Silas: A High-Performance Machine Learning Foundation for Logical Reasoning and Verification

Hadrien Bride[a], Cheng-Hao Cai[b,d,e], Jie Dong[c], Jin Song Dong[a,d], Zhé Hóu[a], Seyedali Mirjalili[f], Jing Sun[b]

[a]*Institute for Integrated and Intelligent Systems, Griffith University, Australia*
[b]*School of Computer Science, University of Auckland, New Zealand*
[c]*Dependable Intelligence Pty. Ltd. (Depintel), Australia*
[d]*School of Computing, National University of Singapore, Singapore*
[e]*Artificial Intelligence Innovation and Commercialisation Centre, National University of Singapore (Suzhou) Research Institute, China*
[f]*Centre for Artificial Intelligence Research and Optimisation, Torrens University Australia, Australia*

Abstract

This paper introduces a new high-performance machine learning tool named Silas, which is built to provide a more transparent, dependable and efficient data analytics service. We discuss the machine learning aspects of Silas and demonstrate the advantage of Silas in its predictive and computational performance. We show that several customised algorithms in Silas yield better predictions in a significantly shorter time compared to the state-of-the-art. Another focus of Silas is on providing a formal foundation of decision trees to support logical analysis and verification of learned prediction models. We illustrate the potential capabilities of the fusion of machine learning and logical reasoning by showcasing applications in three directions: formal verification of the prediction model against user specifications, training correct-by-construction models, and explaining the decision-making of predictions.

Keywords: high-performance machine learning, ensemble trees, explainable artificial intelligence, logical reasoning

---

Email addresses: `h.bride@griffith.edu.au` (Hadrien Bride), `chenghao.cai@auckland.ac.nz` (Cheng-Hao Cai), `jacob@depintel.com` (Jie Dong), `dongjs@comp.nus.edu.sg` (Jin Song Dong), `z.hou@griffith.edu.au` (Zhé Hóu), `ali.mirjalili@laureate.edu.au` (Seyedali Mirjalili), `jing.sun@auckland.ac.nz` (Jing Sun)

1. Introduction

Machine learning has enjoyed great success in many research areas and industries, including entertainment (Gomez-Uribe and Hunt, 2016), self-driving cars (Eliot and Eliot, 2017), banking (Turkson et al., 2016), medical diagnosis (Kononenko, 2001), shopping (Cumby et al., 2004), among many others. However, the wide adoption of machine learning raises the concern that most people use it as a "black-box" in their data analytics pipeline. The ramifications of the black-box approach are multifold. First, it may lead to unexpected results that are only observable after the deployment of software products. For instance, Amazon's Alexa offered prohibited contents to a child (Post, 2016), IBM's Watson recommended "unsafe and incorrect" cancer treatments (Ross and Swetlitz, 2018), etc. Some of these accidents result in lawsuits or even lost lives with an immeasurable cost. Second, it prevents the adoption in some applications and industries where an explanation is mandatory or certain specifications must be satisfied. For example, in the USA, it is required by law to give the reason why a loan application is rejected. Additionally, recent machine learning models, such as neural networks, often need considerable computational resources to run. For example, in automatic speech recognition, acoustic models can be multi-layer neural networks with more than ten millions of parameters that require several weeks to train on CPUs or several days to train on GPUs (Hinton et al., 2012). Although these models can achieve excellent prediction accuracy, industrial applications of such resource-consuming models are still uneconomical. In the industry, there is a need for machine learning models that are not only accurate but also explainable, verifiable and resource-efficient.

A fusion of machine learning and logical reasoning. In recent years, eXplainable AI (XAI) has been gaining attention, and there is a surge of interest in studying how prediction models work and how to provide formal guarantees for the models. A common theme in this space is to use statistical methods to analyse prediction models. On the other hand, Bonacina recently envisaged that *automated reasoning* could be the key to the advances of XAI and machine learning (Bonacina, 2017). This direction aligns well with our interest of building a new machine learning tool with logic and reasoning as the engine to produce "white-box" prediction models. A "white-box" machine learning method in our vision should feature the following key points:

Explainability: The inner workings of produced predictive models should be interpretable, and the user should be able to query the rationale behind the predictions.

Verifiability: The compliance of the produced predictive models with respect to user specifications should be formally verifiable.

Interactability: Data engineers should be able to guide the learning phase of predictive models so that the models conform with given specifications.

Efficiency: Predictive models should only consume reasonable resources to complete learning and prediction tasks.

Ensemble Trees. Towards this direction, we have been searching for a suitable machine learning technique that (1) has excellent predictive performance and (2) is ideal for logical reasoning and formal verification. We have found that some techniques show outstanding performance but are difficult to manifest. For example, neural networks can have multi-layer architectures that model nonlinear data features but are hard to explain using formal logic. On the other hand, linear methods are easy to explain because they can be described using linear equations. However, data in the real world are often nonlinear, so linear methods often do not perform well. Some techniques have a solid probabilistic reasoning foundation, e.g., Bayesian methods, but it is difficult to use formal logic to verify and explain these probabilistic models (Bishop, 2007). Different from the above machine learning techniques, ensemble trees learn data features using tree architectures, where each branch of a tree has specific meanings that can be represented using formal logic. Moreover, ensemble trees have excellent predictive performance that is sometimes better than deep learning on *tabular data* (Pafka, 2018). Further, ensemble trees require less data pre-processing because symbolic representations can be directly taken as input. They are fundamentally different from neural networks that require data vectorisation as a pre-processing step. Ensemble trees have attracted much attention in machine learning applications. A comparison of millions of machine learning models on Kaggle showed that ensemble trees are the preferred methods for structured data, while neural networks and deep learning dominate unstructured problems (Harasymiv, 2015).

Contributions. The current gap in the literature is the lack of understanding of the internal mechanism of ensemble trees and their *perceived* black-box nature, which make them impractical in critical applications (e.g. medicine, law and defence) as discussed above. This gap motivated us to develop a new machine learning tool named *Silas*, which is a fusion of ensemble trees learning and automated reasoning. The first part of this paper has the following contributions.

- We have developed the Silas toolkit to solve classification and regression problems in machine learning. Silas Education version has been made public and can be downloaded from our website. [1]
- We demonstrate that effective and efficient machine learning models can be built with a formal and explicit semantics that support automated reasoning.
- Silas targets high-performance applications of ensemble trees. Its predictive performance is often better than industrial leaders of similar techniques. Silas improves ensemble trees using a number of customised algorithms, e.g., decision trees with a logical foundation, various sampling, weighting, and voting algorithms.
- Silas's high-performance computing mechanisms are developed using our in-house C++ functional & parallel programming framework and data storage & management library, which are efficient and outperform competitors in term of time and memory consumption. As machine learning becomes more prevalent

[1]Silas Edu can be downloaded via Dependable Intelligence: `https://www.depintel.com/silas_download.html`. Silas Edu supports binary classification and machine learning + logical reasoning functionalities in Section 6. Moreover, editors and reviewers can access Silas's multi-class classification and regression functionalities via `https://depintel.com/downloads/Private/SilasDemo_0-8-7_linux_x64_8T.zip`, and relevant documents via `https://www.depintel.com/documentation/v087/_build/html/index.html`.

in everyday applications, Silas will provide increased productivity with lower operating costs.

Additionally, we explore the applications of logical reasoning by showcasing the following four proofs-of-concept that illustrate the capabilities of the *machine learning + logical reasoning* direction that we are currently pursuing:

Model Audit concerns how to formally verify the correctness of *very large* prediction models. Use cases of this module include: (1) checking whether a prediction model meets various "soft" and "hard" criteria and (2) grading the model based on the results.

Enforcement Learning concerns how to train prediction models that are correct-by-construction. A use case of this module is to ensure the satisfiability of "hard" criteria.

Model Insight concerns how to analyse a prediction model and give a general idea of how the model makes predictions on each class. A use case of model insight is to check whether the prediction model is understandable by users and is consistent with the user's domain knowledge.

Prediction Insight concerns how to explain the decision-making of individual predictions by relating them to their significant predictors. A use case of prediction insight is to enable machines to explain predictions for users.

The four proofs-of-concept pave the foundation for more advanced reasoning and verification approaches that we are planning to develop in the future. Although the latest version of Silas supports both classification and regression, the former is a closer fit to the logical reasoning component as they both deal with discrete math and algebra. Therefore, the remainder of this paper is focused on classification tasks. We leave explanation and verification for regression tasks to future work.

Paper organisation. This paper is organised as follows. Section 2 reviews related work. Section 3 provides preliminary knowledge of decision trees and ensemble methods. Section 4 introduces machine learning algorithms in Silas. Section 5 provides experimental results on Silas's ensemble trees. Section 6 describes Silas's logical reasoning abilities to explain and verify machine learning models. Section 7 concludes this paper.

2. Related Work

Explainability is one of the essential differences between decision trees and connectionist models (e.g., artificial neural networks), and this enables decision trees to form core components of expert systems. For example, in the state-of-the-art Alzheimer's disease diagnosis system, decision trees are used to classify meta-data such as fractional anisotropy and mean diffusivity values of domain regions, while convolutional neural networks are used to model the features of magnetic resonance imaging (De and Chowdhury, 2020). Node partitioning functions, which dominate the explainability of decision trees, are usually described using logical expressions. The use of different logical expressions can form different representations in expert systems. The above concept has led to decision trees with polynomial partitioning functions, which has been applied to some industrial scenarios, including concrete strength prediction and energy efficiency analysis, and surpassed a number of traditional decision trees in terms of prediction accuracy in regression analysis (Yang et al., 2017). Besides, the explainability of decision trees enables the description of regression processes using systems of inequalities, which are used to free disposal hull in microeconomics (Esteve et al., 2020).

There are many implementations of ensemble trees, such as XGBoost (Chen and Guestrin, 2016), H2O (Cook, 2016) and Ranger (Wright and Ziegler, 2017). The latter two are more relevant to the bagging implementation of Silas. H2O is a Java implementation that is shown more efficient than other tools such as the R implementation, the Python implementation and Spark; it also gives better predictions than XGBoost on the flight dataset (Pafka, 2018). Ranger is a fast implementation of random forest written in C++ that is designed to handle high dimensional data. There have also been numerous developments on improving ensemble trees in academia, such as Weighted Oblique Decision Trees (Yang et al., 2019), Hoeffding Tree (Zhang and Ntoutsi, 2019), Very Fast Decision Tree (Losing et al., 2018) and Distinct Decision Trees (Ruggieri, 2017). Silas outperforms them for most datasets surveyed in this paper.

It is non-trivial to introspect and extract logical semantics from the structure of decision trees and improve their representations. For example, to improve explainability of decision trees, Iorio et al. (2019) have defined a path length proportional to the impurity decay. When partitioning a node, as both the path length and the impurity are considered, a decision tree will tend to grow branches with shorter paths and abandon branches with longer paths. The above strategy can lead to shorter decision paths that require shorter expressions to describe the semantics of data. Moreover, on complex classification tasks, as a white-box model, decision trees may be difficult to use sufficiently complex representations to distinguish features of data. To solve this problem, Piltaver et al. (2021) have attempted to replace some leaf nodes of decision trees with black-box models, leading to tree classifiers with both interpretable upper layers and accurate lower layers. Further, as the rectifier linear unit (Glorot et al., 2011) in connectionist models preserves both explainability and accuracy, it can be used by the partitioning functions of decision trees (Tao et al., 2020). Additionally, another way to improve decision trees is to improve training strategies, e.g., the use of boosting approaches such as AdaBoost (Freund and E Schapire, 1999), XGBoost (Chen and Guestrin, 2016) and FDT-Boost (Barsacchi et al., 2020) and the use of artificial training data generated from existing training data (Rodríguez et al., 2020). Following this trend, we have developed our own implementation of ensemble trees (Bride et al., 2018) by using a tree structure that is amenable to logical reasoning. We show that our implementation is much faster and more memory efficient than both H2O and Ranger. The literature on ensemble trees and machine learning is rich, and we will only focus on a subset that is related to the interpretability and verification of machine learning.

Although not yet substantial, there have been early steps taken towards understanding prediction models and providing guarantees for them. For instance, the Lime tool (Ribeiro et al., 2016) is able to provide local linear approximations of various types of prediction models and show which features are the most decisive in predictions. Similarly, Hara and Hayashi (Hara and Hayashi, 2018) proposed post-processing for ensemble trees to obtain an approximation of the model with probabilistic interpretations. Another interesting work is Lundberg et al.'s SHAP method (Lundberg and Lee, 2017), which uses the game theory to obtain *consistent* explanations. Ehlers (Ehlers, 2017) developed an SMT based method to verify linear approximations of feed-forward neural networks. While these methods have shown potential in interpreting and verifying predictions, they still treat the prediction model as a black-box and try to analyse or verify an approximation of the black-box. On the contrary, we are interested in treating the prediction model as a white-box and studying the internal mechanism of prediction models.

A logical approach seems more natural for understanding the internal structure of decision trees because decision trees are inherently connected with logical semantics and are very similar to binary decision diagrams (BDDs) which are widely-used in implementations of logical systems such as theorem provers (Goré et al., 2014) and model checkers (Cimatti et al., 2002). Caruana et al.'s work (Caruana et al., 2015) attempts to explain how a boosting machine makes predictions by analysing the logical conditions in the decision trees. However, at the time of writing their Microsoft project was very young, and the cited paper did not give enough details on interpretability.

Complementary to the above work, we are also interested in providing formal guarantees for prediction models. Törnblom and Nadjm-Tehrani (Törnblom and Nadjm-Tehrani, 2019) proposed a method to extract equivalent classes from random forest and verify that the input/output of the model satisfies safety properties. Their approach considers all possible combinations of results from all the trees, which means they have to verify $2^{d \cdot B}$ equivalent classes of the results where d is the depth of trees and B is the number of trees. The advantage of their approach is that they can give bi-directional results: (completeness) if the constraint is satisfied, their verification returns positive, and (soundness) if the verification returns positive, the constraints must be satisfied. The disadvantage of their approach is the high complexity and the verification of 25 trees of depth 20 in practice. Our verification approach focuses on soundness; as a result, we can simplify and parallelise the verification in order to verify very large models.

3. Preliminaries

This section provides the essential definitions of decision trees and their ensembles for classification. The focus is on subtle differences between our implementation and the common definitions in the literature. Specifically, we give a logic-oriented definition of decision trees that facilitates the reasoning and verification of prediction models.

3.1. Decision Trees With a Logical Foundation

In the context of supervised learning, a structured dataset for classification is defined as set of *instances* of the form $\langle x, y \rangle$ where $x = \langle x_1, ..., x_n \rangle$ is an input vector of $n \in \mathbb{N}$ values often called *features* and y is an outcome value often called *label*. We denote by X the feature space and Y the outcome space.

A decision tree is a tree structure composed of internal nodes and terminal nodes called leaves. Internal nodes are predicates over the variables $\{x_1, ..., x_n\}$ corresponding to features. Leaves are sets of instances. Without loss of generality, we focus on binary trees. Internal nodes have two successors respectively called the left and right child nodes. By convention, let $p : X \to \{\top, \bot\}$ be an internal node v, the right (resp. left) child node of v is the root of a decision (sub)tree whose set of leaves L is a set of sets of instances such that $\forall\, x \in \bigcup\{l \mid l \in L\}, p(x) = \top$ (resp. $p(x) = \bot$). Given a decision tree, any input vector is associated with a single leaf. Further, let $D(Y) : Y \to \mathbb{R}^{\geq 0}$ be the set of distributions over Y. Every given leaf l is associated with a distribution $d_l \in D(Y)$ such that for all $y \in Y$, $d_l(y)$ is the weight associated with the outcome y in l. A decision tree is, therefore, a compact representation of a function of the form $X \to D(Y)$.

Let $t : X \to D(Y)$ be a tree and $x \in X$ be an input vector. Further, let $M : D(Y) \to Y$ be a function such that $\forall\, d \in D(Y)$, $M(d) = y_{max}$ such that $d(y_{max}) = max\{d(y) \mid y \in Y\}$. The outcome predicted by t for the input vector x is the outcome value $M(t(x))$.

In Silas, similarly to popular greedy approaches such as C4.5 (Quinlan, 1993), trees are constructed by recursively splitting an input dataset until a stopping criterion is satisfied. The splitting predicates are chosen based on the *information gain* they provide, a measure which is computed by comparing the *entropy* (Shannon, 1948) between the parent node and the child nodes. Contrary to generic decision trees grown by approaches such as C4.5 (Quinlan, 1993), the predicates of internal nodes are logical formulae described below.

A *logical formula* in Silas is defined as an extension of propositional logic with arithmetic terms and comparison operators. The semantics of the logical language follows that of standard arithmetic and propositional logic. An *arithmetic term* T is defined below where c is a constant (discrete or continuous value) and var is a variable corresponding to (the name of) a feature:

$$T := c \mid var \mid -T \mid sqrt(T) \mid T+T \mid T-T \mid T*T \mid T/T \tag{1}$$

A *Boolean formula* F takes the following form where C denotes a set of constants and $\oplus$ is the exclusive disjunction operator:

$$\begin{aligned} F :=& \top \mid \bot \mid var \in C \mid \\ & T < T \mid T \leq T \mid T = T \mid T > T \mid T \geq T \mid \\ & \neg F \mid F \wedge F \mid F \vee F \mid F \rightarrow F \mid F \oplus F \end{aligned} \tag{2}$$

In the implementation, we use $var \in C$ to express formulae of *nominal* features, which have discrete values, and use (in)equalities to express formulae of *numeric* features, which have continuous values.

3.2. Ensemble of Decision Trees

We adopt Cui et al.'s definitions (Cui et al., 2015). Let an ensemble be a set of decision trees of size T. It gives the weighted sum of the trees as follows:

$$E(x) = \sum_{i=1}^{T} w_i \cdot t_i(x) \tag{3}$$

where E is the function for the ensemble, w_i and t_i are respectively the weight and function for each tree. We give some examples of ensemble trees below.

Bagging. Each decision tree is trained using a subset of the dataset that is sampled uniformly with replacement. The remaining instances form the out-of-bag (OOB) set. When selecting the best formula at each decision node in a tree, only a subset of the features are considered. This is commonly found in algorithms such as Random Forest (Breiman, 2001). Bagging grows large trees with low bias, and the ensemble reduces variance.

Boosting. Boosting trains weak learners, i.e., small trees, iteratively as follows:

$$E_{i+1}(x) = E_i(x) + \alpha_i \cdot t_i(x) \tag{4}$$

where t_i is the weak leaner trained at iteration i and α_i is its weight. The final ensemble is thus a special case of $E(x)$ above where w_i is α_i. The ensemble reduces bias. AdaBoost (Freund and E Schapire, 1999) is a well-known example of a boosting approach.

3.3. Silas

The remainder of the paper is focused on bagging, although we also implement boosting approaches for comparison. Contrary to the vanilla Random Forest algorithm (Breiman, 2001), we *may not* grow each tree to maximum depth, which is why we store the instances' distribution at the leaf nodes rather than a single outcome value. Each tree is weighted by its performance on the OOB sample. To obtain a prediction, the majority of Silas methods aggregate weighted *votes for each class* (soft-voting) on the leaves instead of weighted *voted outcomes* (hard-voting). We will discuss a case that uses hard-voting independently. Another unique tweak in Silas is that decision trees are formulated in a logical language which belongs to a subset of first-order logic. This helps with our overarching objectives of explainability and verifiability. As a result, we also treat nominal features differently than other implementations: we directly use set membership to encode nodes for nominal features.

Some industrial users of Silas have specific requirements on computational performance and hardware. For example, some require that the software must be able to perform learning and prediction locally on their existing consumer-grade hardware than on clusters hosted outside. As a result, Silas is built with a focus to be fast and memory efficient. The experiment in this paper demonstrates that the users of Silas can perform machine learning for large datasets on consumer-grade machines.

3.4. Baseline

There are a large number of variants of Random Forest and AdaBoost in the literature. Recent examples include the Weighted Oblique Decision Trees (Yang et al., 2019), XBART (He et al., 2019), Adaptive Neural Trees (Tanno et al., 2019), AugBoost (Tannor and Rokach, 2019), etc. However, obtaining the source code for all these implementations and setting up a platform that can run them in a correct configuration is non-trivial. Further, many recent papers set up their experiment on clusters with many cores, hundreds of GBs of memory and expensive GPUs, which are against the aforementioned resource requirement of Silas.

We will compare various combinations of algorithms in the Silas framework with two well-known implementations in the industry: H2O (Cook, 2016) and Ranger (Wright and Ziegler, 2017).

4. Machine Learning in Silas

This section describes the machine learning algorithms used in Silas. We also discuss how high-performance computing is incorporated to yield faster and more resource-efficient computation for machine learning.

4.1. Customised Algorithms in Silas

We describe a number of customised sampling and weighting algorithms used in Silas for building ensemble trees. We divide the algorithms into tree-level algorithms, which work within the process of building a single tree, and forest-level algorithms, which are in the tree ensemble, or forest building stage. The user can choose any combination of a tree-level algorithm and a forest-level algorithm in the Silas framework.

4.1.1. *Tree-level Algorithms*

Greedy Narrow Tree (GT). Similarly to Random Forest (Breiman, 2001), this class of trees is grown in a greedy fashion and consider only a subset of features at each split. For classification tasks, the default settings randomly select $\sqrt{D}$ features, where D is the dimension, i.e., the number of columns, of the original dataset. For regression tasks, this parameter is $D/3$ by default. When expanding a leaf node, it selects, for each subsampled feature, the best cut-point using information gain, then choose to split using the feature whose best cut-point has the best overall *information gain*. However, contrary to vanilla Random Forest, the information gain of cut-points are, for efficiency reasons, evaluated on a sample of the data-points of the leaf. It is indeed often possible to gain significant knowledge about an overall population cost-effectively by studying a sample. The size of the sample is determined as a function of the prevalence of the minor class. More specifically, given a desired level of precision (i.e., the margin of error) e, the desired confidence level in Z-value, and the proportion of the minority p, the size n of the sample is computed using Cochran's formula (Cochran, 1977):

$$n = \frac{Z^2 p(1-p)}{e^2}. \tag{5}$$

Empirically, Cochranâ's formula is especially appropriate in situations with large populations. Note that the outcome distribution change at each new node. It follows that the sample size needs to be updated for each split. This method confers to the overall learning approach a highly adaptative sampling mechanism.

Random Tree (RT). Similarly to Greedy Narrow Tree, this class of trees is grown in a greedy fashion and consider only a subset of features at each split. However, when expanding a leaf node, it selects, for each subsampled feature, a single random split within its domain.

4.1.2. *Forest-level Algorithms*

Since forest-level algorithms are the focus of this paper, we further divide these algorithms into the ones based on sampling data points, the ones based on weighting data points, and the ones based on voting the results. These algorithms do not necessarily exclude each other; one can actually create a hybrid of these algorithms. However, there would be too many combinations to reasonably survey in this paper, so we choose the following subset based on our experience from real-life datasets.

Sampling-based Algorithms

Uniform Balancing (UB). UB can uniformly undersample the majority class(es) instances to match the size of the minority class. In Silas, class balancing is performed on top of dataset subsampling, which is indicated by a parameter called "sampling proportion". For example, when sampling proportion is 0.8, *UB* yields 80% of randomly selected minority instances and the same amount of randomly selected instances for all other classes.

No Balancing (NB). No balancing between classes. If the sampling proportion is 1.0, *NB* will use all the instances in the dataset. If the sampling proportion is 0.632, then *NB* will use roughly the same amount of randomly sampled instances as bagging *but the sampling is without replacement*.

Prototype Sampling (PS). We are interested in investigating how prototype selection based algorithms, particularly the hybrid ones such as the IB3 (Aha et al., 1991), benefit ensemble trees learning. Unfortunately, those algorithms often rely on computing the nearest neighbour of a data point or the centroid of a set of data points, and these operations are too costly for our use case. For example, the Condensed Nearest Neighbour (CNN) algorithm (Hart, 1968) involves so many computations of the nearest neighbour that it takes much longer time to finish than the entire training time of Silas. Even the Fast Condensed Nearest Neighbour (FCNN) algorithm (Angiulli, 2007), which requires $O(|T| \cdot |S|)$ distance computations where T is the training set, and S is the prototype set, is too slow for our use cases. Thus, to obtain a sampling method that does not hinder the speed of Silas, we have to sacrifice those operations that often lead to a smaller size of the sample or better classification performance.

Instead of computing the nearest neighbour of a data point from a prototype set, we have to resolve to find an approximation of the nearest neighbour. We thus propose a variant of the CNN algorithm that samples k instances from the prototype set and find the nearest neighbour within this subset. The algorithm is named “k-random-CNN” (*krCNN*) and is presented in Algorithm 1.

Algorithm 1: The *krCNN* Algorithm.

```
Data: a training set $T$.
Result: a set $S$ of prototypes.
for each x in T do
    if S contains less than 2 items or class(x) is minority then
        add $x$ to $S$;
    else
        $S_k \leftarrow$ randomly select with replacement $k$ instances from $S$;
        $y \leftarrow$ nearest neighbour of $x$ in $S_k$;
        if class(x) $\neq$ class(y) then
            add $x$ to $S$;
        end
    end
end
```

Training Time (s)	*UB*	*NB*	5rCNN	3rCNN	1rCNN
flight (*RT*)	12	24	34	29	18
flight (*GT*)	17	34	40	34	23
creditcard (*RT*)	6	90	126	111	64
creditcard (*GT*)	5	12	126	108	63

Table 1: Computational performance of *krCNN* on the flight and creditcard dataset.

We evaluate the value of k and observe its computational cost and the sampling result on selected datasets. In particular, we consider two datasets: the 1 million flights dataset (Pafka, 2019) and the creditcard dataset (OpenML, 2019). The ratios of the majority against the minority in these two datasets are respectively 4:1 and 577:1. Table 1 shows the time spent on training 100 decision trees of leaf size 64 instances. We separate the cases where the tree building method is *RT* and *GT*. *UB* is the fastest because it undersamples the majority class to be the same size as the minority class, thus the resultant training set is usually very small. Even when k is as small as 10, despite yielding a much smaller sample set than the original training set, the overall training time is significantly longer than *NB*, which uses all data for training. The training time decreases as k is reduced to 3, in which case the training time is comparable to *NB*. Because of the extreme imbalance in the creditcard dataset, the *krCNN* variants have a more visible performance hit than for the flight dataset.

The above empirical study shows that the computation time of *krCNN* is only satisfactory when $k < 3$. Inspired by various methods in the literature that exploits triangular relations between data points, such as Tomek Link undersampling (Tomek, 1976), we propose the prototype sampling (*PS*) algorithm in Algorithm 2, which is a modification of 2rCNN.

Algorithm 2: The Prototype Sampling Algorithm.

```
Data: a training set $T$.
Result: a set $S$ of prototypes.
for each $x$ in $T$ do
    if $S$ contains less than 2 items or class($x$) is minority then
        add $x$ to $S$;
    else
        let $y_1$ and $y_2$ be two random instances from $S$;
        // Function $d()$ computes the Euclidean distance between two points.
        if $(d(x,y_1) > d(y_1,y_2)$ and $d(x,y_2) > d(y_1,y_2))$ or
        $(d(x,y_1) < d(y_1,y_2)$ and class($x$) $\neq$ class($y_1$)) or
        $(d(x,y_2) < d(y_1,y_2)$ and class($x$) $\neq$ class($y_2$)) then
            add $x$ to $S$;
        end
    end
end
```

The PS method yields 476,594 (59%) and 72,011 (28.1%) majority instances for the flight and creditcard dataset respectively. The sampled size is much bigger than the *krCNN* method. Computationally it is the same as 2rCNN barring a few more conditions in the **if** statement. The logic behind these conditions are as follows: If $d(x,y_1) > d(y_1,y_2)$ and $d(x,y_2) > d(y_1,y_2)$, then x is "far away" from y_1 and y_2, and it may have new information compared to instances in S. If $d(x,y_1) < d(y_1,y_2)$ and class(x) $\neq$ class(y_1), then x is "closer" to y_1 than y_2 is, and x is of a different class than y_1, which means it may provide new information compared to instances in S. The last case is symmetric.

Since the value of k in the above case is very small, *PS* not only selects instances near decision boundaries but also selects instances far from boundaries. Thus, the behaviour of *PS* is more similar to a hybrid method than a condensation or edition method.

Weighting-based Algorithms

Weighted Cascade (WC). BalanceCascade (Liu et al., 2009) is an iterative approach that undersamples previously correctly classified instances to form the training set for the next iteration. It leads to very good predictive performance compared to other sampling algorithms (More, 2016). However, a naïve implementation of BalanceCascade showed poor performance in the Silas framework, because decision trees often overfit and predict a large portion of instances correctly, and the resulting training set for later iterations becomes too small too quickly. Consequently, we propose to modify the weight of incorrectly classified instances rather than removing correctly classified ones. In the Silas framework, adding the weight of an instance by 1 is equivalent to adding a *virtual* copy of the instance. This way of "increasing" the size of the dataset does not increase the training time of Silas. Our new algorithm, called Weighted Cascade, is presented in Algorithm 3.

In the discussion and experiment below, we shall use the following weight modification:

$$w_i \leftarrow w_i + 3, \tag{6}$$

The value 3 is determined based on empirical results from many public datasets and

Algorithm 3: The Weighted Cascade Algorithm.

```
Data: a training set $T$.
Result: a set $E$ of decision trees.
while current number of trees $< |E|$ do
    let $t$ be the number of allowed parallel threads;
    train $t$ decision trees that form a batch $B$;
    predict the training set using $B$;
    for each misclassified instance $i$ do
        change the weight of $i$ with $w_i \leftarrow w_i + C$;    // $C$ stands for a constant.
    end
    add $B$ to the ensemble $E$;
end
```

private projects undertaken by Depintel. The reader can use other methods, such as a weight increase that depends on the predictive performance of B.

AdaBoost (AB). Iteratively modifying the weight of instances is very close to boosting. For example, the AdaBoost (Schapire, 2013) algorithm initialises the weight of instances to $1/n$ where n is the total number of instances. In each iteration, it trains a weak learner h and obtains weighted classification error ε by

$$\varepsilon = \sum_{i=1, h(x_i) \neq y_i}^{n} w_i. \tag{7}$$

The weight for the new tree is

$$\alpha = \frac{1}{2} \ln \frac{1-\varepsilon}{\varepsilon}, \tag{8}$$

and the weight for each data instance is updated by

$$w_i = w_i \cdot e^{-y_i \alpha h(x_i)}. \tag{9}$$

We implement a parallelised variant of AdaBoost. Similar to the proposed Weighted Cascade, our tweak trains t trees at the same time where t is the number of allowed threads. As a result, the weighted classification error is estimated from a batch of tree in each iteration, and the weight α is applied to every tree in the batch. Since we consider multiclass problems, we chose to implement the SAMME variant of AdaBoost (Hastie et al., 2009).

Tree Prediction Aggregation Methods

Given the individual predictions of the trees in an ensemble, there are many ways to aggregate them. The two main methods are hard-voting and soft-voting.

Hard-voting (HV). Hard-voting is a simple form of voting described as below:

$$\hat{y} = mode\{h_i(\boldsymbol{x}), \cdots, h_{|E|}(\boldsymbol{x})\}, \tag{10}$$

where h_i is the ith classifier and E is the ensemble. That is, each classifier votes a class, and the class that obtains the largest number of votes is the final result.

Soft-voting. By default, Silas uses soft-voting as it tends to give better results than hard-voting. Soft-voting is often described as

$$\hat{y} = \arg\max_c \sum_{i=1}^{|E|} \alpha_i p_i^c, \tag{11}$$

where E is the ensemble, α_i is the weight of the ith tree, and p_i^c is the probability that the ith classifier predicts class c. The voted result is the class that maximises the weighted sum of probabilities. For instance, if tree-weights are all 1, a tree votes $(0.8, 0.2)$ and another tree votes $(0.4, 0.6)$, the end result will be $(1.2, 0.8)$, i.e., vote for class 0.

However, we find that normalising the result at each leaf node and obtaining probabilities often lead to worse predictions than directly using the distribution of the classes at each leaf node. So the "soft-voting" in Silas keeps the count for each class at the leaf nodes and aggregates the counts instead, that is,

$$\hat{y} = \arg\max_c \sum_{i=1}^{|E|} \alpha_i D_i^c \tag{12}$$

where D_i^c is the count of instances of class c at the leaf node of the ith classifier. For instance, if tree-weights are all 1, a tree votes $(8, 2)$ and another tree votes $(40, 60)$, the end result will be $(48, 62)$, i.e., vote for class 1, instead of voting for class 0 using Equation 11. This kind of aggregation often performs better, especially when we want to terminate the growing of the tree early. Moreover, our earlier experiment revealed that the AUC would likely be better if the leaf nodes' weights are not normalised before aggregation. Intuitively, this is due to the fact that leave nodes' size may vary, hence a bigger leave node carries more information and votes with greater certainty.

On the other hand, for datasets that contain a large number of classes, we find that soft-voting consumes too much memory and is *not feasible*. For example, if we store the count in a 64-bit integer, then soft-voting requires to store a K-dimensional vector of 64-bit integers, where K is the number of classes, at *every* leaf node of *every* tree. In the case that the depth of each tree is 20 and there are 1000 classes and 100 trees, it would require $2^{20} \times 1000 \times 8(Byte) \times 100 \approx 780GB$ of RAM to store the leaf nodes. In comparison, hard-voting only needs to store the index of the voted class per leaf, which requires 1000 times less memory for storing leaf nodes in the soft-voting case. When we grow each tree until the leaf only contains 1 instance, which is the standard in C4.5, hard-voting and soft-voting should achieve similar results. In the experiment in this paper, all combinations of sub-algorithms use soft-voting except for HV.

4.1.3. Remarks

All the forest-level algorithms can be applied on top of bootstrapping or other sampling methods. For comparison purposes in this paper, we assume that no other data sampling methods are used at forest-level. Under these settings, the combination of UB and GT can be deemed as a variant of the Random Forest (Breiman, 2001) algorithm with more aggressive sampling, whereas the combination of NB and RT is similar to Extremely Randomised Trees (ExtraTrees) (Geurts et al., 2006).

4.2. High-Performance Computing for ML

In this section, we list some of the major implementation and design choices which contribute to the excellent time and memory efficiency of Silas. There are various incentives to develop high-performance machine learning tools that target commodity hardware. The benefits include a smaller ecological footprint as well as cost savings and increase in productivity. Reducing the hardware requirements of machine learning applications also fosters security as the data no longer need to be transmitted through off-site cloud infrastructures and can instead be locally hosted.

We reviewed existing implementations of ensemble tree machine learning for classification and noticed a large difference in performance. It is often assumed that the community and big companies refine open-source implementation over time in such a way that their quality and efficiency is high. Newcomers have often been dissuaded to pursuit their own implementation if not for educational purposes. However, the performance of a piece of software is heavily influenced by the technologies used and the programming paradigm employed.

To our knowledge, at the time of our survey, one of the best if not the best commercial implementation of tree-based machine learning is H2O (Cook, 2016)£¬ which is an open-source java implementation supported by a company of the same name. Another competing implementation is Ranger (Wright and Ziegler, 2017) – a C++ open-source implementation branded as fast and suited for high dimensional data. Both follow an object-oriented programming paradigm. When developing Silas, we settled with the following three key points at the core of Silas code base.

Programming language. The programming language itself must be efficient and low-level enough to give us the liberty to perform cache and instruction-level optimisations. Similarly to Ranger, we chose C++ because it is fast (Heer, 2019) and provides enough high-level programming features, such as template metaprogramming (Abrahams and Gurtovoy, 2004), which are useful when realising the other vital points.

Pure functions in C++. we have developed a novel *C++ functional & parallel programming framework* that enables us to compose pure functions in a straight forward manner statically. This framework is largely inspired by the functional programming paradigm. More specifically, we predominantly employ *pure functions* due to the following beneficial properties (Carmack, 2012): reusability, testability, thread-safety and the absence of side effects. In this framework, we notably employed a static dispatch technique called Curiously Recurring Template Pattern (CRTP) (Abrahams and Gurtovoy, 2004) to offer efficient means of sequential and parallel compositions. In a multi-core execution environment, this organisational framework incurs a low run-time overhead.

Data-oriented paradigm. In contrast with Ranger and H2O, our codebase follows a data-oriented programming paradigm. The emphasis is placed on the data being created, manipulated and stored. The main advantage of data-oriented programs is the constraint on the locality of reference, which enables safe and effective parallelism (e.g. vectorisation of code). Another benefit of data-oriented programming is the efficient use of memory caching, an essential aspect of modern hardware.

Management of big data. We have developed a new *data storage and management library* to deal with datasets that have a large number of rows and columns more efficiently. This library includes features such as *stable vector*, a memory optimisation that stores multi-dimensional data into a flat array with stable referencing during training; *data storage by columns*, which significantly reduces cache-miss when selecting nodes for decision trees; and *high information density*, which reduces cache and memory usage based on the type and values of each feature.

These three overarching design choices, together with rigorous profiling, were key to the high-performance of Silas, as demonstrated by the empiric results of the following section. To foster the development of high-performance machine learning, and as part of our technical contribution to the community, we open-sourced the *C++ functional & parallel programming framework* as well as the data-structures we developed. [2]

5. Experimental Results

This section compares the computational performance and predictive performance of the methods mentioned above on numerous datasets. Section 5.1 uses experiment on medium and large datasets to compare different methods of Silas. Section 5.2 uses experiment on larger datasets to highlight Silas' abilities of high-performance computing. The experiment was conducted on a desktop equipped with an Intel Core i7-7700 quad-core CPU and 32GB RAM running on Ubuntu 19.04. Silas source code is writ-

[2]Our *C++ functional & parallel programming framework* is available via `https://depintel.com/downloads/public/code-release-01.zip`.

ten in C++. Extra libraries such as Intel Thread Building Blocks (TBB) are used for efficient parallel computation. GPU features have been disabled.

5.1. Results on Medium and Large Datasets

Dataset	**Size**	**# Classes**	**# Num. Feat.**	**# Nom. Feat.**	**Class Ratio**
Binary Classification					
diabetes	768	2	8	0	1.87:1
jm1	10,885	2	21	0	4.17:1
mozilla4	15,545	2	5	0	2.04:1
adult	48,842	2	2	12	3.18:1
kick	72,983	2	14	18	7.13:1
creditcard	284,807	2	30	0	577.88:1
flight	1,000,000	2	2	6	4.13:1
Multi-class Classification					
connect-4	67,557	3	0	42	6.90:2.58:1
fashion-mnist	70,000	10	784	0	1.00:…:1
mnist-784	70,000	10	784	0	1.24:…:1
walking-activity	149,332	22	4	0	24.14:…:1
cover-type	581,012	7	54	0	103.13:…:1
led5000	1,000,000	10	0	24	1.01:…:1

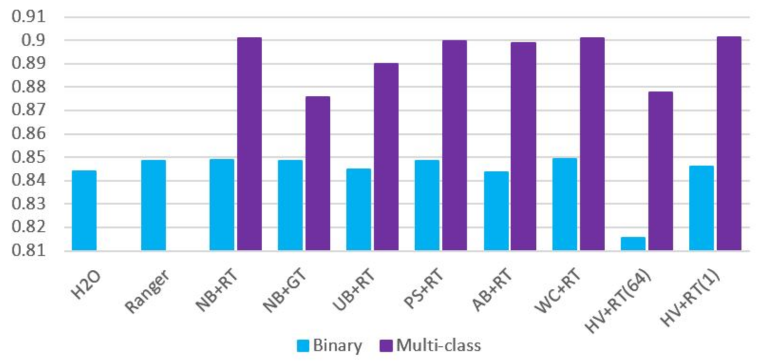
Table 2: Selected datasets.

We use the datasets in Table 2 as a benchmark for the remainder of this paper. Except for the flight dataset (Pafka, 2019), all the other datasets can be found on OpenML (Vanschoren et al., 2013). These datasets are selected on the following basis: they are from real-life problems; they have a large number of instances (except diabetes); they are a mixture of binary classification and multi-class classification problems; the ratios between classes range from balanced to imbalanced; they involve numerical features (Num. Feat.) and nominal features (Nom. Feat.). Note that when there are too many classes, we only show the ratio of the largest class and the smallest class. For example, the ratio between the largest class and the smallest class in cover-type is 103.13:1.

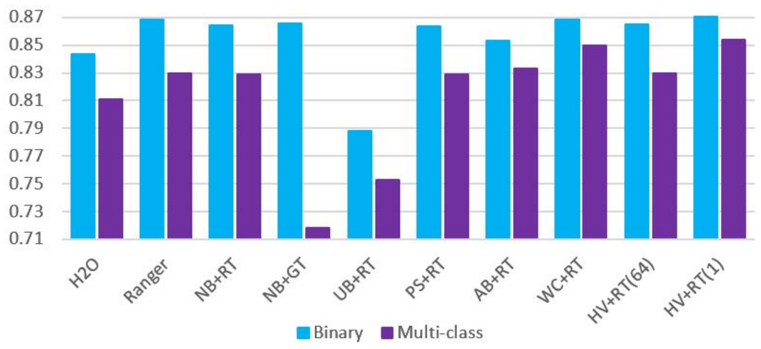
The experiment is run 10 times, and we present the average accuracy, AUC, and training time that includes time for loading data. The 95% confidence interval is usually below 0.001 for accuracy and AUC, so we do not show them in the tables. In each run, all datasets except flight are tested using 10-fold cross-validation, and the results are the average of the validations. The flight dataset is tested using a separated dataset. Different tools have different default hyper-parameters for tree depth and leaf size (number of instances at each leaf node). To ensure that the experiment and comparison are fair, we use the following fixed hyper-parameters: 100 trees, 64 max tree depth, 64 min leaf size, and default settings otherwise, across all tested tools. We present extracts of the full table below and discuss what they demonstrate. We give the full table in Appendix A. The reader can refer to Appendix B for experiment results using default settings of H2O and Ranger. We highlight the "best results", which are defined as the highest values when rounding to the third decimal place, across all the tools and methods with bold font.

Overall results. Figure 1 shows the overall results of different Silas methods in comparison with H2O and Ranger. Figure 1 (a) reveals that Silas ($NB+RT$) and Silas ($WC+RT$) give the highest AUC for binary problems, they are slightly higher than Ranger and noticeably higher than H2O. H2O and Ranger do not report multi-class AUC. Figure 1 (b) shows that Silas ($HV+RT$ (1), i.e., with leaf size 1) and Silas ($WC+RT$) give comparable accuracies than Ranger and better than H2O for binary problems. On the other hand, these two Silas methods yield significantly better accuracies than H2O and Ranger for multi-class problems. Figure 1 (c) shows that most Silas methods are significantly faster than both H2O and Ranger for both binary and multi-class datasets. An exception is Silas ($PS+RT$), which is slow for multi-class problems. H2O is generally very slow for multi-class problems (1587s, off the chart). An outlier is Silas ($HV+RT$ (1)), which is not that slow for most datasets but spend 38,143s on the led5000 dataset. See Table 7 for details. Overall, Silas ($WC+RT$) gives better AUC and accuracy than H2O and Ranger on both binary and multi-class problems and is much faster than H2O and Ranger. Other Silas methods have various trade-offs, which we will elaborate below.

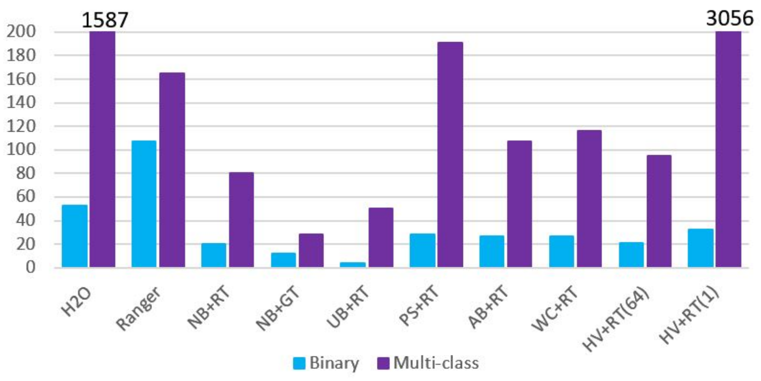
Comparing Silas with H2O and Ranger. We give the results for H2O and Ranger as a baseline in Table 3 and results for Silas in Tables 4, 5, 6 and 7. As both H2O and



(a) Average AUC on binary and multi-class datasets.



(b) Average prediction accuracy on binary and multi-class datasets.



(c) Average training time (in seconds) on binary and multi-class datasets.

Figure 1: A comparison of AUC, accuracy and and training time of considered tools/methods.

	H2O			Ranger		
Dataset	**AUC**	**Acc.**	**Time (s)**	**AUC**	**Acc.**	**Time (s)**
Binary Classification						
diabetes	0.8155	0.7550	4	**0.8390**	**0.7637**	< 1
jm1	0.7319	0.6997	15	**0.7550**	0.8169	2
mozilla4	0.9655	0.9346	14	**0.9790**	0.9459	2
adult	0.9147	0.8492	31	**0.9185**	**0.8656**	11
kick	0.7609	0.8669	95	0.7658	**0.9011**	39
creditcard	0.9760	**0.9993**	161	0.9602	**0.9994**	557
flight	0.7442	0.7996	53	0.7225	0.7838	139
Average (binary)	0.8441	0.8434	53	0.8486	0.8681	107
Multi-class Classification						
connect-4	-	0.7269	174	-	0.7700	19
fashion-mnist	-	0.8567	3,266	-	0.8754	176
mnist-784	-	0.9392	3,026	-	0.9580	143
walking-activity	-	0.6174	948	-	0.6488	116
cover-type	-	0.8763	4,444	-	0.8315	343
led5000	-	0.6236	8,395	-	0.6252	603
Average (binary + multi-class)	-	0.8111	1,587	-	0.8296	165

Table 3: Results from H2O and Ranger.

Ranger do not report multi-class AUCs, we leave multi-class AUCs in Table 3 blank. The results reveal that most Silas methods are much faster than H2O and Ranger. For instance, with similar predictive ability, Silas' $NB+RT$ (i.e., 80 seconds in Table 4) is 19x faster than H2O (i.e., 1,587 seconds in Table 3) and 2x faster than Ranger (i.e., 165 seconds in Table 3) in terms of average training time on all datasets. On predictive performance, Ranger gives the best AUC for four binary-class datasets, but it gives poor AUC results on the other three binary-class datasets. On average, Silas' $NB+RT$ has a slightly better AUC (i.e., 0.8489 in Table 4) than Ranger (i.e., 0.8486 in Table 3)) on binary-class datasets. Considering both all datasets, both $AB+RT$ (i.e., 0.8331 in Table 6) and $WC+RT$ (i.e., 0.8498 in Table 6) have better average accuracies than both H2O (i.e., 0.8111 in Table 3) and Ranger (i.e., 0.8296 in Table 3).

Next we give some comparisons of algorithms within the Silas framework.

	Silas ($NB+RT$)			Silas ($NB+GT$)		
Dataset	**AUC**	**Acc.**	**Time (s)**	**AUC**	**Acc.**	**Time (s)**
Binary Classification						
diabetes	0.8297	0.7520	< 1	0.8087	0.7488	< 1
jm1	0.7490	0.8149	1	0.7527	0.8146	3
mozilla4	0.9724	0.9363	1	0.9679	0.9286	3
adult	0.9079	0.8552	6	0.9174	0.8646	10
kick	**0.7680**	0.8998	13	**0.7682**	**0.9008**	24
creditcard	0.9777	**0.9992**	91	0.9635	**0.9989**	12
flight	0.7377	0.7942	24	**0.7615**	**0.8037**	34
Average (binary)	0.8489	0.8645	20	0.8486	0.8657	12
Multi-class Classification						
connect-4	0.8927	0.7697	12	0.8917	0.7843	13
fashion-mnist	**0.9881**	0.8582	85	0.9351	0.4972	14
mnist-784	0.9975	0.9465	72	0.9762	0.7562	27
walking-activity	**0.9676**	0.6363	57	0.8446	0.1853	7
cover-type	0.9901	0.8841	273	0.8925	0.4883	26
led5000	0.9320	0.6276	396	0.9044	0.5688	192
Average (binary + multi-class)	0.9008	0.8288	80	0.8757	0.7185	28

Table 4: A comparison between $NB+RT$ and $NB+GT$.

RT and GT. Table 4 shows the results on $NB+RT$ and $NB+GT$. Although the runtime differs case by case, overall, it is consistent that GT is faster than RT. RT does not involve the computation to find the best cut-point, but it often leads to deeper trees, whereas GT is able to yield shorter trees. GT has obtained three good accuracies for binary classification problems, i.e., 0.9008 on kick, 0.9989 on creditcard and 0.8037 on flight, but GT performs poorly for multi-class problems. GT is not suited for datasets such as covertype and walking-activity because they have a large number of imbalanced classes. For example, the class ratio of covertype is 103.13 : … : 1, which means that the largest class has approximately 100 times more instances than the smallest class. In such cases, the number of instances for minor classes tends to be very small. As GT grows leaf nodes by finding the best cut-points where the size of samples is determined by the prevalence of minor classes, the number of instances in each leaf node tends to be

very small, resulting in poor accuracies for multi-class problems. On the other hand, *RT* performs well for multi-class problems because its number of instances in each leaf node is not restricted by the prevalence of minor classes, which means that each leaf node can gain more instances to acquire more reasonable features to distinguish different classes. For such reasons, *RT* can more generally deal with imbalanced data, while *GT* is suitable for datasets with a sufficient number of instances in minor classes.

	Silas ($UB+RT$)			Silas ($PS+RT$)		
Dataset	**AUC**	**Acc.**	**Time (s)**	**AUC**	**Acc.**	**Time (s)**
Binary Classification						
diabetes	0.8298	0.7498	< 1	0.8297	0.7584	< 1
jm1	0.7344	0.6801	1	0.7456	0.8110	2
mozilla4	0.9694	0.9297	1	0.9710	0.9329	2
adult	0.9054	0.7937	4	0.9072	0.8490	11
kick	0.7633	0.7341	4	0.7675	0.8976	26
creditcard	0.9799	0.9878	5	**0.9814**	**0.9993**	127
flight	0.7307	0.6411	12	0.7365	0.7983	28
Average (binary)	0.8447	0.7880	4	0.8484	0.8638	28
Multi-class Classification						
connect-4	0.8560	0.6850	5	0.8861	0.7847	28
fashion-mnist	**0.9881**	0.8579	84	**0.9881**	0.8579	545
mnist-784	0.9974	0.9459	66	0.9975	0.9466	522
walking-activity	0.9419	0.5131	21	0.9674	0.6360	76
cover-type	0.9397	0.6387	48	0.9890	0.8798	491
led5000	0.9319	0.6275	398	0.9312	0.6261	623
Average (binary + multi-class)	0.8898	0.7526	50	0.8999	0.8290	191

Table 5: A comparison between $UB+RT$ and $PS+RT$.

UB and PS. Table 5 shows results for *UB* and *PS*. *UB* is very fast due to its aggressive sampling. It can obtain good AUC, but it often yields bad accuracy. Consequently, the combination of $UB+GT$ gives the worst accuracy of all tested methods, although it gives very good AUC on some binary classification problems (e.g., flight dataset). When undersampling the majority class(es), *PS* (with *RT*) yields both high AUC and high accuracy. This indicates that the proposed sampling improves accuracy compared to uniform balancing. However, even the few distance computations make the computation much slower.

	Silas ($AB+RT$)			Silas ($WC+RT$)		
Dataset	**AUC**	**Acc.**	**Time (s)**	**AUC**	**Acc.**	**Time (s)**
Binary Classification						
diabetes	0.8129	0.7486	< 1	0.8162	0.7551	< 1
jm1	0.7433	0.7964	2	0.7498	**0.8184**	3
mozilla4	0.9776	0.9338	2	**0.9789**	**0.9486**	2
adult	0.8922	0.8259	9	0.9109	0.8571	11
kick	0.7625	0.8837	18	0.7654	**0.9009**	21
creditcard	0.9747	**0.9992**	121	0.9771	**0.9994**	111
flight	0.7419	0.7833	37	0.7458	0.8013	41
Average (binary)	0.8436	0.8530	27	**0.8492**	**0.8687**	27
Multi-class Classification						
connect-4	0.8994	0.8365	18	0.8988	**0.8489**	20
fashion-mnist	**0.9879**	0.8687	128	**0.9880**	**0.8842**	133
mnist-784	0.9983	0.9624	107	**0.9986**	**0.9659**	113
walking-activity	0.9667	0.6348	63	0.9643	**0.6663**	71
cover-type	0.9926	0.9253	355	**0.9961**	**0.9544**	398
led5000	**0.9329**	0.6311	531	0.9241	**0.6474**	582
Average (binary + multi-class)	0.8987	0.8331	107	0.9011	0.8498	116

Table 6: A comparison between $AB+RT$ and $WC+RT$.

WC and AB. Table 6 compares WC and AB. Changing the weight of data instances is shown to improve the accuracy of both binary and multi-class problems. $AB+RT$ has the second-best accuracy in the tested Silas methods. Although $WC+RT$ gives sub-par AUC for diabetes, jm1 and flight, it still has the best average AUC and accuracy for binary problems. It also gives the best accuracy for every tested multi-class dataset. However, both WC and AB do not yield good AUC when combined with GT.

Space to improve. We note that all the tested methods, including H2O and Ranger, may give better results under other parameters. For example, we find that the parameter “min leaf size” largely affects the predictive performance, and every method performs well on a different value for each dataset. There are also a wide range of other parameters for each implementation that may affect the prediction results. For example, boosting is often used to grow shallow trees in the literature, but our implementation of AdaBoost performs better when tree depth is large. It is, therefore, infeasible and out of the scope of this paper to find the best parameters for each method on each dataset. We chose leaf size 64 because it generally yields high AUC for tested methods.

HV with different leaf sizes. The sub-algorithm HV often performs well with "min leaf size" set to 1. Thus, we show its results separately in Table 7. Note that the "best results" in Table 7 are better, but do not follow the same rule as previous tables because the parameters are different. The combination of $HV + RT$ with leaf size 1, although gives sub-par AUC on some binary classification problems, often gives even better accuracy than $WC + RT$. Again, $WC + RT$ may yield better results on the parameters we have not tested.

Comparison with recent research results. It is challenging to compare with other methods without directly running them on the same hardware, but we can still have some slightly indirect comparisons of predictive performance with some recent research papers that propose new machine learning algorithms (rather than the ones that aim to solve a specific classification problem, in which case the authors would have done extensive parameter tuning to optimise the results, which we have not).

The Weighted Oblique Decision Trees (AAAI 2019) (Yang et al., 2019) obtained 0.7161 accuracy on diabetes, 0.9434 on mnist, and 0.7415 on connect-4 *in 5-fold cross-validations*, which are well below our results. The vanilla Hoeffding Tree obtained an accuracy of 0.8391 on the adult dataset (IJCAI 2019) (Zhang and Ntoutsi, 2019), and the Fairness-aware Hoeffding Tree, which is focused on fairness than accuracy, obtained an accuracy of 0.8183. The Very Fast Decision Tree (ICDM 2019) (Losing et al., 2018) obtained an accuracy of 0.6973 for the cover-type dataset. The optimal method of Distinct Decision Trees (ICML 2017) (Ruggieri, 2017) obtained 0.8569 accuracy on the adult dataset in a 10-fold cross validation. Even without extensive hyperparameter tuning, Silas ($WC + RT$, cf. Table 6) outperformed all the above, and

	Silas ($HV+RT$) - Leaf Size 64			Silas ($HV+RT$) - Leaf Size 1		
Dataset	**AUC**	**Acc.**	**Time (s)**	**AUC**	**Acc.**	**Time (s)**
Binary Classification						
diabetes	0.8196	0.7505	< 1	0.8185	**0.7657**	< 1
jm1	0.7151	0.8153	2	0.7588	**0.8205**	5
mozilla4	0.9652	0.9400	1	**0.9819**	**0.9556**	2
adult	0.8806	0.8551	8	0.8957	0.8503	24
kick	0.6872	0.9000	15	0.7559	**0.9014**	37
creditcard	0.9303	0.9994	93	0.9536	**0.9995**	92
flight	0.7106	0.7937	29	0.7588	0.8013	67
Average (binary)	0.8155	0.8649	21	0.8462	**0.8706**	32
Multi-class Classification						
connect-4	0.8638	0.7760	17	0.9130	0.8326	100
fashion-mnist	0.9866	0.8620	86	0.9908	0.8828	121
mnist-784	0.9975	0.9513	73	0.9991	**0.9723**	104
walking-activity	0.9484	0.6354	53	0.9687	**0.6697**	530
cover-type	0.9850	0.8825	330	0.9979	**0.9556**	506
led5000	0.9213	0.6251	534	0.9253	**0.6904**	38,143
Average (binary + multi-class)	0.8778	0.8297	95	0.9014	0.8537	3,056

Table 7: Experimental results for medium to large datasets on Silas with $HV+RT$.

Silas ($HV+RT$, cf. Table 7) with leaf size 1 outperformed all the above except the 0.8569 accuracy on adult.

5.2. Results on Larger Datasets

In this section we focus on large datasets as they highlight the importance of code optimisation.

Large number of instances. The 10 million flights dataset (Pafka, 2019) is a good binary classification problem with a mixture of numerical features and nominal features. The characteristic of the 10 million flights dataset is the same as the 1 million version in Table 2, except that it has 10 million instances. We compare $UB+GT$, $NB+GT$, $WC+GT$, which are good performers for binary classification problems, with H2O and Ranger on this dataset with the default settings of 100 trees, 64 tree depth and 64

Dataset - Flight 10M	**AUC**	**Acc.**	**Time (s)**	**Mem. (GB)**
Silas (UB+GT)	0.7924	0.6976	184	4.3
Silas (NB+GT)	0.8033	0.8134	453	10
Silas (WC+GT)	0.8045	0.8082	699	9.7
H2O	0.7737	0.7250	1,198	12.6
Ranger	0.7403	0.7838	2,047	29.4

Table 8: Experiment results for the 10 million flight dataset. Time includes loading data, training, and computing the predictive performance. "Mem." shows the *peak memory usage* during the computation.

min leaf size. The results are shown in Table 8. The $UB+GT$ method has a very small computational footprint, and it gives competitive AUC. The best predictive results are from $NB+GT$ and $WC+GT$. H2O and Ranger do not give as good results, and they take a rather long time.

The 10-million flights dataset was also used in the LightGBM paper (NIPS 2017) (Ke et al., 2017), but their best AUC was below 0.79. The best AUC from the benchmarked tools in (Pafka, 2019) is 0.812 from H2O with 5000 trees in 9.5 hours on a 32-core CPU and 250GB RAM machine. In comparison, we are able to obtain **0.8147** AUC with $UB+GT$, feature proportion 1.0 (use all features when finding the best split), and 500 trees in 28.5 minutes using a 4-core CPU and less than 10GB RAM.

Large number of features and classes. The ASR-200 dataset includes 79,157 instances and 1,332 classes of phoneme recognition, which is a subprocess of automatic speech recognition (ASR). The instances are generated using the Kaldi ASR toolkit (Povey et al., 2011) and 200 sentences in the Librispeech ASR corpus (Panayotov et al., 2015). For each instance, the input is a 143-dimensional feature vector, which consists of 13-dimensional Mel-Frequency Cepstral Coefficients (MFCCs) of 11 consecutive voice frames, and the label is an ID (called a senone) that corresponds to the phoneme of the voice frames. The labels are obtained by running Kaldi's default Librispeech acoustic model training script and the force alignment algorithm on the trained tri4b GMM-HMM (i.e., Gaussian Mixture Models and Hidden Markov Models) acoustic model. We use 160 sentences to generate a training set and the remaining 40 sentences to

Dataset - ASR-200	**# Trees**	**Acc.**	**Time (s)**	**Mem. (GB)**
Silas (NB+RT)	100	0.5121	161	23
Silas (WC+RT)	100	0.5149	168	22.6
Silas (HV+RT)	100	0.5873	160	2.3
Silas (HV+RT)	1,000	0.6253	1,558	6.7
Ranger	100	0.5239	29,782	3.0
Scikit-learn	100	0.5253	2,152	29.5 + 27.4 (swap)

Table 9: Experiment results for the ASR-200 dataset. Time and memory usage is measured the same way as in Table 8.

generate a test set. The training set contains 63,197 instances, and the test set contains 15,960 instances. Overall, each class has at most 380 instances. We give the results in Table 9. We do not report the AUC because calculating multi-class AUC for the 1,332 classes is too time-consuming. H2O could not load the dataset because there are too many classes, so we test Scikit-learn (Pedregosa et al., 2011) instead. We use max depth 64, leaf size 1 and feature proportion 0.4 because these settings perform well for this dataset using our methods. Results from default settings of Ranger and Scikit-learn are reported in Appendix C, where they obtained 0.54 accuracy. Even NB and WC use too much memory when dealing with a large number of classes. As a result, we are only able to train more trees with HV on our machine within 10 hours. $HV+RT$ obtained significantly better results than others in both Table 9 and Appendix C.

5.3. *Summary of Experimental Results*

We propose numerous customisations to the bagging and the AdaBoost algorithm and test various combinations of the proposed sub-algorithms in the Silas framework on a range of medium to large datasets. Experimental results show that our methods, especially $NB+RT$, $WC+RT$ and $HV+RT$, often outperform state-of-the-art tools and recent research results. This shows that the customisations in different parts of the ensemble trees method add up, and they together contribute to noticeable advantages over the existing methods. We summarise the characteristic of the proposed methods in Table 10, which acts as a guide for using the Silas machine learning framework.

Method	Pro	Con
$UB+GT$	Fast, memory-efficient, good AUC for binary classification.	Poor accuracy.
$PS+RT$	Produces a subsample of data that yields good AUC and accuracy.	Slightly slower than other Silas methods.
$NB+RT$	Well-rounded, excellent AUC and accuracy for most problems.	Slightly larger memory footprint than other Silas methods.
$WC+RT$	High accuracy for multi-class problems.	Sub-par AUC on *some* binary-class datasets.
$HV+RT$	High accuracy for multi-class problems, memory-efficient when #classes is large.	Generally poor AUC for binary classification, very slow on some datasets (e.g., led5000).

Table 10: Summary of the Silas methods in this paper.

In the experiment, different datasets have been used to evaluate Silas and led to varying sizes of decision trees. In a number of cases, decision trees can be huge. For example, the 1M flight dataset can result in decision trees that usually contain 30K leaves. Although decision trees are white-box models that can be directly observed and explained by humans, it is extremely difficult for humans to observe and explain very large decision trees and large forests of large decision trees, e.g., 100 trees with 30K leaves on each tree. This motivated us to use automated reasoning techniques to enable computers to discover explanations for decision trees automatically. In the next section, we will describe the explanation module in Silas.

6. Applications of Logical Reasoing in Machine Learning

This section concerns eXplainable AI and safe machine learning. We describe our solution towards a more trustworthy machine learning technique using logic and automated reasoning as the backbone. We discuss the Model Audit module for formally verifying prediction models against user specifications, the Enforcement Learning module for training correct-by-construction models, Model Insight for explaining prediction models and Prediction Insight for explaining prediction instances. The discussion in this section is focused on *binary classification*, which is closer to most logical reasoning tasks than multi-class classification, though a multi-class problem can also be handled by the methods in this section through encodings such as one-vs-one and one-vs-all. We consider logical reasoning for general multi-class problems as future work.

6.1. Logical Semantics of Decision Trees

Given a decision tree, we can obtain the following types of logical formulae:

Internal Node formula: a logical formula corresponding to every internal node.

Branch formula: a logical formula $(\bigwedge N) \rightarrow (y = M(d_l))$, where N is the set of internal node formulae along the branch leading to the leaf l and d_l is the distribution associated with l.

Tree formula: a logical formula $\bigvee B$, where B is the set of branch formulae.

Each of the formula mentioned above is associated with a weight. An internal node formula is weighted by the information gain computed during training. A branch formula leading to a leaf l is weighted by the value $log_2(2) - H(l)$ where $H(l)$ is the entropy (Shannon, 1948) of the leaf l. A tree formula is weighted by the ROC-AUC score obtained on its out-of-bag sample during training.

6.2. Model Audit

The purpose of the model audit module is to provide the means to certify that the prediction model complies with user specifications formally. To do so, we adopt advanced automated reasoning techniques, especially satisfiability modulo theories (SMT) solvers (De Moura and Bjørner, 2011). SMT solvers determine the *satisfiability* of logical formulae with respect to combinations of background theories expressed in classical first-order logic with equality. A logical formula f is said satisfiable, denoted by $SAT(f)$, if and only if there exists a valuation assigning values to its variables such that it is evaluated to true.

A *user specification* is a tuple $S = \langle s_\perp, s_\top \rangle$ where $s_\perp$ and $s_\top$ are logical formula over $\{x_1, ..., x_n\}$. In real-life applications, there are often some mandatory specifications, which we refer to as "hard specs", and some that are optional, which we refer to as "soft specs". The user can use the Model Audit feature to check both types of specs and rate the model based on the verification results. For "hard specs", we propose Enforcement Learning (cf. Section 6.3) to train models that are guaranteed correct.

A prediction model complies with the user specification S if for all input instance $x \in X$ leading to a *positive* (resp. *negative*) prediction, $s_\top(x)$ (resp. $s_\perp(x)$) evaluates to true. Formally, a user specification S is valid over a prediction model $G : X \to D(Y)$, denoted by $G \models S$, if and only if:

$$\forall\, x \in X, ((M(G(x)) = \mathit{negative}) \to s_\perp(x)) \wedge ((M(G(x)) = \mathit{positive}) \to s_\top(x)). \quad (13)$$

To verify the validity of a user specification S over an ensemble trees model E_T, i.e. $E_T \models S$, we propose to reduce the problem to the verification of the validity of S over each of the decision trees in T.

Theorem 6.1 (Soundness)**.** *If S is valid over all tree in T, i.e. $\forall\, t_i \in T, t_i \models S$, then S is valid over E_T, i.e. $E_T \models S$.*

Proof. (Outline) Without loss of generality, let us consider an ensemble trees model E_T based on two trees, i.e. $T = \{\langle w_1, t_1\rangle, \langle w_2, t_2\rangle\}$. Now assume that $\forall\, t_i \in T, t_i \models S$. Given an arbitrary $x \in X$, we have to consider two case: (i) $M(t_1(x)) = M(t_2(x))$ and (ii) $M(t_1(x)) \neq M(t_2(x))$. **Case i:** Let $y \in Y$ such that $M(t_1(x)) = M(t_2(x)) = y$ then, by definition of E_T, we have $M(E_T(x)) = y$ and we can conclude that $((M(E_T(x)) = \mathit{negative}) \to s_\perp(x)) \wedge ((M(E_T(x)) = \mathit{positive}) \to s_\top(x))$ holds. **Case ii:** Without loss of generality, let us consider the case where $M(t_1(x)) = \mathit{positive}$ and $M(t_2(x)) = \mathit{negative}$. Since we assumed that $t_1 \models S$ and $M(t_1(x)) = \mathit{positive}$, we know that $s_\top(x)$ holds. Similarly, since we assumed that $t_2 \models S$ and $M(t_2(x)) = \mathit{negative}$, we know that $s_\perp(x)$ holds. We can conclude that $s_\top(x) \wedge s_\perp(x)$ holds, hence $((M(E_T(x)) = \mathit{negative}) \to s_\perp(x)) \wedge ((M(E_T(x)) = \mathit{positive}) \to s_\top(x))$ holds. □

We note that this reduction is not sound when considering multiclass classification, where the number of classes is greater than two. Further, this reduction is not complete since a tree could violate the specification while being outnumbered by trees that comply with the specification in the aggregation phase of the ensemble tree model. This is a trade-off purposefully made to reduce the overall complexity in order to achieve better scalability because the completeness of verification renders the computation unscalable (Törnblom and Nadjm-Tehrani, 2019).

We now proceed to show that, using the reduction we described, SMT solver can be efficiently applied to the verification of user specification over ensemble tree models. Let t be a tree and F_t its corresponding tree formula, the user specification S is valid over t if and only if:

$$\neg SAT(F_t \wedge \neg(((y = negative) \rightarrow s_{\perp}(x)) \wedge ((y = positive) \rightarrow s_{\top}(x)))) \qquad (14)$$

By Theorem 6.1 this means that we can use SMT solvers to verify the validity of a user specification over ensemble trees models. This can be done in a parallel fashion since each tree of an ensemble trees model can be verified independently.

The Model Audit feature employs the Z3 solver (de Moura and Bjørner, 2008). The interaction with Z3 is straightforward as Silas supports direct translation from logical formulae to the `z3::expr` type in Z3 C++ binding.

The remainder of this section presents a case study and report experimental results demonstrating the feasibility and efficiency of the proposed approach.

Case study. We use the Kick dataset as a real-life application to illustrate the Model Audit feature. The goal of this dataset is to predict whether a used car at an auction is a good buy or a bad buy. In a hypothetical scenario where carmaker B discovered that model C produced in year YY have problems and they had recalled all those cars. We wish to check if our prediction model already "knows" this. We formulate the specification as follows: $(y = positive) \rightarrow \neg(make = B \wedge model = C \wedge year = YY)$. The Model Audit feature can be used to check if the prediction model meets the specification. In case it does not, we can use Enforcement Learning described in the following section to train a new model that builds in this information. Running Model Audit on the new model again shows that it meets the specification (guaranteed).

To evaluate the efficiency of the verification procedure, we generate models of various sizes (in terms of the number of trees and leaf size) for the Kick dataset and record the computation time of the Model Audit feature when verifying the above property.

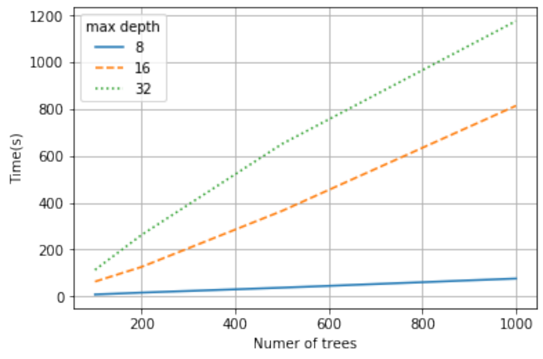


Figure 2: Experiment results of Model Audit.

Experimental results are given in Figure 2. We observe that, as expected, the computation time grows linearly with respect to the number of trees and exponentially with respect to the depth of trees. Full trees in this example are often smaller than depth 32, so the increase from depth 16 to 32 is not large. The verification time for models with positive results and negative results are almost identical. Overall, the verification can be done in a reasonable time ($<$ 20 min) for models with 1000 trees of depth 32.

6.3. Enforcement Learning

The purpose of the *Enforcement Learning* module is to provide the means to build prediction models that, given a user specification, are correct-by-construction. This feature is notably used in the context of critical or regulated applications. It can also be used to enforce additional knowledge given by domain experts.

Given a user specification $S = \langle s_\bot, s_\top \rangle$, Enforcement Learning proceeds as follows: (1) It filters out from the dataset all instance $\langle x, y \rangle$ where:

$$((y = negative) \wedge \neg(s_\bot)) \vee ((y = positive) \wedge \neg(s_\top)) \tag{15}$$

(2) It constructs trees of the form given by figure 3 where t is a tree grown from the filtered dataset.

Trees built according to the above procedure are, by construction, valid with respect to the given user specification. By theorem 6.1 the resulting ensemble tree models are also valid with respect to the given user specification.

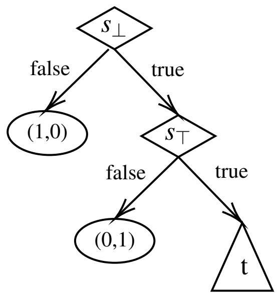


Figure 3: Template of correct-by-construction trees

6.4. Model Insight

When the user obtains a model with satisfactory performance, we provide a feature named Model Insight for analysing the general decision-making of the model.

Given a label v (e.g. *positive*), we are interested in knowing which set of input values would be predicted as v by an ensemble tree model E_T. This way, domain experts may use their knowledge to confirm or refute the rationale exhibited by E_T when predicting label v.

To achieve this, we consider the set B of branches in trees of E_T that predict v. This set corresponds to the set F_B of weighted branch formulae. We can then apply automated reasoning techniques to extract the maximum satisfiable subset corresponding to the set of input values on which the majority of branches in B agree, and this subset is called the maximum satisfiable core (MSC) (Liffiton and Sakallah, 2009). Generally speaking, for a logical formula ϕ in the conjunctive normal form, e.g., $\phi \equiv D_1 \wedge \ldots \wedge D_N$, where N is the number of clauses and each D_i $(i = 1, \ldots, N)$ is a disjunctive clause, a MSC of ϕ can be defined as a subset $S_{MSC} \subseteq \{D_1, \ldots, D_N\}$ such that all clauses in S_{MSC} are satisfiable and the cardinality of S_{MSC} is maximum. In our application, we additionally associate with each sub-formula D_i a weight, which is computed from the information gain of the corresponding node. We then try to find a satisfiable subset S_{MSC} that maximises the total weight. The resultant subset thus represents the

"most informative explanations that are consistent".

This variant of the problem is called Max-Sat, which can be solved using SMT solvers such as Z3 (de Moura and Bjørner, 2008). However, when considering all formulae in F_B, the resulting MSC relates to a very small set of input values for which the model predicts v with very high confidence. Such an MSC corresponds to very specific cases that are not useful in general explanations of the prediction model. Therefore, to broaden the scope of the explanation, we sample at three different levels: the node level, the branch level and the tree level. The sampling results in MSCs that correspond to more general explanations.

The explanations based on MSC extraction can be combined with feature importance to illustrate the decision-making of the prediction model better. There are several methods to compute feature importance in the literature, e.g., (Altmann et al., 2010). Our computation and presentation of feature importance are based on the change in entropy, which is similar to the LIME tool (Ribeiro et al., 2016) and the SHAP method (Lundberg and Lee, 2017). In turn, these methods improve upon earlier work, such as the ranking of the importance of predictors (Breiman et al., 1984). However, feature importance is only one of many components in our explanation, which also includes other aspects such as logical explanation and visualisations in pie charts and bar charts.

Case study. Consider the diabetes dataset (Dua and Graff, 2019). The eight features are the number of times pregnant (preg), plasma glucose concentration (plas), diastolic blood pressure (pres), 2-hour serum insulin (insu), triceps skinfold thickness (skin), body mass index (mass), diabetes pedigree function (pedi) and age. We build a forest of 100 trees with the default settings of Silas and perform the Model Insight analysis on the best model in 10-fold cross-validation. Figure 4 shows the feature importance score of the model. The values are normalised into percentages; thus, we can read the figure as “the feature age contributes 26.55% of the decision making of the model”. Table 11 gives the general decision logic of the same model derived from our MSC extraction method. The pedi and insu features are less important, and we do not show them in Table 11. The decision logic is divided into the constraints that lead to positive diabetes and those that lead to negative diabetes. Medical practitioners can cross-reference the pie chart and the table to evaluate whether the logic of the model is consistent with their knowledge. Disclaimer: the diabetes dataset only contains 768 instances, and their characteristics may not be representative for a larger population.

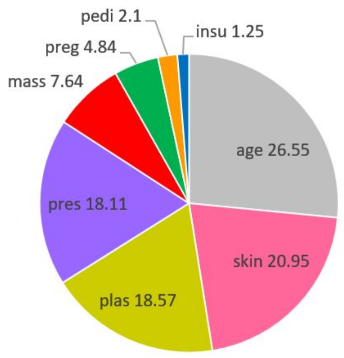


Figure 4: Model Insight: feature importance.

Decision Logic			
Positive	$30 \le$ age < 47	**Negative**	$21 \le$ age < 27
	$31 \le$ skin < 99		$0 \le$ skin < 31
	$155 \le$ plas < 157		$103 \le$ plas < 120
	$40 \le$ pres < 122		$0 \le$ pres < 68
	$30 \le$ mass < 40.8		$0 \le$ mass < 29.8

Table 11: Model Insight: decision logic.

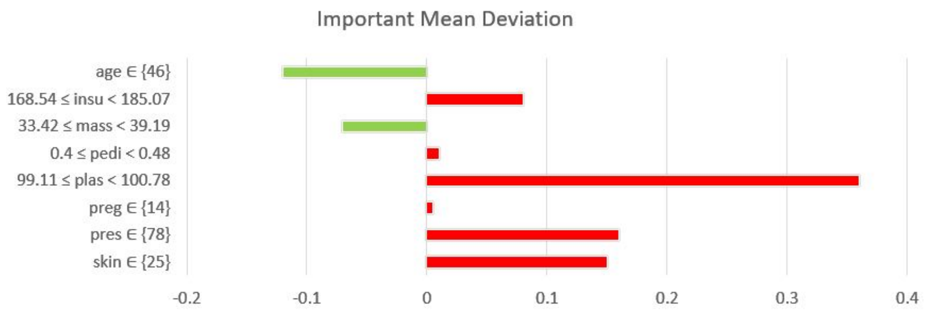


Figure 5: Prediction Insight examples.

6.5. Prediction Insight

The Prediction Insight feature aims at providing users with the decision logic corresponding to individual predictions. This aspect is often an essential element in critical or regulated predictive applications.

Any decision tree, and by extension any ensemble tree model, can be seen as a simple decision rules system composed of rules of the form:

$$\texttt{if}\ \textit{condition}\ \texttt{then}\ \textit{prediction}. \tag{16}$$

Consider an instance $x \in X$ and the decision tree t, the condition of the decision rule associated with the prediction $t(x)$ is the branch formula that leads to the prediction $t(x)$. Likewise, consider the ensemble tree E_T, the condition of the decision rule associated with the prediction $E_t(x)$ is the conjunction of all the branch formula that lead to the predictions $t_1(x), ..., t_m(x)$.

Similar to model insights, prediction insights can be mixed with feature importance scores of individual predictions obtained from feature attribution methods such as SHAP (Lundberg and Lee, 2017) as illustrated by the following case study.

Case study. Figure 5 illustrates a typical prediction insight's output on an instance from the diabetes dataset. The model predicts that there is 62.96% chance that the patient has diabetes. The figure shows how each feature contributes to the prediction and the range at which it does so.

7. Conclusion and Future Work

This work introduced a new machine learning tool called Silas: an ensemble trees learning framework with a formal foundation and customised algorithms. We give empirical evidence that Silas has state-of-the-art predictive performance and is often faster and more memory-efficient than others. Our framework enables the application of logical reasoning and formal verification in machine learning. We demonstrated the "white-box" potential of our approach through a number of proof-of-concept features: Model Audit, which formally verifies user specifications against predictive models;

Enforcement Learning, which generates predictive models that are guaranteed to satisfy user specifications; Model Insight, which provides explanations on how the model works; and a special case of the above called Prediction Insight, which explains how a particular prediction is made.

In the future, we plan to develop the above concepts into mature applications further. We notably intend to pursue the following research directions:

- *Model Audit:* develop sound and complete verification method that adopts formulae simplification, model reduction and model checking techniques.
- *Model Insight:* develop automated insight ranking, selection and visualisation techniques, evaluate the insight through theoretically founded metrics.
- *Other ML tasks:* we also intend to address the "white-box" aspects for *multi-class classification* and *regression*.

Additionally, we will apply Silas to more industrial projects. As Silas' high-performance computing mechanisms can significantly reduce the cost of computational resource, we believe that Silas can be used on machines ranging from mobile devices to workstations, which will broaden the applications of ensemble trees.

References

Abrahams, D. and Gurtovoy, A. (2004). *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional.

Aha, D. W., Kibler, D., and Albert, M. K. (1991). Instance-based learning algorithms. *Machine Learning*, 6(1):37–66.

Altmann, A., Toloşi, L., Sander, O., and Lengauer, T. (2010). Permutation importance: a corrected feature importance measure. *Bioinformatics*, 26(10):1340–1347.

Angiulli, F. (2007). Fast nearest neighbor condensation for large data sets classification. *IEEE Transactions on Knowledge and Data Engineering*, 19(11):1450–1464.

Barsacchi, M., Bechini, A., and Marcelloni, F. (2020). An analysis of boosted ensembles of binary fuzzy decision trees. *Expert Syst. Appl.*, 154:113436.

Bishop, C. M. (2007). *Pattern recognition and machine learning, 5th Edition*. Information science and statistics. Springer.

Bonacina, M. P. (2017). Automated reasoning for explainable artificial intelligence. In *ARCADE 2017, 1st International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements, Gothenburg, Sweden, 6th August 2017*, pages 24–28.

Breiman, L. (2001). Random forests. *Mach. Learn.*, 45(1):5–32.

Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and Regression Trees*. Wadsworth.

Bride, H., Dong, J., Dong, J. S., and Hóu, Z. (2018). Towards dependable and explainable machine learning using automated reasoning. In *Formal Methods and Software Engineering - 20th International Conference on Formal Engineering Methods, ICFEM 2018, Gold Coast, QLD, Australia, November 12-16, 2018, Proceedings*, pages 412–416.

Carmack, J. (2012). In-depth: Functional programming in C++. `https://www.gamasutra.com/view/news/169296/Indepth_Functional_programming_in_C.php`.

Caruana, R., Lou, Y., Gehrke, J., Koch, P., Sturm, M., and Elhadad, N. (2015). Intelligible models for healthcare: Predicting pneumonia risk and hospital 30-day readmission. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 1721–1730, New York, NY, USA. ACM.

Chen, T. and Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 785–794.

Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A. (2002). NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark. Springer.

Cochran, W. G. (1977). *Sampling Techniques, 3rd Edition.* John Wiley.

Cook, D. (2016). *Practical machine learning with H2O: powerful, scalable techniques for deep learning and AI.* O'Reilly Media, Inc.

Cui, Z., Chen, W., He, Y., and Chen, Y. (2015). Optimal action extraction for random forests and boosted trees. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 179–188, New York, NY, USA. ACM.

Cumby, C., Fano, A., Ghani, R., and Krema, M. (2004). Predicting customer shopping lists from point-of-sale purchase data. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 402–409. ACM.

De, A. and Chowdhury, A. S. (2020). Dti based alzheimer's disease classification with rank modulated fusion of cnns and random forest. *Expert Syst. Appl.*, Article In Press.

de Moura, L. and Bjørner, N. (2008). Z3: An efficient SMT solver. In Ramakrishnan, C. R. and Rehof, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg. Springer Berlin Heidelberg.

De Moura, L. and Bjørner, N. (2011). Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77.

Dua, D. and Graff, C. (2019). UCI machine learning repository. `http://archive.ics.uci.edu/ml`.

Ehlers, R. (2017). Formal verification of piece-wise linear feed-forward neural networks. In *Automated Technology for Verification and Analysis - 15th International*

Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings, pages 269–286.

Eliot, L. and Eliot, M. (2017). *Autonomous Vehicle Driverless Self-Driving Cars and Artificial Intelligence: Practical Advances in AI and Machine Learning*. LBE Press Publishing, 1st edition.

Esteve, M., Aparicio, J., Rabasa, A., and Rodríguez-Sala, J. J. (2020). Efficiency analysis trees: A new methodology for estimating production frontiers through decision trees. *Expert Syst. Appl.*, 162:113783.

Freund, Y. and E Schapire, R. (1999). A short introduction to boosting. *Journal of Japanese Society for Artificial Intelligence*, 14:771–780.

Geurts, P., Ernst, D., and Wehenkel, L. (2006). Extremely randomized trees. *Machine Learning*, 63(1):3–42.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011*, pages 315–323.

Gomez-Uribe, C. A. and Hunt, N. (2016). The netflix recommender system: Algorithms, business value, and innovation. *ACM Trans. Management Inf. Syst.*, 6(4):13:1–13:19.

Goré, R., Olesen, K., and Thomson, J. (2014). Implementing tableau calculi using BDDs: BDDTab system description. In *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*, pages 337–343.

Hara, S. and Hayashi, K. (2018). Making tree ensembles interpretable: A Bayesian model selection approach. In *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain*, pages 77–85.

Harasymiv, V. (2015). Lessons from 2 million machine learning models on Kaggle. `https://www.kdnuggets.com/2015/12/harasymiv-lessons-kaggle-machine-learning.html`.

Hart, P. (1968). The condensed nearest neighbor rule (corresp.). *IEEE Transactions on Information Theory*, 14(3):515–516.

Hastie, T., Rosset, S., Zhu, J., and Zou, H. (2009). Multi-class AdaBoost. *Statistics and its Interface*, 2(3):349–360.

He, J., Yalov, S., and Hahn, P. R. (2019). Xbart: Accelerated Bayesian additive regression trees. In Chaudhuri, K. and Sugiyama, M., editors, *Proceedings of Machine Learning Research*, volume 89 of *Proceedings of Machine Learning Research*, pages 1130–1138. PMLR.

Heer, N. (2019). Speed comparison of programming languages. `https://github.com/niklas-heer/speed-comparison`.

Hinton, G., Deng, L., Yu, D., Dahl, G., rahman Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T., and Kingsbury, B. (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Process. Mag.*, 29(6):82–97.

Iorio, C., Aria, M., D'Ambrosio, A., and Siciliano, R. (2019). Informative trees by visual pruning. *Expert Syst. Appl.*, 127:228–240.

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T. (2017). LightGBM: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 3146–3154.

Kononenko, I. (2001). Machine learning for medical diagnosis: History, state of the art and perspective. *Artif. Intell. Med.*, 23(1):89–109.

Liffiton, M. H. and Sakallah, K. A. (2009). Generalizing core-guided max-sat. In *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, pages 481–494.

Liu, X., Wu, J., and Zhou, Z. (2009). Exploratory undersampling for class-imbalance learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 39(2):539–550.

Losing, V., Wersing, H., and Hammer, B. (2018). Enhancing very fast decision trees with local split-time predictions. In *IEEE International Conference on Data Mining, ICDM 2018, Singapore, November 17-20, 2018*, pages 287–296.

Lundberg, S. M. and Lee, S.-I. (2017). A unified approach to interpreting model predictions. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc.

More, A. (2016). Survey of resampling techniques for improving classification performance in unbalanced datasets. arXiv:1608.06048.

OpenML (2019). Creditcard dataset. `https://www.openml.org/d/1597`.

Pafka, S. (2018). A minimal benchmark for scalability, speed and accuracy of commonly used open source implementations of the top machine learning algorithms for binary classification. `https://github.com/szilard/benchm-ml`.

Pafka, S. (2019). Flight dataset. `https://github.com/szilard/benchm-ml/tree/master/z-other-tools`.

Panayotov, V., Chen, G., Povey, D., and Khudanpur, S. (2015). Librispeech: An ASR corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5206–5210.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., VanderPlas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *J. Mach. Learn. Res.*, 12:2825–2830.

Piltaver, R., Lustrek, M., Dzeroski, S., Gjoreski, M., and Gams, M. (2021). Learning comprehensible and accurate hybrid trees. *Expert Syst. Appl.*, 164:113980.

Post, N. Y. (2016). Toddler asks Amazon's Alexa to play song but gets porn instead, New York Post, December 30 [online]. `https://nypost.com/2016/12/30/toddler-asks-amazons-alexa-to-play-song-but-gets-porn-instead/`.

Povey, D., Ghoshal, A., Boulianne, G., Burget, L., Glembek, O., Goel, N., Hannemann, M., Motlicek, P., Qian, Y., Schwarz, P., Silovsky, J., Stemmer, G., and Vesely, K. (2011). The Kaldi speech recognition toolkit. In *IEEE 2011 workshop on automatic speech recognition and understanding, IEEE Signal Processing Society*.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Ribeiro, M. T., Singh, S., and Guestrin, C. (2016). "Why should I trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 1135–1144.

Rodríguez, J. J., Juez-Gil, M., Arnaiz-González, Á., and Kuncheva, L. I. (2020). An experimental evaluation of mixup regression forests. *Expert Syst. Appl.*, 151:113376.

Ross, C. and Swetlitz, I. (2018). IBM's Watson supercomputer recommended "unsafe and incorrect" cancer treatments, internal documents show, STAT, July 25 [online]. `https://www.statnews.com/2018/07/25/ibm-watson-recommended-unsafe-incorrect-treatments/`.

Ruggieri, S. (2017). Enumerating distinct decision trees. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2960–2968, International Convention Centre, Sydney, Australia. PMLR.

Schapire, R. E. (2013). Explaining AdaBoost. In *Empirical Inference - Festschrift in Honor of Vladimir N. Vapnik*, pages 37–52.

Shannon, C. E. (1948). A mathematical theory of communication. *Bell system technical journal*, 27(3):379–423.

Tanno, R., Arulkumaran, K., Alexander, D., Criminisi, A., and Nori, A. (2019). Adaptive neural trees. In Chaudhuri, K. and Salakhutdinov, R., editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6166–6175, Long Beach, California, USA. PMLR.

Tannor, P. and Rokach, L. (2019). Augboost: Gradient boosting enhanced with stepwise feature augmentation. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 3555–3561. International Joint Conferences on Artificial Intelligence Organization.

Tao, Q., Li, Z., Xu, J., Xie, N., Wang, S., and Suykens, J. A. (2020). Learning with continuous piecewise linear decision trees. *Expert Syst. Appl.*, Article In Press.

Tomek, I. (1976). Two modifications of CNN. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-6(11):769–772.

Turkson, R. E., Baagyere, E. Y., and Wenya, G. E. (2016). A machine learning approach for predicting bank credit worthiness. In *2016 Third International Conference on Artificial Intelligence and Pattern Recognition (AIPR)*, pages 1–7.

Törnblom, J. and Nadjm-Tehrani, S. (2019). *Formal Verification of Random Forests in Safety-Critical Applications: 6th International Workshop, FTSCS 2018, Gold Coast, Australia, November 16, 2018, Revised Selected Papers*, pages 55–71. Springer.

Vanschoren, J., van Rijn, J. N., Bischl, B., and Torgo, L. (2013). OpenML: networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60.

Wright, M. and Ziegler, A. (2017). Ranger: A fast implementation of random forests for high dimensional data in C++ and R. *Journal of Statistical Software*, Vol. 77.

Yang, B., Shen, S., and Gao, W. (2019). Weighted oblique decision trees. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The*

Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019, pages 5621–5627.

Yang, L., Liu, S., Tsoka, S., and Papageorgiou, L. G. (2017). A regression tree approach using mathematical programming. *Expert Syst. Appl.*, 78:347–357.

Zhang, W. and Ntoutsi, E. (2019). Faht: An adaptive fairness-aware decision tree classifier. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 1480–1486. International Joint Conferences on Artificial Intelligence Organization.

Appendix A. Full Table for Mid-Large Datasets

Table A.12 shows the experimental results for 100 trees, leaf size 64, and tree max depth 64 from all the tested tools and methods. We compare the Silas methods, which include 10 combinations of the the sub-algorithms described in the previous section, with H2O and Ranger. The settings used are: 100 trees, 64 max tree depth, 64 min leaf size, and default settings of each tool otherwise. The predictive performance measures include area under the ROC curve (AUC) and predictive accuracy (Acc). For multi-class classification, we compute an approximation of the multi-class AUC (avg. of one vs rest) and the overall accuracy for all classes. H2O and Ranger do not report multi-class AUC by default, so we only compare accuracy for multi-class datasets. Training time is in seconds, and it includes the time for loading data and computing the predictive performance. The "Total-binary" row reports the total numbers for binary classification datasets. Except for flight, which is tested on a distinct testing dataset, all other datasets are tested using 10-fold cross-validation and the AUC and anccuracy are the average of those validations. We run the experiment 10 times (i.e., 10 times 10-fold cross-validations for most datasets) and report the average AUC, accuracy, and time. The 95% confidence interval of AUC and accuracy for the tested methods are usually smaller than 0.001, so we do not show them in the table. We highlight the top results in bold font. Top results are defined as those that are the best when rounded to the 3rd decimal point.

Appendix B. Experimental Results for Mid-Large Datasets Using Default Settings of H2O and Ranger

Table B.13 gives the experimental results on mid-large datasets from H2O and Ranger using their default settings and 100 trees. By Default, H2O uses leaf size 1 and tree max depth 20, while Ranger uses leaf size 1 and unlimited tree depth. Time is in seconds. H2O refused to run the led5000 dataset because it estimated that it would use more than the 32GB memory limit, although it could use some swap memory.

	Silas (UB+RT)			Silas (UB+GT)			Silas (NB+RT)			Silas (NB+GT)			H2O			Ranger		
Dataset	AUC	Acc	Time	AUC	Acc	Time	AUC	Acc	Time	AUC	Acc	Time	AUC	Acc	Time	AUC	Acc	Time
diabetes	0.8298	0.7498	0	0.8082	0.7296	0	0.8297	0.7520	0	0.8087	0.7488	0	0.8155	0.7550	4	**0.8390**	**0.7637**	0
kick	0.7633	0.7341	4	0.7676	0.7366	8	**0.7680**	0.8998	13	**0.7682**	**0.9008**	24	0.7609	0.8669	95	0.7658	**0.9011**	39
adult	0.9054	0.7937	4	0.9164	0.8116	6	0.9079	0.8552	6	0.9174	0.8646	10	0.9147	0.8492	31	**0.9185**	**0.8656**	11
mozilla4	0.9694	0.9297	1	0.9666	0.9269	2	0.9724	0.9363	1	0.9679	0.9286	3	0.9655	0.9346	14	**0.9790**	0.9459	2
jm1	0.7344	0.6801	1	0.7431	0.6784	1	0.7490	0.8149	1	0.7527	0.8146	3	0.7319	0.6997	15	**0.7550**	0.8169	2
creditcard	0.9799	0.9878	5	0.9793	0.9801	5	0.9777	**0.9992**	91	0.9635	**0.9989**	12	0.9760	**0.9993**	161	0.9602	**0.9994**	557
flight	0.7307	0.6411	12	0.7535	0.6691	17	0.7377	0.7942	24	**0.7615**	**0.8037**	34	0.7442	0.7996	53	0.7225	0.7838	139
mnist-784	0.9974	0.9459	66	0.9762	0.7743	26	0.9975	0.9465	72	0.9762	0.7562	27		0.9392	3026		0.9580	143
fashion-mnsit	**0.9881**	0.8579	84	0.9352	0.4977	14	**0.9881**	0.8582	85	0.9351	0.4972	14		0.8567	3266		0.8754	176
connect-4	0.8560	0.6850	5	0.8668	0.6923	6	0.8927	0.7697	12	0.8917	0.7843	13		0.7269	174		0.7700	19
led5000	0.9319	0.6275	398	0.9043	0.5691	195	0.9320	0.6276	396	0.9044	0.5688	192		0.6236	8395		0.6252	603
walking-activity	0.9419	0.5131	21	0.8232	0.2700	5	**0.9676**	0.6363	57	0.8446	0.1853	7		0.6174	948		0.6488	116
cover-type	0.9397	0.6387	48	0.8370	0.2895	11	0.9901	0.8841	273	0.8925	0.4883	26		0.8763	4444		0.8315	343
Total-binary	5.9128	5.5163	**26**	5.9348	5.5322	38	5.9424	6.0517	138	5.9399	6.0601	86	5.9406	5.9591	378	5.9399	6.0764	750
Total	11.5679	9.7844	647	11.2775	8.6250	**295**	11.7104	10.7742	1034	11.3845	9.3402	365	5.9087	10.5441	20625	5.9399	10.7853	2151

	Silas (AB+RT)			Silas (AB+GT)			Silas (PS+RT)			Silas (PS+GT)			Silas (WC+RT)			Silas (WC+GT)		
Dataset	AUC	Acc	Time	AUC	Acc	Time	AUC	Acc	Time	AUC	Acc	Time	AUC	Acc	Time	AUC	Acc	Time
diabetes	0.8129	0.7486	0	0.7528	0.7067	0	0.8297	0.7584	0	0.8043	0.7427	0	0.8162	0.7551	0	0.7891	0.7283	0
kick	0.7625	0.8837	18	0.7393	0.8845	29	0.7675	0.8976	26	0.7662	0.8985	33	0.7654	**0.9009**	21	0.7543	0.8997	31
adult	0.8922	0.8259	9	0.8906	0.8152	13	0.9072	0.8490	11	0.9175	0.8579	14	0.9109	0.8571	11	0.9143	0.8592	15
mozilla4	0.9776	0.9338	2	0.9699	0.9244	4	0.9710	0.9329	2	0.9674	0.9282	3	**0.9789**	**0.9486**	2	0.9723	0.9378	4
jm1	0.7433	0.7964	2	0.7065	0.7659	4	0.7456	0.8110	2	0.7471	0.8138	3	0.7498	**0.8184**	3	0.7305	0.8165	4
creditcard	0.9747	**0.9992**	121	0.9648	**0.9994**	142	**0.9814**	**0.9993**	127	0.9694	**0.9994**	113	0.9771	**0.9994**	111	0.9697	**0.9992**	64
flight	0.7419	0.7833	37	0.7437	0.7850	48	0.7365	0.7983	28	0.7591	0.8026	36	0.7458	0.8013	41	0.7570	**0.8037**	53
mnist-784	0.9983	0.9624	107	0.9808	0.8246	61	0.9975	0.9466	522	0.9765	0.7674	483	**0.9986**	**0.9659**	113	0.9858	0.9062	68
fashion-mnsit	**0.9879**	0.8687	128	0.9395	0.5732	53	**0.9881**	0.8579	545	0.9353	0.4884	489	**0.9880**	**0.8842**	133	0.9356	0.5774	54
connect-4	0.8994	0.8365	18	0.8924	0.8293	18	0.8861	0.7847	28	0.8906	0.7991	29	0.8988	**0.8489**	20	**0.9003**	**0.8494**	21
led5000	**0.9329**	0.6311	531	0.9001	0.5682	253	0.9312	0.6261	623	0.9044	0.5684	433	0.9241	**0.6474**	582	0.9080	0.6007	276
walking-activity	0.9667	0.6348	63	0.8491	0.1824	10	0.9674	0.6360	76	0.8440	0.1860	27	0.9643	**0.6663**	71	0.8427	0.2199	11
cover-type	0.9926	0.9253	355	0.8659	0.5801	47	0.9890	0.8798	491	0.8968	0.4964	275	**0.9961**	**0.9544**	398	0.8889	0.6451	53
Total-binary	5.9051	5.9709	189	5.7677	5.8811	240	5.9389	6.0465	197	5.9310	6.0430	203	**5.9442**	**6.0808**	188	5.8872	6.0444	170
Total	11.6827	10.8299	1391	11.1955	9.4388	682	11.6982	10.7775	2482	11.3786	9.3488	1937	**11.7140**	**11.0480**	1505	11.3484	9.8432	653

Table A.12: Experimental results for medium to large datasets.

	H2O default			Ranger default		
Dataset	AUC	Acc	Time	AUC	Acc	Time
diabetes	0.8212	0.7639	4	0.8337	0.7617	0
kick	0.7520	0.8710	164	0.7618	0.9019	45
adult	0.9178	0.8541	45	0.9175	0.8648	13
mozilla4	0.9823	0.9534	15	0.9824	0.9542	3
jm1	0.7631	0.7513	21	0.7606	0.8225	3
creditcard	0.9760	0.9996	219	0.9558	0.9995	565
flight	0.7516	0.6781	126	0.7212	0.7838	156
Mnist-784		0.9689	5815		0.9698	162
Fashion-mnsit		0.8864	6771		0.8836	200
Connect-4		0.8214	302		0.7880	22
led5000	Out of Memory				0.6473	695
Walking-activity		0.6694	1054		0.6638	138
Cover-type		0.9286	3755		0.8427	335
Average	0.8520 (binary)	0.8455 (excl. led5000)	1,524	0.848 (binary)	0.8372	180

Table B.13: Experimental results using H2O and Ranger's default settings and 100 trees.

Appendix C. Experimental Results for ASR-200 dataset Using Default Settings of Ranger and Scikit-learn

Table C.14 gives the experimental results on the ASR-200 dataset from Ranger and Scikit-learn using their default settings. Time and memory usage is measured the same way as in Table 8. The main difference from Table 9 is that only $\sqrt{D}$, where D is the total number of features, features are considered when selecting each node. We also give the results from selected Silas methods under the same settings. Note that the actual parameters in Silas have different code-names than the abbreviations used in this paper. We give the mapping below. $RT \rightarrow$ RdGreedy1D; $GT \rightarrow$ GreedyNarrow1D; $UB \rightarrow$ ClassicForest; $NB \rightarrow$ SimpleForest; $PS \rightarrow$ PrototypeSampleForest; $WC \rightarrow$ CascadeForest; $AB \rightarrow$ AdaBoostForest; $HV \rightarrow$ SimpleValueForest.

ASR-200	#trees	Acc	Time (s)	Mem (GB)
Silas (NB+RT)	100	0.5115	48	27
Silas (WC+RT)	100	0.5124	59	27
Silas (HV+RT)	100	0.5738	45	2.6
Ranger	100	0.5487	4588	2.9
Scikit-learn	100	0.5467	397	30 + 32.9 (swap)

Table C.14: Experiment results for the ASR-200 dataset using Ranger and Scikit-learn's default settings and 100 trees.