# Experiments on the generalization of machine learning algorithms

Arthur Franz[0000−0003−2255−2431]

Independent researcher
af@occam.com.ua

**Abstract.** The inductive programming system WILLIAM is applied to machine learning tasks, in particular, centralization, outlier detection, linear regression, linear classification and decision tree classification. These examples appear as a special case of WILLIAM's general operation of trying to compress data without any special tuning.

**Keywords:** inductive programming · incremental compression · algorithmic complexity · machine learning · WILLIAM · generalization

## Introduction

Machine learning (ML) techniques and applications have revolutionized the world in recent decades, mostly promising to learn by themselves from data, as opposed to hand-crafted algorithms and feature detectors from earlier times. However, the term "learning", just as many other AI terms, has turned out to be euphemistic and exaggerating, referring mostly to parameter optimization within a fixed representation, and only vaguely related to human learning whose breadth and scope has remained unmatched. In the AGI context it therefore appears to be important to make machine learning truly general, thereby boosting its success even more.

Since various ML algorithms utilize different objective functions the first step is to identify a common optimization goal. Indeed, the minimum description length (MDL) principle has emerged to be a promising candidate [1, 7]:

> "The goal of statistical inference may be cast as trying to find regularity in the data. 'Regularity' may be identified with 'ability to compress.' MDL combines these two insights by viewing learning as data compression: it tells us that, for a given set of hypotheses $\mathcal{H}$ and data set $\mathcal{D}$, we should try to find the hypothesis or combination of hypotheses in $\mathcal{H}$ that compresses $\mathcal{D}$ most." (p. 8 in [1])

Unfortunately, applications of MDL have been mostly limited to the selection of ML models and parameter numbers, including meta-parameter selection as in AutoML, failing to break out from a given representation space into a broader set of algorithmic data descriptions. The fact that such a widening could be important for the generalization of machine learning heading toward AGI has long been suggested by the father of algorithmic probability, Ray Solomonoff [8, 9]. However, these suggestions did not go beyond theory, bringing forth a long-standing central problem for AGI: the difficulty of making inference both general and efficient. To the best of my knowledge, apart from the present paper, the only attempt of going beyond theory and realizing machine learning by means of a general compression algorithm was done just recently, using a novel set of techniques for computing lower bounds on algorithmic probability based on the Coding theorem and Block Decomposition methods [5].

This paper explores the abilities of WILLIAM [4] – an inductive programming method based on the theory of incremental compression (IC) [3] – to deal with a set of simple machine learning problems. In Section 1 a short overview of WILLIAM's novel aspects is given followed by the discussion of its application to data centralization, outlier detection, linear regression, linear classification and decision tree classification.

## 1 Overview of the algorithm

WILLIAM's core functionality is given by an inductive programming algorithm already described in [4], albeit with several major improvements. Most importantly, the data representation has moved up from trees to directed acyclic graphs (DAGs), enabling the reuse of previously computed values. Further, the graph is bipartite (see figures below), consisting of operator nodes and value nodes (denoted by a box). Another innovation is the principle that any data used by the algorithm has to be computed by the algorithm itself. For example, even integers are not "given for free", except for the integer 1 (called *vacuum*), all other integers have to be computed using the given operators.

WILLIAM's main operation is to implement IC, i.e. given a data string $x$ to find a description by a composition of functions, $x = f_1\left(f_2\left(\cdots f_s\left(r_s\right)\right)\right)$, by searching for stacked autoencoders. In particular, for a given *residuum* $r_{i-1}$ a pair of functions $\left(f_i, f_i'\right)$ is searched such that $r_{i-1} = f_i\left(r_i\right)$, where $r_i := f_i'\left(r_{i-1}\right)$ and $l\left(f_i\right) + l\left(r_i\right) < l\left(r_{i-1}\right)$, i.e. compression is achieved at every step ($x \equiv r_0$). The $f_i$ are called *features* and $f_i'$ *descriptive maps*. One of the main results is that the (prefix) Kolmogorov complexity $K(x)$ can be approximated in this way, when picking the shortest possible feature $f_i^*$ at every step:

$$K\left(x\right) = \sum_{i=1}^{s} l\left(f_i^*\right) + K\left(r_s\right) + O\left(s \cdot \log l(x)\right) \tag{1}$$

In practice, in order to bound the search for descriptive maps, the shortest autoencoders, i.e. the shortest sum $l\left(f_i\right) + l\left(f_i'\right)$ is searched, in compliance with computable IC (see Greedy-ALICE [3, Chapter 4.1]).

In order to apply the introduced notions to a DAG, a residual is represented by a *cross section*, defined as a set of value nodes separating the graph. The algorithm takes a set of operators and tries to attach them to the current residual cross section (= target cross section at start), computing new values on the way. Attaching nodes from above, i.e. using the residual in order to compute new values from it, corresponds to the descriptive map (e.g. len, getitem, urange and repeat operators in Figs. 3B and 1). Attaching nodes from below entails an inversion of the involved operators and corresponds to parts of a feature (e.g. setitem operator in Fig. 1). Cycles can be introduced in this way, but are removed once the graph is cut at its so-called *bottleneck*, defined as the shortest cross section. For example, the sum of description lengths (DLs) of the bottleneck nodes in Fig. 1 is smallest compared to all other possible cross sections. The cutting is performed as soon as the bottleneck has a lower DL than the current residual (the blue nodes in Fig. 1). At this point the bottleneck becomes the new residual and the algorithm continues iterating ad infinitum. Additionally, values in nodes can be replaced by other values in other nodes and propagated through the graph (see e.g. Fig. 4F-H). The search for the shortest compressing autoencoder in this way is exhaustive, i.e. all directed graphs are being constructed ordered by their size, even though only those combinations leading to the solution are shown in the figures.

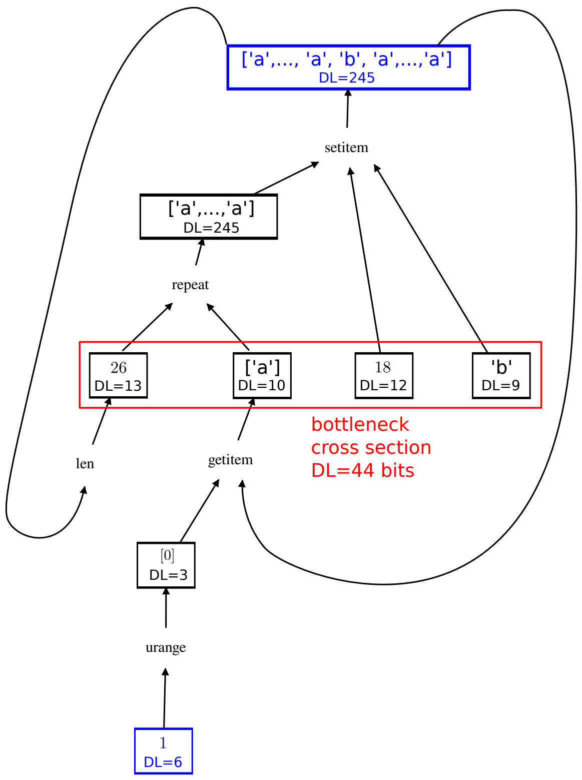

**Fig. 1.** The directed bipartite graph is cut at the most narrow cross section, called the bottleneck (red, DL = 44 bits), leading to the shortest description of the target cross section (blue, DL = 251 bits). The descriptive map, i.e. the graph from target to bottleneck containing the len, getitem and urange operators are removed. The result is displayed in Fig. 3C. DL denotes the default description length defined in paragraph *Default descriptions*.

*Default descriptions* A positive integer $n \in \mathbb{N}$ is described by the Elias delta code [2]. The length $l(n)$ of the (default) description is given by $E(n) = \lfloor \log_2(n) \rfloor + 2 \lfloor \log_2 (\lfloor \log_2(n) \rfloor + 1) \rfloor + 1$ bits. Including zero and negatives, an integer $n \in \mathbb{Z}$ has DL $l(n) = E(2|n| + 1)$. Rational numbers $x$ are described by a pair of mantissa $m \in \mathbb{Z}$ and exponent $a \in \mathbb{Z}$, $x = m \cdot 10^a$. Chars take the fixed amount of 8 bits relating to the ASCII table. Strings $s$ carry the DL $l(s) = E(|s|) + 8|s|$ by describing their length $|s|$ and then each char separately. Similarly, for arrays, lists, tuples and sets their lengths are described following the elementwise description of their contents.

## 2 Results

### 2.1 Centralization

Consider a set of one-dimensional data, $x = \{x_1, \ldots, x_n\}$ sampled from a Gaussian $X \sim \mathcal{N}(\mu, \sigma)$. Centralization refers to the subtraction of the sample mean $\hat{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$ from every $x_i$: $x'_i = x_i - \hat{x}$. This preprocessing step is meaningful if $\sigma \ll \mu$, i.e. the cluster center is a large number relative to the standard deviation. Centralization transforms $n$ large numbers $x_i$ into one large number $\hat{x}$ and $n$ small numbers $x'_i$.

This example consists of of the target cross section $x$ being an array of $n = 1000$ i.i.d. samples taken from $\mathcal{N}(143, 1)$ and a precision of 4 decimals (Fig. 2). As described Section 1, all sorts of operators are attached to the target from above and from below, leading to the computation of the sample mean $\langle x \rangle$ in Fig. 2A, followed by an inversion of the add operator using that target $x$ and the just computed mean $\langle x \rangle$, i.e. the "error" is $E = x - \langle x \rangle$ (denoting elementwise subtraction of $\langle x \rangle \approx 143$ from the array $x$, Fig. 2B). Since the significand of the entries in $x$ have 7 digits and the error merely 4 the DL of the error is much lower than that of the target, $l(E) \ll l(x)$. Finally, the `mean` operator is removed since it is part of the descriptive map (during bottleneck cutting as in Fig. 1). It was merely helpful in computing the residual but does not belong to the description of the target (Fig. 2C).The residual cross section consisting of one big number $\langle x \rangle$ and $n$ smaller numbers $x - \langle x \rangle$ is shorter, i.e. compression has been achieved.

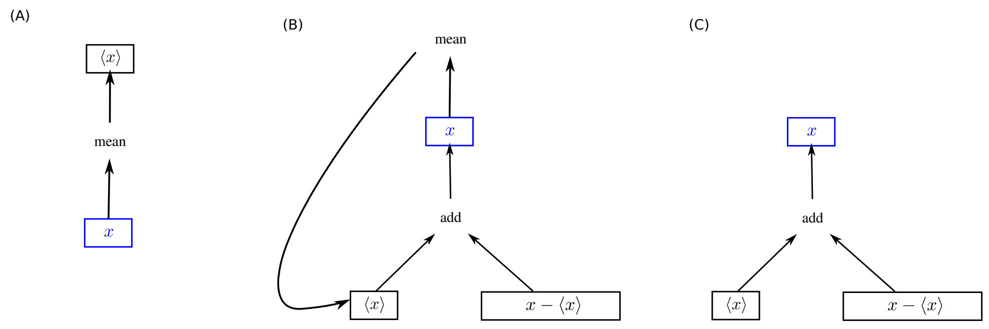

**Fig. 2.** Data centralization $x \to x - \langle x \rangle$ performed by WILLIAM, shown as a series of operators attached to target data array $x$.

## 2.2 Outlier detection

Fig. 3 shows a target cross section consisting of a list of 25 chars ’a’ and a single char ’b’ at index 18 in the first value node, and the number 1 in the second value node, also called *vacuum* node, since it is merely a helper node and not to be compressed. The final residuum consists of of leaves in Fig. 3C, i.e. the value nodes 1, 26, ’a’, 18 and ’b’. Since the residuum’s DL is shorter than that of the target (see also Fig. 1), compression has been achieved and the outlier ’b’ has been filtered out. Note how the description of the data becomes meaningful and interpretable after proper compression: a verbal description of the target as “a list with length 26 consisting of letters ’a’ with a letter ’b’ at index 18” corresponds to the retrieved leaf values.

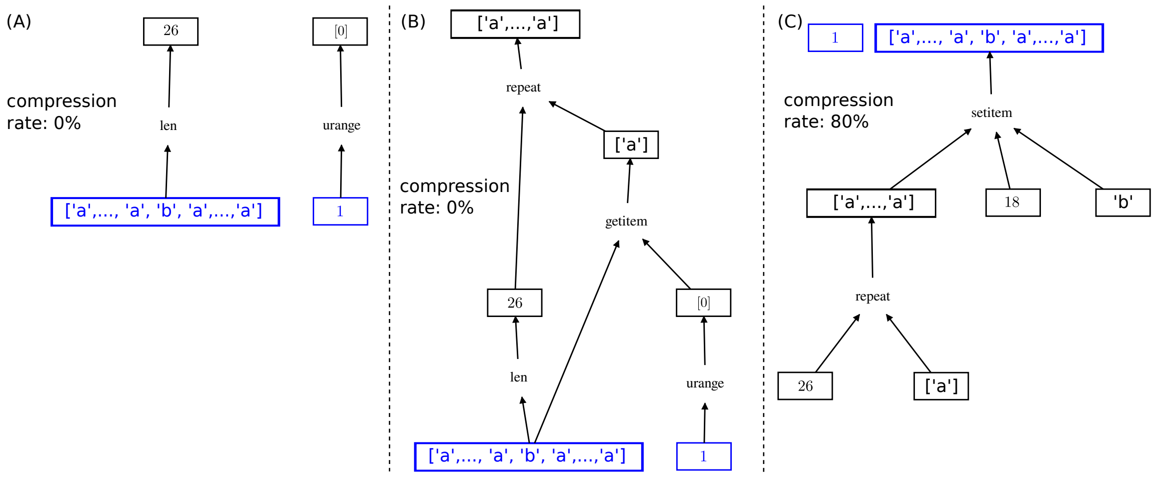

**Fig. 3.** Outlier detection. (A) and (B) show how various operators are applied to the blue target cross section, leading to a “cleaned” list consisting of chars ’a’ only. (C) The cleaned list is used to invert operator `setitem` yielding the index 18 and thar char ’b’.

## 2.3 Linear regression

Fig. 4 shows how the target cross section consisting of $x$ and $y$ is compressed incrementally. Through a series of steps the value nodes of the target $x$ and $y$ are connected, hence instead of both being described independently $y$ is described in terms of $x$ which reduces the DL. Subplots (A)-(D) show how the numbers 1, 2, 4 and 16 are subtracted from $y$ in order to centralize the data. As we have seen in Subsection 2.1, this process leads to compression: the residual is shorter than the target $l\,(2x+2+\epsilon)+l\,(16.0)+l\,(x) < l\,(y)+l\,(x)$. $x$ is also a leaf, describing “itself”, neglecting the comparatively short description of the operators. Note that there is no specialized optimization involved here. The numbers 1, 2, 4 and 16 are computed by using the `add`-operator applied to the existing value nodes, e.g. `16` $=$ `add(8, 8)`, starting with the vacuum 1 (not shown in the figure). Subplot (E) uses the value node $x$ and subtracts it from the residual node $2x+2+\epsilon$. Overall the residual cross section (leaves) is shorter than the previous residual: $l\,(x)+l\,(x+2+\epsilon)+l\,(16.0) < l\,(2x+2+\epsilon)+l\,(16.0)+l\,(x)$. Subplot (F) shows that the target node $x$ is compressed further by

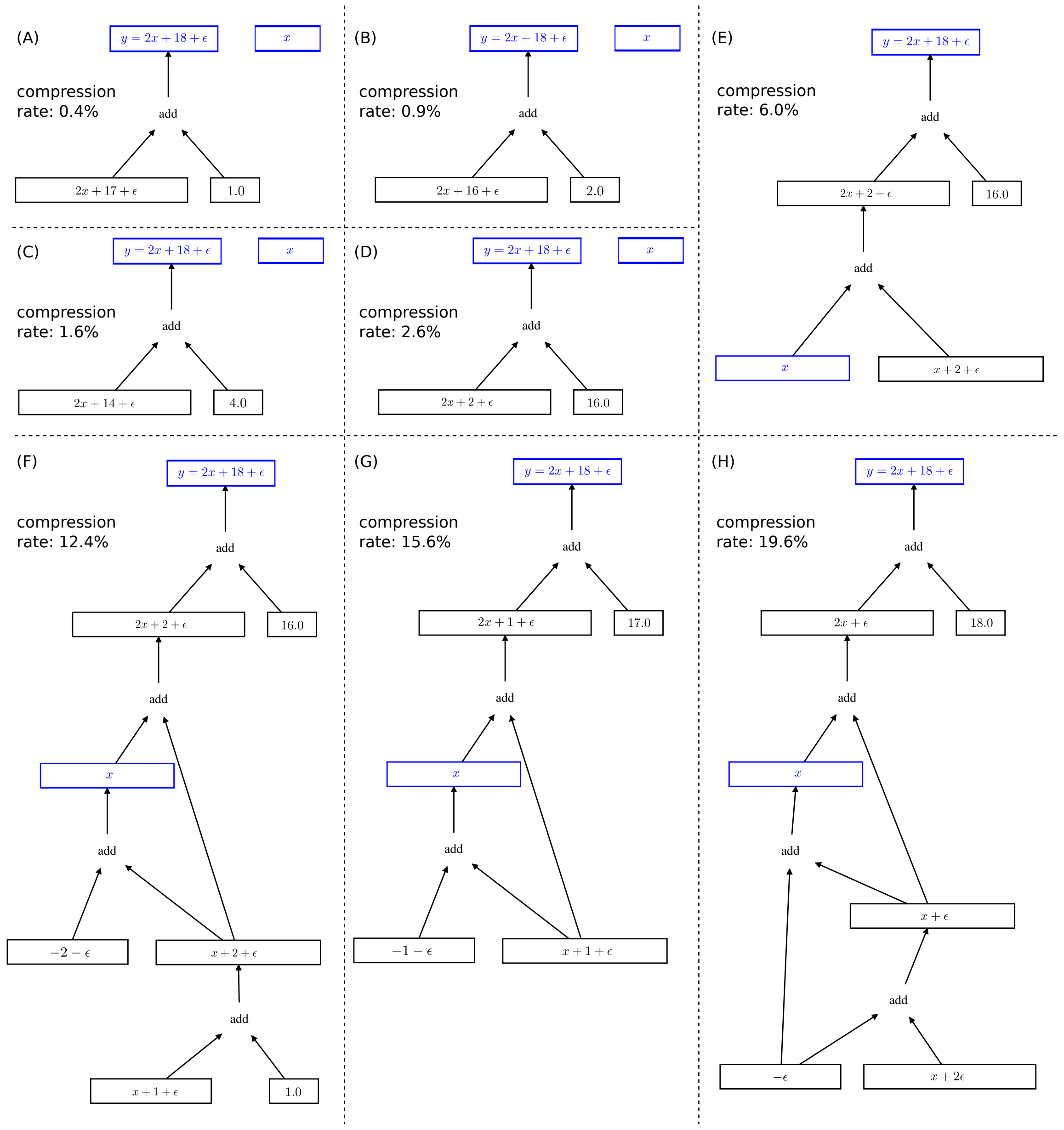

**Fig. 4.** Linear regression performed by WILLIAM. $x$ is an array of length $n = 1000$ with values ranging between $-10$ and 10 in steps of 0.2, $\epsilon$ is an array of $n$ i.i.d. samples from the standard normal distribution $\mathcal{N}(0, 1)$ and $y$ given by $2x + 18 + \epsilon$ (see Fig. 7A). The target is incrementally compressed.

subtracting it from the $x + 2 + \epsilon$ node. Note that $x$ and $y$ are treated completely equally as target nodes that are to be compressed. There is no special meaning of “dependent” and “independent” variables involved here as usually in regression tasks. Subplots (G) and (H) achieve even more compression by replacing 16 to 17 to 18 and propagating this change through the graph. Fig. 7A visualizes how the various estimates (red dashed lines) incrementally fit the data. These estimates are obtained by setting all leaf arrays (=“errors”) to 0.

## 2.4 Linear classification

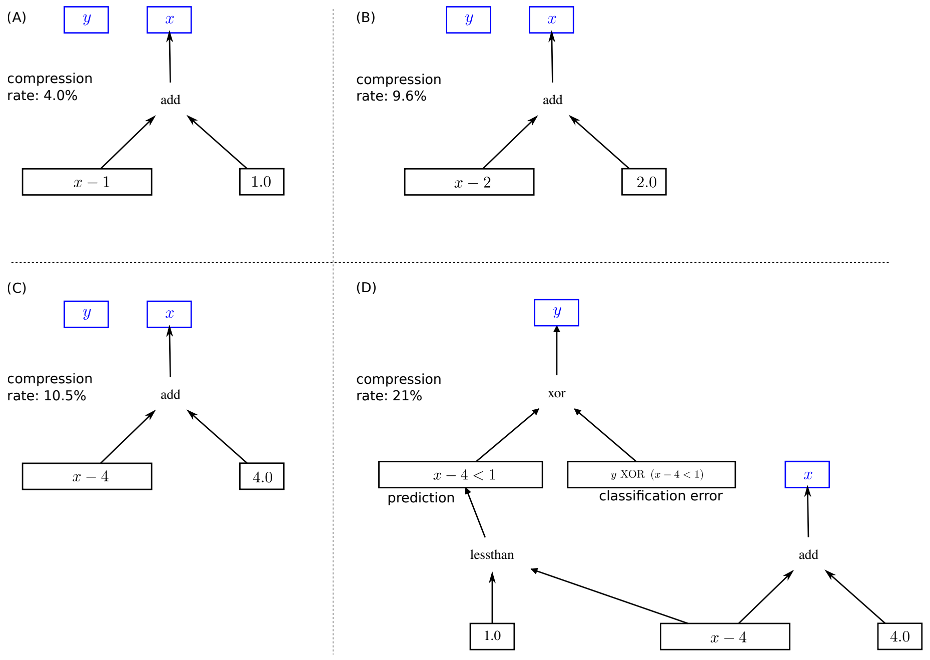

**Fig. 5.** Classification performed by WILLIAM. $y$ is an array of 2 classes, True and False, color coded in Fig. 7B. $x$ is an array of i.i.d. samples taken from $\mathcal{N}(3.5, 1)$ for class True and from $\mathcal{N}(6.5, 1)$ for class False.

Fig. 5 shows how an array of data $x$ is “fitted” the array of classes $y$ (see Fig. 7B for the distribution of $x$). As before, subplots (A)-(C) centralize $x$. In (D) a prediction $p = (x - 4 < 1)$ is generated and subtracted from the class array by the elementwise xor-operator. Compression is achieved due to the fact that the error $y$ XOR $p$ contains True values only if the prediction goes wrong, i.e. a few times in a good classifier. A boolean array with significantly fewer True than False values has a shorter description than an array with a balanced number of True’s and False’s, due to the following reason. There are $\binom{n}{k}$ ways of distributing $k$ True’s in an array of length $n$ hence providing an index $i$ of the permutation together with $k$ and $n$ constitutes a full description of the array. This permutation-based method is part of the default descriptions in WILLIAM. In sum, compression in classification is achieved by replacing array of classes $y$ containing a balanced number of True's and False's by an unbalanced error array having a shorter description.[1]

## 2.5 Decision tree classification

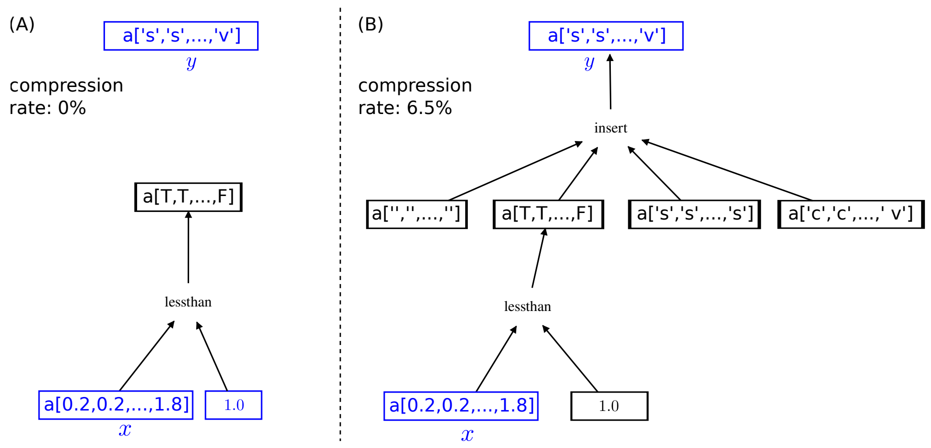

**Fig. 6.** A first step of decision tree classification. The data is taken from Python's scikit-learn Iris Dataset of three different types of flowers, see Fig. 7C. The target cross section (blue) consists of the flower classes setosa ('s'), versicolour ('v'), and virginica ('c') in array $y$. The a[...] notation stands for arrays. The float array $x$ denotes the petal width factor. By applying the threshold 1.0 the setosa types (leaf a['s','s'...,'s']) are separated from the other two types (leaf a['c','c',...,'v']).

Fig. 6 shows the first step in decision tree classification. In subplot (A), lessthan creates a boolean array denoting petal widths smaller than 1. In subplot (B) the insert-operator is inverted: it inserts the array a['s','s'...,'s'] into array a['',...,''] of empty strings at the indices denoted by the boolen array a[T,T...,F]. The remaining indices are filled by a['c','c',...,'v']. As can be glanced from Fig. 7C all setosa flowers have petal width $< 1$ and all the others have petal width $\geq 1$. Therefore, setosa is separated from the rest by this decision tree step. This constitutes compression since the original target $y$ consists of all three types of flowers and the permutation-based description is therefore longer due to the larger multinomial coefficient $\binom{n}{k_1\ k_2}$ as compared to two types of flowers in a['c','c',...,'v'] and a single type in a['s','s'...,'s']. Here only the first decision tree separation is shown but it can be continued using other factors to separate the a['c','c',...,'v'] array. Note that neither the Gini coefficient nor any other particularities of decision tree classification are used to perform this "training" step. There is no need for measuring the dispersion of values by the Gini coefficient since the permutation-based code automatically becomes shorter if there are fewer different classes present in an array and/or the classes are not represented uniformly.

[1] It can be shown that $l\left(\binom{n}{k}\right) + l(k) + l(n) < n$ if $k \ll n$.

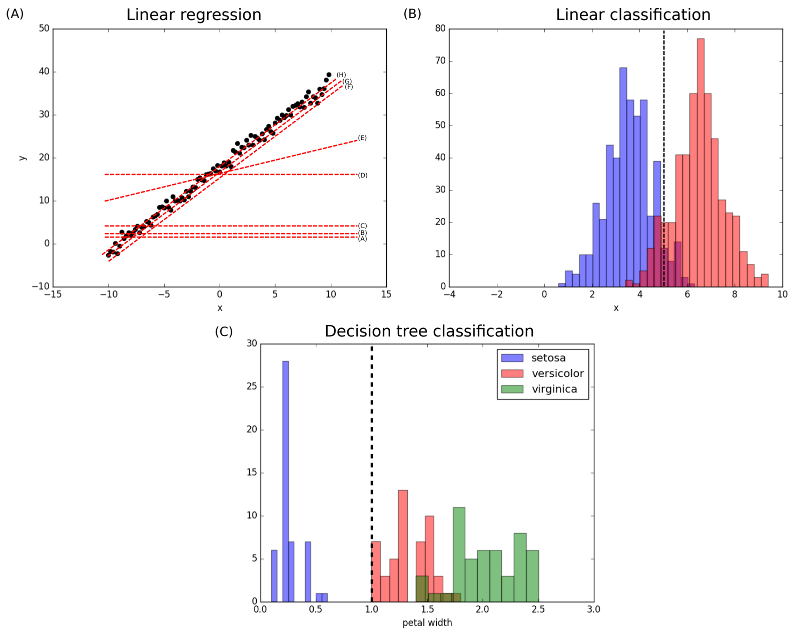

**Fig. 7.** (A) Estimates (red dashed lines) incrementally fit the linear regression data (the letters (A)-(H) correspond the subplots of Fig. 4). (B-C) Distribution of data for classification, the class is color coded.

## Discussion

In this paper we have demonstrated that various machine learning algorithms can be viewed as performing data compression as has been suggested previously in theory. In particular, their core functionality emerges as a special case of WILLIAM's general performance without being specifically tuned to these algorithms. WILLIAM neither uses any specialized ML optimization nor any other heuristics for that matter and is developed in a fully general fashion according to IC theory. This generality enables WILLIAM to deal a wide range of tasks beyond machine learning, as reported in [4]. Nevertheless, these examples had to be rather simple mainly due to the following limitations.

### Limitations

*Accumulation of description overhead* Since IC proceeds greedily, there is no mechanism for avoiding the accumulation of overhead, as discussed in [3]. In particular, eq. (1) entails an overhead of the order $O\left(s \cdot \log l(x)\right)$ where $s$ is the number of compression steps. For example, in the regression case, nothing hinders the algorithm from keeping subtracting 1 from the target array achieving some compression at every step. Currently, such values are being replaced by newly generated values and propagated through the graph (see Fig. 4F-H, where the value `16` is replaced by `17`, since `17` has been previously generated by `add(16,1)`). Even though this propagation solution helps combat the overhead accumulation, it would be helpful to have theoretical guarantees for avoiding overhead altogether.

*Alternative descriptions* In many cases, it is desirable to consider alternative short descriptions of data. IC theory only shows how to search for *some* short description, but not for several ones. One option is to search for many incremental strands in parallel which would however put additional computational strain on performance. A different option is to allow the lack of progress in further compressing the first short description to guide the search for other descriptions – a strategy apparently used by problem solving in the psychology of insight [6]. Again, theoretical guidance would be of great help in this endeavor.

*No reuse of successful functions* An important way of accelerating the algorithm is to reuse successful, i.e. compressing combinations of operators. Currently, a DAG of operators can be used to define a composite operator, which can be inserted an used just like a primitive operator. This makes our graph a hypergraph – there can be graphs inside the nodes. However, it is unclear which subgraph of a solution is to be encapsulated and how it is to be reused. Essentially, this issue comes down to the non-trivial task of finding a theory of memory and its retrieval.

*Computing power and parallelization* On a positive note, while thinking through many tasks over the last years, it appears that most interesting tasks do not appear to demand steps requiring deeper graphs than 6 or 7 operators until some compression is achieved. Currently, WILLIAM manages to search exhaustively through graphs of depth 4-5 running on pure Python code. Rewriting core parts on a faster language in the back end and parallelization could boost the performance considerably.

## Conclusion

If the discussed limitations can be overcome both in theory and practice the results show a promising path to create a general algorithm for solving machine learning problems and going beyond them.

## References

1. Peter D. Grünwald, In Jae Myung, and Mark A. Pitt. *Advances in minimum description length: Theory and applications.* MIT press, 2005.
2. Peter Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.
3. Arthur Franz, Oleksandr Antonenko, and Roman Soletskyi. A theory of incremental compression. *Information Sciences*, 547:28–48, 2021.
4. Arthur Franz, Victoria Gogulya, and Michael Löffler. WILLIAM: A monolithic approach to AGI. In *International Conference on Artificial General Intelligence*, pages 44–58. Springer, 2019.
5. Santiago Hernández-Orozco, Hector Zenil, Jürgen Riedel, Adam Uccello, Narsis A Kiani, and Jesper Tegnér. Algorithmic probability-guided machine learning on non-differentiable spaces. *Frontiers in artificial intelligence*, 3, 2020.
6. Craig A. Kaplan and Herbert A. Simon. In search of insight. *Cognitive Psychology*, 22(3):374–419, 1990.
7. Alexey Potapov. *Raspoznavanie obrazov i mashinnoe vospriatie.* Politechnica, 2007. See https://pureportal.spbu.ru/en/publications.
8. Ray Solomonoff. A system for machine learning based on algorithmic probability. In *Conference Proceedings., IEEE International Conference on Systems, Man and Cybernetics*, pages 298–299 vol.1, 1989.
9. Ray Solomonoff. Algorithmic probability, heuristic programming and AGI. *Advances in Intelligent Systems Research*, 10:151–157, 2010.