of the package and our suggestions were incorporated by the author. However, the method still doesn't seem to be working in a reliable manner.

To decrease these gaps and add a cutting-edge method in Julia, we implemented Mondrian trees and forests. With the related papers being published in 2014 [30] and 2016 [31], Mondrian forests are very much on the level of the state-of-the-art. A decision tree algorithm, a random forest algorithm and an algorithm for efficient online training are included. In the following sections we give an overview over the underlying theory and functionality, explain details about the implementation that

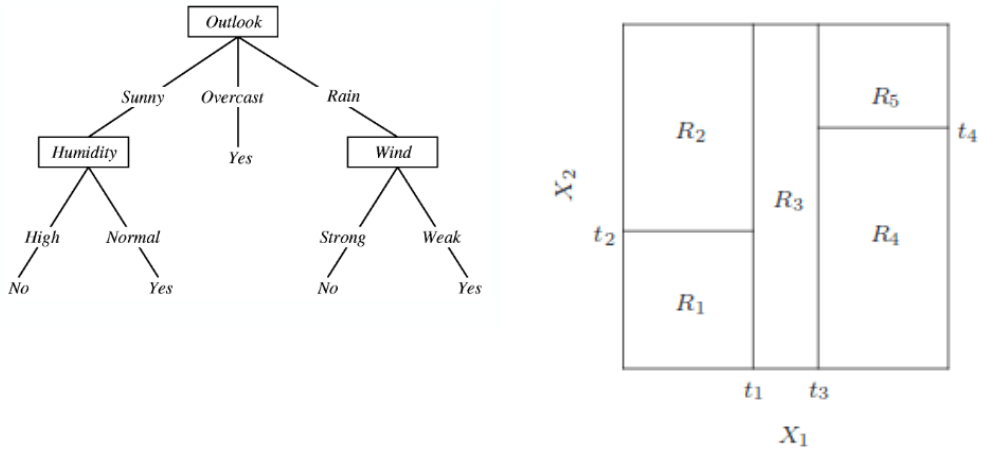

Figure 6: (Left) Example of a decision tree constructed on a toy dataset and (Right) A diagram of splits in an arbitrary decision tree constructed on two dimensional data.

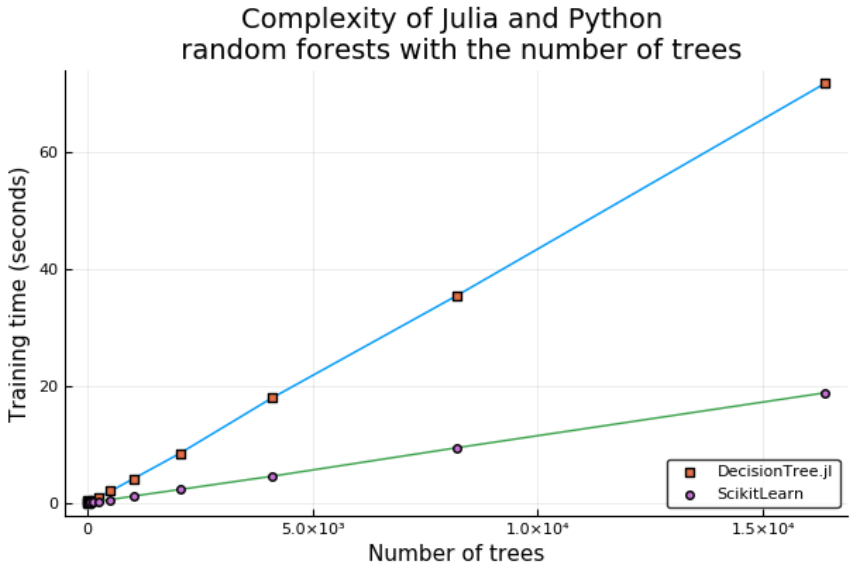

Figure 7: Training time scaling of random forests in Julia (`DecisionTree.jl`) and `Python` (`ScikitLearn`) with the number of trees trained, over a fixed number of data points.

are specific to Julia or that we altered from the versions proposed in the paper and show benchmarks against the equivalent implementation in Python.

6.2 Definitions of Mondrian Processes, Trees and Forests

Using Mondrian Processes as underlying processes for random forests was proposed in 2014 by B. Lakshminarayanan, Daniel M. Roy and Yee Whye Teh [30] who defined the so called Mondrian Forests for classification tasks. In 2016 they provided an extension on how to perform regression tasks using Mondrian Forests [31]. Here we first want to give a short introduction to the theory involved.

6.2.1 Mondrian Processes

Before we can define Mondrian processes we need to introduce some underlying concepts following the definitions given in [8].

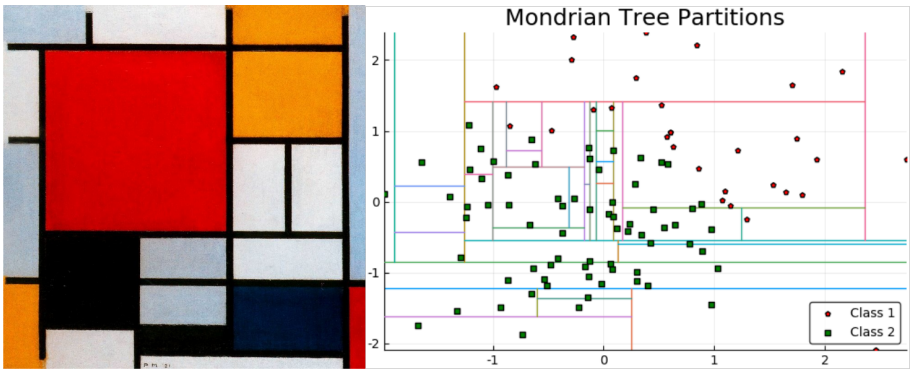

Figure 8: Left: *Composition with Large Red Plane, Yellow, Black, Gray and Blue*, Piet Mondrian, 1921 [1]; Right: Visualisation of Mondrian Tree Partitions. The smaller boxes are a refinement of the larger ones.

Definition 6.1 *A temporal stochastic process taking values in a space S is a collection* $(M_t)_{t\geq 0}$ *of S-valued random variables, indexed by a parameter* $t \in [0, \infty)$ *that we think of as time.*

Definition 6.2 *An (axis-aligned) box* Θ *in* $\mathbb{R}^D$ *is a set of the form* $\Theta = \Theta_1 \times ... \times \Theta_D \subseteq \mathbb{R}^D$*, where each* Θ_d *is a bounded interval* $[a_d, b_d]$*.*

Definition 6.3 *Given a box* Θ *in* $\mathbb{R}^D$*, a guillotine partition of* Θ *is a hierarchical partition of* Θ *obtained by recursively splitting boxes of the partition by some hyperplane orthogonal to one of the D coordinate axes.*

Based on these definitions we define a **Mondrian Process**.

Definition 6.4 *The Mondrian process on* $\mathbb{R}^D$ *with lifetime* $\lambda \geq 0$ *written* $MP(\lambda, \mathbb{R}^D)$ *is the temporal stochastic process taking values in (infinite) guillotine partitions of* $\mathbb{R}^D$ *with the property that its restriction to any bounded axis-aligned box* Θ *is again a Mondrian process.*

Intuitively, we can understand Mondrian processes as families $\{M_t : t \in [0, \infty)\}$ of random, hierarchical binary partitions of $\mathbb{R}^D$ such that M_t is a refinement of M_s whenever $t > s$. To visualise a Mondrian process, we can think of the art painted by the dutch artist Piet Mondrian in 1921, whose paintings inspired the name of the stochastic process (figure 8).

6.2.2 Mondrian Trees and Forests

Before we define Mondrian trees, we need to introduce the mathematical definition of decision trees that will be used for our purposes [30].

Definition 6.5 *A rooted, strictly-binary tree is a finite tree* T *on* $\mathbb{R}^D$ *such that every node in* T *is either a leaf or internal node with exactly two children. Every node has exactly one parent node except the root. Each node* $j \in T$ *is associated with a block* $B_j \subseteq \mathbb{R}^D$*.*

Definition 6.6 *Let* T *be a rooted, strictly-binary tree on* $\mathbb{R}^{\mathbb{D}}$ *and* j *a node in* T*. A split is a tuple* (δ_j, ζ_j)*, where* $\delta_j \in \{1, 2, ..., D\}$ *denotes the dimension of the split and* ζ_j *denotes the location of the split along dimension* δ_j*. We define* $B_{left(j)} := \{x \in B_j : x_{\delta_j} \leq \zeta_j\}$ *and* $B_{right(j)} := \{x \in B_j : x_{\delta_j} > \zeta_j\}$ *as the boxes associated with the left and right children of* j*. Define the box associated with the root as* $B_{root} := \mathbb{R}^D$*.*

Definition 6.7 *A decision tree on* $\mathbb{R}^D$ *is a hierarchical, binary partitioning of* $\mathbb{R}^D$ *and a rule for predicting the label of the test points given training data. It is represented by the tuple* $T = (T, \delta, \zeta)$*.*

Mondrian processes are very well suited to construct random decision trees. We can think of guillotine partitions as k-d trees, which are binary trees where every node is a k-dimensional point being used

for space partitioning in a k-dimensional space. In the case of the guillotine partitions, each node corresponds to a box in $\mathbb{R}^D$. Each non-leaf node has exactly two children whose boxes correspond to the two boxes obtained by cutting the box associated with the parent node with a hyperplane that is orthogonal to one of the D coordinate axes. Mondrian processes therefore naturally have a structure that matches the partition structure of random decision trees, but are in general unrestricted, infinite structures. Thus we define Mondrian trees as follows:

Definition 6.8 *A **Mondrian tree** is a restriction of a Mondrian process to a finite set of points. It can be represented by a tuple (T, δ, ζ, τ) where (T, δ, ζ) is a decision tree and $\tau = \{\tau_j\}_{j \in T}$ associates a time of split $\tau_{j \geq 0}$ with each node j.*

Mondrian trees differ from standard decision trees (i.e. CART algorithm) in a few important properties.

1. The splits are sampled independently of the labels.
2. Every node j is associated with a split time denoted by τ_j.
3. A lifetime parameter λ controls the total number of splits (comparable to maximum depth parameter for standard decision trees).
4. The split associated to an internal node j holds only within the smallest subbox B_j^x containing the data points of B_j.

A visualisation of Mondrian tree partitions is shown in figure 8. Details about the how Mondrian trees predict can be found in section 6.4.1.

A **Mondrian forest** is an independent collection $T_1, ..., T_M$ of Mondrian trees.

6.3 Implementation

An implementation of Mondrian Forests exists in `Python`, however the framework of this implementation is not transferable to Julia as `Python` uses object-oriented programming and Julia does not. Instead, Julia relies on data structures and functions operating on these data structures. We therefore developed a whole new framework to represent Mondrian trees and forests in Julia. In our implementation we put a special focus on two aspects. Firstly, we used the technical properties that make Julia unique to optimise our algorithms for speed and secondly, we payed great attention to the usability of the package and built a simple, convenient user interface as well as providing proper documentation.

Multiple dispatch programming gave our implementation its core structure. We operate on very few abstract types that have slightly different configurations in different scenarios. For example, the abstract type "Mondrian node" is used in all algorithms but adapted to keep different variables and containers. As the theoretical purpose of the structure does not change it makes sense to also keep the name for clear understanding.

Furthermore we sub-sectioned main functions into helper functions wherever possible, as shown in listings 3 and 4. On top of the gain of efficiency this also makes our code much more interpretable for the user. When using abstract types, we made sure to declare the types of variables and containers within the type. Usually type declaration is not needed in Julia as the compiler automatically knows the types of functions arguments, local variables and expressions. However, when defining abstract types the definition becomes necessary otherwise the compiler must infer them at runtime which slows computations. Lastly, the mathematical expressions in our helper functions are de-vectorised wherever possible.

An important part of any package is its ease-of-use. The Python implementation [7] uses a bash shell interface for all functionality. This method is compatible with the use of a high-performance cluster where a single script must be submitted for a job but not necessarily intuitive in its use on a standard machine. In the spirit of Julia we chose an interface which is compatible with the high-performance cluster usage scenario but also allows for rapid-prototyping and flexible dynamic code. Our interface consists of Mondrian tree and forest classifiers and regressors. These can be applied to machine

learning tasks using train! and predict! functions. The expansion of a Mondrian forest can be done using the function expand! which will incorporate all new data points automatically in a Mondrian tree classifier or a Mondrian Forest classifier respectively. The "train!" and "expand!" functions use Julia's parallelisation macros for training/expanding the individual Mondrian trees in parallel.

6.3.1 Sampling Mondrian Trees

To sample Mondrian trees we follow a modification of the outline of algorithm 9 [30] which is the 'paused-Mondrian' version of algorithm 2 in the same paper. A shortened version of our code can be seen in listing 3.

First we choose the split time τ of the current node or set it to λ if the set of labels is unique (paused-mondrians) (l. 3). If the split occurred in time, we then sample our split dimension and direction δ, ζ (l. 5). If both sides of the split contain data (Mondrian trees are strictly binary) we set up the left and right children with the split information (l. 11,12). As long as the boxes associated to each of the children contain data we recurse by sampling the children (l. 14,15). If the split didn't occur in time the node is made a leaf (l. 18).

As we descend down the tree, the data at each node decreases in size, as only a subset of the data points is present at the node. We take advantage of it and only search the labels present in the current Mondrian Block as opposed to searching the whole data set for each node when initialising the children (Algorithm 5 [30]). This divide-and-conquer adaption was added to the original outline of the algorithms to increase the efficiency.

Listing 3: Sample Mondrian Block code

```
1  function Sample_Mondrian_Block!(j,θ,λ,Data,Labels,Tree)
2      # get the appropriate split time
3      choose_τ(j,λ)
4      if j.τ < λ
5          d,x = sample_split_dimension(θ)
6          Data^R = find(Data[:,d] .> x)
7          Data^L = find(Data[:,d] .<= x)
8          # strictly binary tree
9          if (size(Data^R,1)>0 && size(Data^L,1)>0)
10             # children set up
11             j.right = set_up_node(j,θ^R,Labels[Data^R],Tree,%[true,false,false])
12             j.left = set_up_node(j,θ^L,Labels[Data^L],Tree,[true,false,false])
13             # recurse
14             Sample_Mondrian_Block!(j.left,θ^L,λ,Tree,Data[Data^L,:],Labels[Data^L])
15             Sample_Mondrian_Block!(j.right,θ^R,λ,Tree,Data[Data^R,:],Labels[Data^R])
16         end
17     # set j as leaf for time out/paused mondrian/non-binary
18     set_leaf(j)
19     return
20 end
```

6.3.2 Efficient Online Mondrian Forests

Mondrian Forests enable efficient online training[4] due to the fact that the distributions of the Mondrian trees are derived from the distribution of the Mondrian Process on $\mathbb{R}^D$. Mondrian Processes are projective, therefore the tree distributions are self consistent. This property is further described in algorithm 19 in [8]. It allows us to choose one out of three options to change each Mondrian tree in a forest to incorporate a new data point:

1. Introduction of a new split above an existing split (add a new node between parent and one of the children or add a new root)
2. Extension of the block associated to an already existing split to incorporate the new data point

[4] If a Mondrian forest is trained in an online fashion, the datapoints are not all considered at the same time but one by one. This has multiple advantages in the practical use of the algorithm, for example if data arrives sequentially or the underlying distribution changes continuously.

3. Splitting an existing leaf node into two children.

Usually, online random forest algorithms only perform the third operation. This variability in online Mondrian trees has potential for faster training times and a better trade-off between accuracy and computational time.
The algorithms our implementation is based on are algorithms number 3 and 10 in [30]. Here we present a shortened version of our code and explain the main points.

Listing 4: Extend Mondrian Block code

```
1  function Extend_Mondrian_Block!(T,λ,j,X,Y)
2      # the input node is a leaf
3      if sum(j.c .>0) == 1
4          Θ = update_intervals(get(j.Θ),X)
5          j.Θ
6              if findmax(j.c)[2] == Y       # -> case 2
7                  UpdatePosteriorCounts(j,Y)
8
9              else
10                 Sample_Mondrian_Block!(j,get(j.Θ),λ,T,p_data,p_labels) # -> case 3
11
12     # not a leaf
13     else
14         E = rand(Exponential(1/Extended_dimension(get(j.Θ),X)))
15
16         if τ_p + E < j.τ  # -> case 1
17
18             d,x = sample_extended_split_dimension(get(j.Θ),X)
19             Θ = update_intervals(get(j.Θ),X)
20             j_wave = init_j_wave(j,E,d,x,Θ)
21
22             j_prime = init_j_prime(j_wave,Y)
23             Sample_Mondrian_Block!(j_prime,get(j_prime.Θ),λ,T,A,[Y])
24         else # -> case 2
25             Θ = update_intervals(get(j.Θ),X)
26             j.Θ = Θ
27             Extend_Mondrian_Block!(T,λ,get(j.child),X,Y)
```

The function "Extend_Mondrian_Block!" is called by another algorithm [alg. 3, [30]] and takes three parameters: the tree to be extended, the lifetime parameter λ and the new data point (features X, label Y).

We first check whether the input node is a leaf (l. 3). If so, the extend of the box gets updated (l. 4). If after this extension all labels present at the node are still identical we leave this node as a leaf and the extension is finished, performing case 2 of the possible extension cases. Finally we then update the counts of classes at each node in the tree (l. 7). If the new label differs from the others the leaf gets sampled again and split into children (l. 10) (extension possibility 3). The input data for the extension is the data that was already present at the leaf together with the new data point. For efficiency reasons we decided to let paused leaves store the data they were sampled on as this avoids using the original data as an input for the extension.

If the input node is not a leaf we check if the extension occurred in time (l. 16). If so, we insert a new node j_wave between the node j we're currently looking at and its parent (extension possibility 1). If j is the root we replace the root. Helper functions sample split dimension and direction as well as the box associated to the new node (l. 18,19). We then finally initialise the new node as well as a second node j' which is a sibling to j. We arrange the initialisation such that j' always has the new data point in its associated box. It becomes a leaf holding the new data point (l. 23). If the extension doesn't occur in time we update the extend of the box of j (extension case 2) and then recurse on the child of j whose associated box has the new data point.

A special challenge while implementing this algorithm was the interface with the predict function as one needs to keep detailed track of the change of the root and leaf status of nodes as the tree changes.

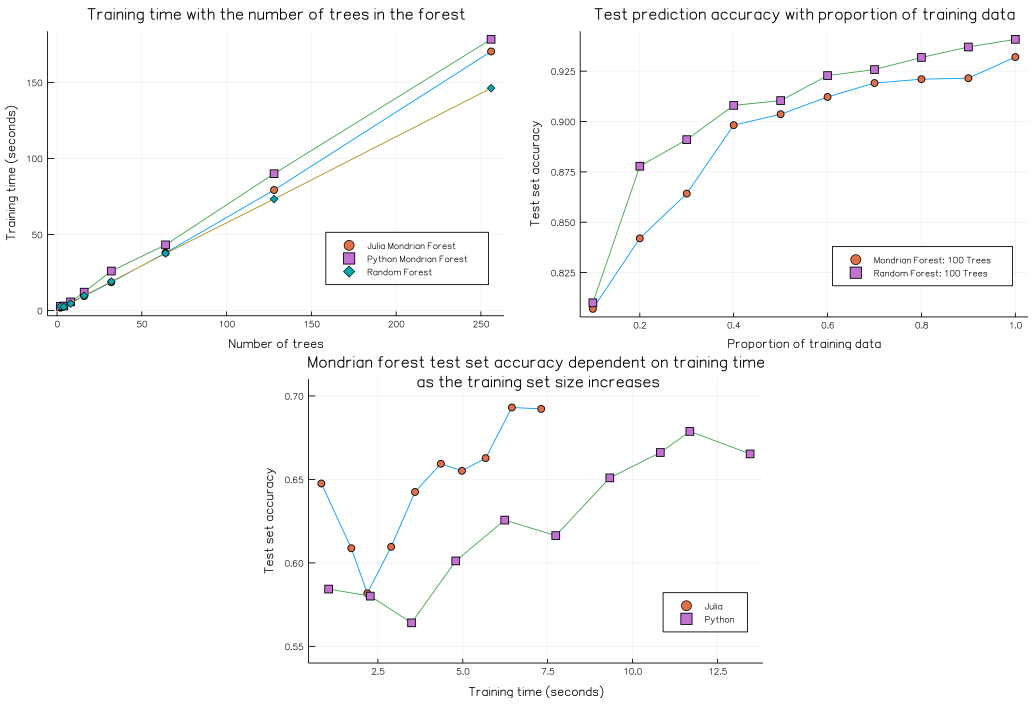

Figure 9: (Left) Training time for a forest with 100 trees, DecisionTree.jl random forest is used. (Right) prediction accuracy on the letters data set using a testing set of 5000 data points and increasing proportions of the remaining 15000 data points for training, we verify the same result as [30]. (Bottom) test set accuracy against training time on *dna* data set with all attributes.

6.4 Application of Mondrian Forests to Machine Learning Tasks

6.4.1 Classification

To perform classification tasks we need to define a way of associating a predictive label distribution to each leaf in the tree. The aim is to get for each leaf l a smoothed empirical distribution of labels for data points belonging to the box associated with l and nodes near l. This is achieved using a hierarchical Bayesian approach, meaning that a prior is chosen for the label distribution of each node which is similar to the label distribution of the parent. We then use marginalisation to find the final label distribution at the node. Details about this can be found in algorithm 11 in [30].
Mondrian forests can be used in the same way for classification tasks as standard decision trees. To simplify the usage, we defined train! and predict! functions that operate on the abstract types Mondrian_Forest_Classifier and Mondrian_Tree_Classifier, adapting the API to the one of popular machine learning packages like Scikitlearn [35]. Here, we present an example of classification on the real world data sets *letters*, and *dna* which are also used in [30] (figure 9).

On the *letters* data set we find our algorithm performs marginally faster than the equivalent Python implementation in terms of training time over the number of trees within the Mondrian forest. The accuracy is comparable to the random forest (C backend). These results match the results in [30] (figure 11, Appendix C). The time taken to achieve a certain accuracy on the *dna* data set is lower in our Mondrian forest than in the `Python` implementation. On the *dna* dataset we also achieve a higher accuracy overall and have a faster training time.

6.4.2 Regression

The major difference between the classification and regression tasks with Mondrian forests lies in the predict! function. To perform regression, we predict a Gaussian posterior over the labels Y instead of the class probability. Due to the special structure of the Mondrian trees the posterior will be a mixture of Gaussians. Details about this can be found in [31], algorithm 5.

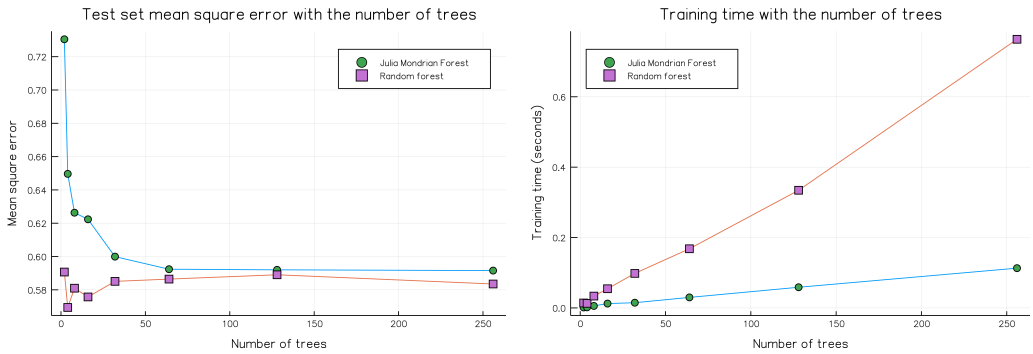

Figure 10: (left) Mean square error of Mondrian forest and and random forest algorithms depending on the number of trees. (right) Training time of Mondrian forest and random forest dependent on the number of trees.

For an example of a regression problem we use the (red only) wine-quality dataset [33]. We compare training time and performance (mean square error) for our Mondrian forest regressor and the `DecisionTree.jl` [38] random forest which were both pre-pruned to the same depth. The Python implementation [7] encountered impassable errors (NaN errors) during the training with more than 4 trees making a comparison impossible. For 2 and 4 trees the mean square error was about 0.77 and 0.65 respectively.
Our implementation shows a similar performance to the random forest for forests consisting of more than 16 trees. The training time for the Mondrian forest is consistently lower (figure 10).

7 MLJ

7.1 Motivation

As mentioned in *Section 3*, community packages tend to have a strong lack of structural consistency and there currently is no common interface connecting them. This is a big obstacle toward a growing machine learning community, especially when compared to what is available in other languages, such as `scikit-learn` in `Python` [35] or `Machine Learning R (MLR)` in `R` [10]. Altogether, this lead to the creation of `MLJ`, an attempt to fill this gap by providing a well-structured API to multiple packages, built to allow for very simple integration of new packages.

The syntax and structure of `MLJ` is strongly based around that of `MLR`, because it is the cleanest. Additionally, having a syntax they are already familiar with is appealing to users as it makes the transition simpler.

Although a very similar syntax was adopted, the back-end is completely redesigned. The main reason why is that Julia code should be written in a very differently form than `R` code, which is object-oriented. Furthermore, `MLR` suffers from an array of design flaws, some of which affect core parts of the execution like excessive memory allocation.

7.2 Structure

On the front-end, the interface is composed of a few core objects: `Learner`, `Task`, and `ParametersSet`, together with the core functions `learn`, `predict`, and `tune`. The `Learner` gives an abstract view of what the model should be and should use, whereas the `Task` fixes the task type (i.e. regression, classification, etc.), the data, and the target variable. The `ParametersSet` is used to give the set of parameter-values which should be fitted during the tuning.

On the back-end, the `learn` and `predict` simply take a `Learner` and a `Task`, whereas the `tune` also requires a `ParametersSet`. The tuning iterates over every parameter combination in the set, calling `learn` each time.

The framework is structured so that only four functions need to be written per wrapper:

- `getParameters`: used to get the list of parameters and their types.

- `makeModelname`[5]: used to check the validity of parameters, store them and initialise the object. It is required since `MLJ` takes advantage of the way function with parametric types are called by Julia.
- `learn`: given a model and parameters, how to train it.
- `predict`: given a trained model and data, how to predict.

Once the `make` function has been called, the user is in possession of an initialised model of a certain `Type`, which should should inherit from `MLRModel` and be unique to every wrapper. This is primordial for Julia to be able to find which of the several wrappers `learn` function needs to be called, which is done through parametric inference. Parametric types and inference are explained in more details in Appendix B.

7.3 Parameter Tuning

Parameter tuning is one of the key features of `MLJ`, allowing users to easily define a set of parameters and models to tune. It comes integrated with cross-validation, and allows the user to chose the re-sampling method as well as the evaluation metric to use.

The `ParameterSet` is simply an array of `Parameters`, which can be of two types: continuous and discrete parameters. For the continuous variables, the user specifies four values:

- `name`: the name of the parameter
- `lower`: the lower iteration bound
- `higher`: the higher iteration bound
- `transform`: how to transform the current iteration to get the final paramter value[6]

For the discrete set on the other hand, only two variables need to be specified:

- `name`: the name of the parameter
- `values`: array of values to check

The tuning will then check every combination of all the `Parameters` part of the `ParameterSet`, and return a storage variable with all the evaluations.

7.4 Issues Encountered

7.4.1 Loading of Packages

During the package review, we came across a package called `Orchestra.jl` [41], being very similar to what `MLJ` is attempting to be. This package had not been updated since Julia-0.3, and is completely deprecated. One issue we particularly noticed was that it tried to load all of its wrappers when initialised. This meant that a single error in a wrapper would break the whole interface. This issue is avoided in `MLR` by loading wrappers separately and only on user demand, through the function `load(wrapper_name)`. There is also a `loadAll` for convenience, which is not safe from this issue.

7.4.2 Model Types

While wrapping the `LIBSVM` [34] package, we noticed that some packages are developed so that all parameters are given at the training stage, with no previous initialisation, meaning that no "model" `Type` is defined pre-training. As mentioned, the framework requires developers to write a `makeModelname` function, which, in addition to checking the validity of parameters, must return an initialised model of a unique `type`. In the case where no such model is used, Julia cannot infer which `learn` function to call.

[5] *Modelname* should be replaced with the name of the package, for instance *makeLibsvm* when using the `LIBSVM` wrapper.

[6] A common example, to test the parameters at $0.001, 0.01, 0.1, 1$ would be lower bound of -3, higher bound of 0, and transform of $x \to 10^x$

An attempt was made to restructure the code to allow for the `make` function not to be required, but this turned out very complicated and caused errors in various parts of the code. The best solution was therefore simply to force developers encountering this issue to create a unique model `type`, used only to contain the training parameters.

7.4.3 Automatic Tuning

Automatic tuning is a very useful feature to tune continuous variables and available in most comparable packages. An attempt was made to implement it through simple gradient descent. The `JuliaDiff` [37] package in Julia provides first order differentiation through the use of the dual numbers. In Julia, these are defined as subtypes of `AbstractFloats`, but not of `Float`. The issue then occurs when packages force an argument to be of type `Float`, instead of `AbstractFloat`. Unfortunately this cannot be solved in `MLJ`, since the package being wrapped is the cause.

The best solution, which is the accepted convention in the Julia community but not applied by all developers yet, is to always make the type `AbstractFloats` when a `Float` is expected, since this will cause no difference whatsoever, but will allow dual numbers and other similar types to work properly.

Furthermore, an attempt was made to use numerical difference packages, such as `DiffEqDiffTools` [2], but preliminary tests found them to perform poorly, in addition to being slower, so they weren't investigated further.

7.5 Discussion and Future Work

`MLJ` is an attempt to fill the current gap for a common interface, as well as to make it easier for users to tune multiple learning models under one consistent structure. It is still in progress, with a first version, containing the core components, to be released in June 2018. A wrapper has also successfully and easily been made for the Mondrian Forest package discussed earlier. The clustering metrics implemented will also be available as soon as they're included in `MLMetrics` [5].

On the shorter term, some really big improvements can be made by wrapping around more models, in order to increase the overall practicality of the package for the community, and clean up the way classification and regression tasks are done. Additionally, Bayesian automatic tuning, present in `MLR` as `MLR-MBO` [11], could solve the current auto-tuning problem. Further more, it would be a great feature to be able to easily save a current environment, in order to provide a full, reproducible workflow. This could be done by integrating it to the OpenML[**?**] environment.

8 Conclusion

The main objective of the project was to positively contribute to the Julia machine learning community. In a preliminary review of core machine learning packages in Julia we identified educational and functional issues and then used a twofold approach to make improvements by providing educational instructions on the one hand and implementing missing functionalities on the other.

All results of the package review are available on Github for the Julia community to access and can therefore provide guidance for anyone wishing to use Julia for machine learning. Some of our notebooks found the attention of the package authors and are being adopted as official documentation like for example in `LearninigStrategies.jl`. We raised issues to the packages `LearningStrategies.jl`, `OnlineStats.jl`, and `MLMetrics.jl` that were either solved or are in the process of being solved. Ideally our work will lead to more people being enthused to try machine learning in Julia as they find more guidance and easier instructions, particularly for deep learning. We expect the package review collection to grow over time as the community takes it over. The effort done in this project will certainly help to solve the educational issues that were identified.

The contributions to the `MLMetrics.jl` [5] are going to be incorporated as soon as possible. Furthermore, the implementation of the Local Outlier Factor algorithm, which is one of the four important anomaly detection algorithms, is completed and can be published in an independent package specific to anomaly detection.

The implementation of Mondrian forests in Julia will add a new tile to the fragmented puzzle of machine learning packages. It hopefully helps to inspire a new standard in terms of usability and

documentation for people authoring packages in Julia. The functionality implemented decreases the gap in the decision tree landscape by providing an implementation that can compete with the equivalent in Python. Furthermore we provide the first online decision tree algorithm in Julia. Further work here includes the actual release of the package and the further improvement of the online algorithm. Ideally this package will have an impact on the solving of real-world problems.

Even though the project branched out in many different directions, the implementation of `MLJ` unites the different development efforts as it provides a common syntax not only for the contributions made by us but for all core machine learning packages in Julia. This package has the potential to become a central building block for further development in machine learning in Julia.

The project will be brought to the attention of the wider Julia community through conversations over the Julia slack channel, where there has recently been an upcoming interest on the topics we've been working on. Furthermore we will present our project to Mike Inns and some of his colleagues from Julia Computing who have a particular interest in machine learning in Julia. Most importantly, we will present a poster at JuliaCon 2018 including all our results.

9 Acknowledgments

We would like to thank our supervisor, Dr Sebastian Vollmer, for his mentoring throughout the project and our external partner Mike Innes from Julia Computing for his expertise and interest in the project. Then, we would like to acknowledge Dr Franz Kiraly, for his help in structuring the `MLJ` package to avoid all the mistakes made by `MLR`. Furthermore we would like to acknowledge Dr Gergo Bohner, and Thibaut Lienart. Finally, we would like to acknowledge the funding bodies, the Medical Research Council, the Engineering and Physical Sciences Research Council and the General Charities of Coventry.

References

[1] Piet Mondrian, Composition with Large Red Plane, Yellow, Black, Grey and Blue (1921). Retrieved from https://www.artsy.net/artwork/piet-mondrian-composition-with-large-red-plane-yellow-black-grey-and-blue, 2018-06-10, 13:59h.

[2] DiffEqDiffTools.jl: Differentiation tools using the performance overloads and traits for the DiffEq common interface. June 2018. original-date: 2017-01-11T19:55:48Z, https://github.com/JuliaDiffEq/DiffEqDiffTools.jl.

[3] Flux.jl: Relax! Flux is the ML library that doesn't make you tensor. June 2018. original-date: 2016-04-01T21:11:05Z, https://github.com/FluxML/Flux.jl.

[4] LearningStrategies.jl: A generic and modular framework for building custom iterative algorithms in Julia, June 2018. original-date: 2017-02-23T20:44:56Z, https://github.com/JuliaML/LearningStrategies.jl.

[5] MLMetrics.jl: Metrics for scoring machine learning models in Julia, May 2018. original-date: 2016-02-22T15:33:40Z, https://github.com/JuliaML/MLMetrics.jl.

[6] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[7] Balajiln. Code for Mondrian Forests (for classification and regression). *GitHub*, 2014. https://github.com/balajiln/mondrianforest.

[8] M. Balog and Y. W. Teh. The Mondrian Process for Machine Learning. *arXiv:1507.05181 [cs, stat]*, July 2015. arXiv: 1507.05181.

[9] J. Bezanson, A. Edelman, S. Karpinski, and V. Shah. Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, 59(1):65–98, Jan. 2017.

[10] B. Bischl, M. Lang, L. Kotthoff, J. Schiffner, J. Richter, E. Studerus, G. Casalicchio, and Z. M. Jones. mlr: Machine learning in r. *Journal of Machine Learning Research*, 17(170):1–5, 2016.

[11] B. Bischl, J. Richter, J. Bossek, D. Horn, J. Thomas, and M. Lang. *mlrMBO: A Modular Framework for Model-Based Optimization of Expensive Black-Box Functions*, 2017.

[12] M. Bouchet-Valat. FreqTables.jl: Frequency tables in Julia, Dec. 2017. original-date: 2014-06-28T13:40:19Z.

[13] L. Breiman. Random Forests. *Machine Learning*, 45(1):5–32, Oct. 2001.

[14] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984.

[15] M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Lof: Identifying density-based local outliers. In *PROCEEDINGS OF THE 2000 ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA*, pages 93–104. ACM, 2000.

[16] K. Carlsson. NearestNeighbors.jl: High performance nearest neighbor data structures and algorithms for Julia, May 2018. original-date: 2015-10-31T23:16:37Z, https://github.com/KristofferC/NearestNeighbors.jl.

[17] F. Chollet et al. Keras. https://github.com/fchollet/keras, 2015.

[18] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley, 1991.

[19] J. Day. OnlineStats.jl: Single-pass algorithms for statistics, June 2018. original-date: 2015-02-04T18:12:54Z, https://github.com/joshday/OnlineStats.jl.

[20] denizyuret. AutoGrad.jl: Julia port of the Python autograd package, June 2018. original-date: 2016-08-08T20:24:26Z, https://github.com/denizyuret/AutoGrad.jl.

[21] denizyuret. Knet.jl: Koç University deep learning framework, June 2018. original-date: 2015-09-29T23:42:37Z, https://github.com/denizyuret/Knet.jl.

[22] Fugro Roames and Julia Computing. Protecting the Electrical Grid – Julia Computing. https://juliacomputing.com/case-studies/fugro-roames-ml.html.

[23] Google Scholar. Citation statistics of Google Scholar. statistics retrieved from https://scholar.google.com/citationsuser=PJQNw9oA=end=gsmdcitadp=u=2Fcitations3Fview and https://scholar.google.com/citations?user=mXSv1UAAAAJhl=deoi=srad=gsmdcitadp=u=2Fcitations3Fview, 11-06-2018, 11:45h.

[24] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 edition, 2009.

[25] IBM and Julia Computing. Deep Learning for Medical Diagnosis – Julia Computing. Presented at SC16, https://juliacomputing.com/press/2016/11/18/supercomputing-2016.html.

[26] M. Innes. Flux: Elegant machine learning with julia. *Journal of Open Source Software*, 2018.

[27] M. Innes et al. Generic gpu kernels. 2017. http://mikeinnes.github.io/2017/08/24/cudanative.html.

[28] Julia Computing. Newsletter - January 2018 – Julia Computing. *Julia Computing Blog*, Jan. 2018. https://juliacomputing.com/blog/2018/01/04/january-newsletter.html.

[29] T. O. Kvalseth. Entropy and correlation: Some comments. *IEEE Transactions on Systems, Man, and Cybernetics*, 17(3):517–519, 1987.

[30] B. Lakshminarayanan, D. M. Roy, and Y. W. Teh. Mondrian Forests: Efficient Online Random Forests. *Advances in Neural Information Processing Systems 27 (NIPS), pages 3140-3148*, 2014.

[31] B. Lakshminarayanan, D. M. Roy, and Y. W. Teh. Mondrian Forests for Large-Scale Regression when Uncertainty Matters. *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2016.

[32] J. E. Nguyen Xuan Vinh and J. Bailey. Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance. *Journal of Machine Learning Research*, pages 2837–2854.

[33] P. Cortez, A. Cerdeira, F. Almeida, T. Matos, J. Rei. Modeling wine preferences by data mining from physicochemical properties, 2009.

[34] M. Pastell. LIBSVM.jl: LIBSVM bindings for Julia, May 2018. original-date: 2013-04-15T01:50:21Z, https://github.com/mpastell/LIBSVM.jl.

[35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python . *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[36] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, Mar. 1986.

[37] R. Goedman, J. Revels, L. Benet, T. Papamarkou. JuliaDiff. *GitHub*. http://www.juliadiff.org/.

[38] B. Sadeghi. DecisionTree.jl: Decision Tree Classifier and Regressor, June 2018. original-date: 2013-01-14T01:29:53Z.

[39] A. Saffari, C. Leistner, J. Santner, M. Godec, and H. Bischof. On-line Random Forests. In *2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops*, pages 1393–1400, Sept. 2009.

[40] A. Strehl and J. Ghosh. Cluster ensembles - a knowledge reuse framework for combining multiple partitions. *Journal of Machine Learning Research*, pages 583–617.

[41] svs14. Orchestra.jl: Heterogeneous ensemble learning for Julia, Dec. 2017. original-date: 2014-03-31T10:30:52Z, https://github.com/svs14/Orchestra.jl.

[42] J. E. Xuan Vinh Nguyen and J. Bailey. Information theoretic measures for clusterings comparison: Is a correction for chance necessary. In *ICML*, 2009.

[43] Y. Y. Yao. *Information-Theoretic Measures for Knowledge Discovery and Data Mining*, pages 115–136. Springer Berlin Heidelberg, 2003.

[44] Z. G. Z. Liu and M. Tan. Constructing tumor progression pathways and biomarker discovery with fuzzy kernel kmeans and dna methylation data. *Cancer Inform*, 6:1–7, 2008.

10 Appendix

.1 Appendix A

Listing 5: Neural Network using Flux(Static Approach)

```
#Defining the weights and biases of connections between input and hidden layer
W1 = param(rand(7, 13))
b1 = param(rand(7))
layer1(x) = W1 * x .+ b1

#Defining the weights and biases of connections between hidden and output layer
W2 = param(rand(1, 7))
b2 = param(rand(1))
layer2(x) = W2 * x .+ b2

#Defining the model for neural network
predict(x) = layer2((layer1(x)))
loss(x, y) = mean((predict(x) .- y).^2)

#Update function of the neural network
function update()
  η = 0.01
  for p in (W1, b1)
    p.data .-= η .* p.grad
    p.grad .= 0
  end
  for p in (W2, b2)
    p.data .-= η .* p.grad
    p.grad .= 0
  end
end
for i=1:10
    l = loss(x, y);
    back!(l)
    update()
end
loss(x, y)
```

Listing 6: Neural Network using Flux(Dynamic Approach)

```
#Defining the model for neural network by chaining layers, fixing learning rate, and choosing optimiser and loss

model = Chain(Dense(13, 7),
              Dense(7, 1))
η = 0.01
loss1(x, y) = mean((model(x) .- y).^2)
dataset = repeated((x, y), 10)
opt = SGD(params(model),η)
Flux.train!(loss1, dataset, opt)
loss1(x, y)
```

.2 Appendix B

In Julia, parametric types are a type of structure where one or more of the internal variables are not fixed, that is one can define a type as such:

```
immutable struct MyType{T}
    x::T
end
```

Here, T is any type, and it will be fixed when the constructor is called, for instance `MyType(0.5)` will parse 0.5 as a `Float64`, and therefore this instance of `MyType` will be of type `MyType{Float64}`.

Function can be made that only accept a particular type, for instance `MyFunction( a::MyTypeFloat64 )` will only be called if the parameter given is of type `MyType{Float64}`.

.3 Appendix C

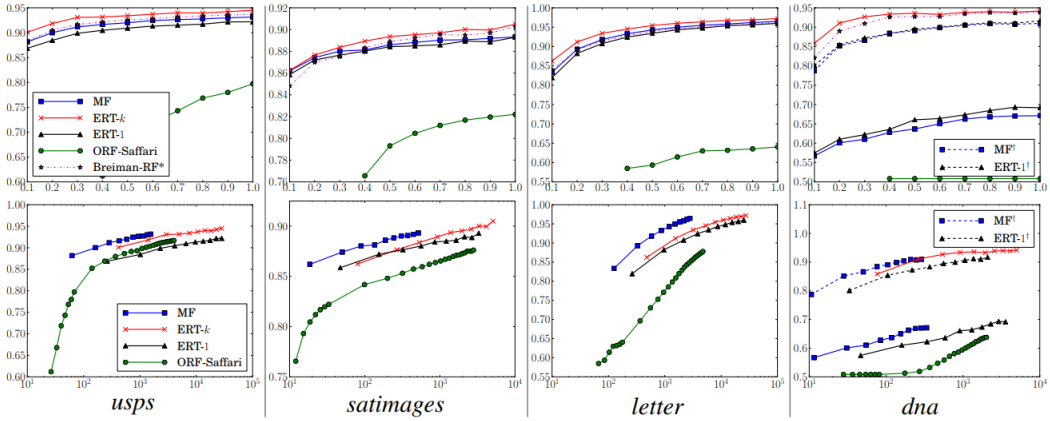

Figure 11: Results on various datasets: y-axis is test accuracy in both rows. x-axis is fraction of training data for the top row and training time (in seconds) for the bottom row. Source: [30]

.4 Appendix D

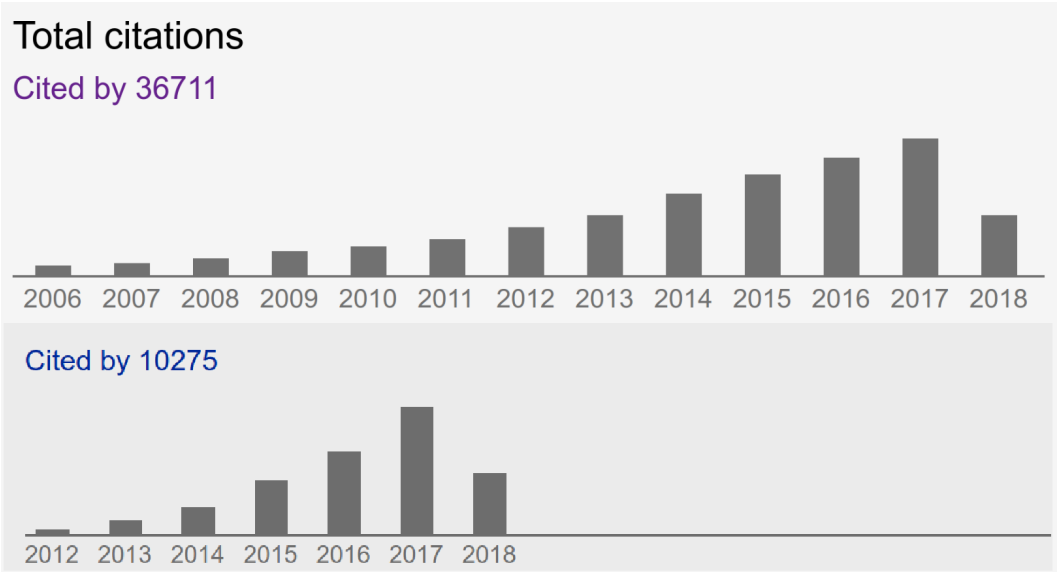

Figure 12: Top: Distribution of citations of "Random Forests" by Leo Breiman [13]. Bottom: Distribution of citations of "Scikit-learn: Machine learning in Python" [35] Source: Google Scholar [23]