The learnability of Business Rules

Olivier Wang[1,2], Changhai Ke[1], Leo Liberti[2], Christian de Sainte Marie[1]

[1] IBM France, 9 Rue de Verdun, 94250 Gentilly, France
[2] CNRS LIX, Ecole Polytechnique, 91128 Palaiseau, France
olivier.wang@polytechnique.edu

Abstract.

Among programming languages, a popular one in corporate environments is Business Rules. These are conditional statements which can be seen as a sort of "programming for non-programmers", since they remove loops and function calls, which are typically the most difficult programming constructs to master by laypeople. A Business Rules program consists of a sequence of "IF *condition* THEN *actions*" statements. Conditions are verified over a set of variables, and actions assign new values to the variables. Medium-sized to large corporations often enforce, document and define their business processes by means of Business Rules programs. Such programs are executed in a special purpose virtual machine which verifies conditions and executes actions in an implicit loop. A problem of extreme interest in business environments is enforcing high-level strategic decisions by configuring the parameters of Business Rules programs so that they behave in a certain prescribed way on average. In this paper we show that Business Rules are Turing-complete. As a consequence, we argue that there can exist no algorithm for configuring the average behaviour of all possible Business Rules programs.

1 Introduction

Business Rules (BR) are used in corporate environments to define business processes. Since not every employee is a computer programmer, the BR language is conceived as a "programming language for non-programmers". Typically, laypeople understand conditions and assignments much better than function calls and loops. Therefore, BR programs consist of sequences of conditional statements of the form

```
IF
      condition
THEN
      actions
```

where the *condition* is enforced on a vector of BR program variables $x = (x_1, \ldots, x_n)$, and the *actions* are assignments of the form

$$x \leftarrow f(x),$$

for some function f specified with the usual arithmetic operators and transcendental functions found in any standard mathematical library. Currently, there exist software packages called business rules management systems for creating, verifying, storing, retrieving, managing and executing BR programs, such as e.g. IBM's Operational Decision Management (ODM) [5], once known as ILOG's JRules.

A typical example is provided by a bank which defines the process of deciding whether to grant a loan to a customer. The conditions will verify anagraphical, work-related and credit ranking type information about the customer: since there may be dependency relationship in such information, it makes sense to break down this verification as a set of BRs, some of which might depend on the preceding ones. The action triggered by a BR might, through the assignment of a binary value to a Boolean variable or a scoring value to a scalar variable, switch on or off the activation of subsequent BRs. Finally, the output of this BR program will be an assignment of a binary value to a Boolean variable linked to whether the loan will be granted or not. The BRs will be initially written by employees on the basis of policies, regulations and personal work experience, and then modified to take the evolution of the business process into account.

Banks, however, come under political scrutiny, and they might be required to prove to the legislators that they grant a certain percentage of all demanded loans. Accordingly, banks ask their technical staff that their business processes, encoded as BR programs, be configured so as to behave as required on average. The average mentioned here refers to all possible requests to come; banks would probably interpret it over "average over all requests received so far", and their technical staff would likely pick an appropriate sample and run a supervised Machine Leaning (ML) algorithm to learn the BR program configuration parameter values on this sample used as a training set.

This begs the question: does there exist an algorithm for the task? More precisely, is there an algorithm such that, given a configurable BR program and a training set as input, would provide parameter values as output that conform to an accepted learning paradigm? This sounds like a philosophical rather than scientific question, but if we fix the learning paradigm to some concrete example thereof, such as *Probably Approximately Correct* (PAC) learning [10], then the question can be answered.

The objective of this paper is precisely to answer this question: and, unfortunately, it turns out that the answer is negative. We prove that the BR language is universal, meaning that interpreting BR programs in their natural computational environment is Turing-complete [7]. This means that any program can be written in the BR language, including, e.g. the implementation of certain types of pseudorandom functions. Such functions are known to be *PAC unlearnable* [3,6], which proves our result. Of course, our general negative result does not prevent the existence of learning algorithms which work on restricted classes of instances. Although such algorithms will not be discussed in this paper, some can be devised by means of mathematical programming [11].

2 How Business Rules work

As already mentioned, BR consist of IF ... THEN constructs. For simplicity, they do away with function calls and loops. On the other hand, this removal is only apparent, as BR replace function calls with *meta-variables* and loops with their execution environment.

Meta-variables and variables are stored in different symbol tables. Whereas the variables' symbol table simply matches variable symbols to variables, the meta-variables' symbol table matches meta-variable symbols to variable symbols. Every time a condition or an action in a BR refers to a meta-variable, the BR is instantiated for every variable symbol corresponding to the meta-variable, creating a matching number of IF ... THEN code fragments. This is akin to calling a function with a given parameter.

BR programs consist of a sequence of BRs. The execution environment is set up as an implicit loop which keeps scanning and executing rules until a termination condition is achieved, at which point the loop ends and the return variable – a Boolean in our example – is passed back to the calling process .Such a condition is usually that every rule condition is evaluated to FALSE.

2.1 Definitions and notation

A BR program consists in a set of type declarations and a set of rules. A type declaration consists of either the creation of a type (*create_new_type*(type)) or the assignment of a type to a variable (*type*(var) $\leftarrow$ type), where var can be either a variable or a meta-variable. A rule is defined as follows. Given α the typed meta-variables and x the typed variables, a rule is written:

if $T(\alpha, x)$ **then**
  $\alpha \leftarrow A(\alpha, x)$
  $x \leftarrow B(\alpha, x)$
end if

where T is the condition and the couple (A, B) describes the action. The action can be modifying the value of some fixed variable (function B) or the value of the variable matched to a meta-variable (function A), or both. In the first case, that action will be the same across all rule instances made from this rule; while in the second case the action will be different depending on the instance. For example, with variables x_1 and x_2 and meta-variable α_1, all of type int we might have the following rule:

if $\alpha_1 = 13$ **then**
  $\alpha_1 \leftarrow \alpha_1 + 3$
end if

in which case we have an action that has only A: $A(\alpha, x) = \alpha + 3$; $B(\alpha, x) = x$. The rule instance matching α_1 to x_1 would modify x_1, and the rule instance matching α_1 to x_2 would modify x_2. If the rule was instead:

if $\alpha_1 = 13$ **then**
  $x_1 \leftarrow x_1 + 2$

$x_2 \leftarrow 30$
end if

we would have an action with only B: $A(\alpha, x) = \alpha$; $B(\alpha, x) = (x_1 + 2, 30)$. The rule instance matching α_1 to x_1 and the rule instance matching α_1 to x_2 would execute the same action under different conditions. At least one of the variables, say x_1 without loss of generality, is selected to be the *output* of the BR program.

The first part of a BR program interpreter is to compile the rule instances derived from each rule. At compile time, α is replaced by every type-feasible reordering of the x input variable vector. For $x \in X \subseteq \mathbb{R}^n$, the explicit set of rule instances compiled from this rule is the type-feasible part of the following code fragments, using $(\sigma_j \mid j \in \{1, \ldots, n!\})$ the permutations of $\{1, \ldots, n\}$:

if $T((x_{\sigma_1(1)}, \ldots, x_{\sigma_1(n)}), (x_1, \ldots, x_n))$ **then**
$(x_{\sigma_1(1)}, \ldots, x_{\sigma_1(n)}) \leftarrow A((x_{\sigma_1(1)}, \ldots, x_{\sigma_1(n)}), (x_1, \ldots, x_n))$
$(x_1, \ldots, x_n) \leftarrow B((x_{\sigma_1(1)}, \ldots, x_{\sigma_1(n)}), (x_1, \ldots, x_n))$
end if
...
if $T((x_{\sigma_n(1)}, \ldots, x_{\sigma_n(n)}), (x_1, \ldots, x_n))$ **then**
$(x_{\sigma_n(1)}, \ldots, x_{\sigma_n(n)}) \leftarrow A((x_{\sigma_n(1)}, \ldots, x_{\sigma_n(n)}), (x_1, \ldots, x_n))$
$(x_1, \ldots, x_n) \leftarrow B((x_{\sigma_n(1)}, \ldots, x_{\sigma_n(n)}), (x_1, \ldots, x_n))$
end if

The size of this set varies. The typing of the variables and meta-variables matters, and depending on T, A and B some of these operations might also be computationally equivalent. The number of rule instances compiled from a given rule can be 0 for an invalid rule ($T(\alpha, x) = \mathsf{False}$), 1 for a static rule ($T(\alpha, x)$ and $B(\alpha, x)$ do not vary with α, and $A(\alpha, x) = \alpha$), and up to $n!$ for some rules if every variable has the same typing.

2.2 The BR interpreter

The "implicit loop" referred to above can be considered as a *BR interpreter*: it takes a BR program and turns it into a computational process which provides an output corresponding to a given input.

Any common interpreter for a BR program is algorithmically equivalent to creating all rule instances as a first step, and as a second step continuously verifying the given conditions, executing one assignment action corresponding to a condition which is True at each iteration of the loop, until either every condition evaluates to False or a user-defined termination condition becomes active. Although there exist many such interpreters, such as [2], for the purpose of this paper we only consider the most basic interpreter $\mathscr{I}_0$ which follows the simple algorithm below.

1. Match variables to type-appropriate meta-variables in rules to create all possible rule instances
2. Select the rule instances for which the condition is true, using the current values of the variables

3. Execute the action of the first rule instance in the current selection or stop if there is no such rule instance
4. Restart from step 2.

The order of rule instances used in Step 3 is defined by the order induced by a predefined order on the rules in the BR program and on the input variables.

We expect any more complicated ones to be able to simulate the most basic, so that BR executed using any interpreter are at least as powerful as what our study concludes. We also expect that $\mathscr{I}_0$ is able to simulate other interpreters, by using additional variables. This expectation comes from the idea that the difference with our interpreter will come from Step 3. The conflict resolution strategy, i.e. the strategy for selecting which rule among the ones obtained in Step 2 is to be executed, is what characterizes BR interpreters. Our algorithm has a simple conflict resolution strategy: whenever more than one rule instance could be executed, the choice is obtained from a given (fixed) total order on rule instances. An example of the execution of such an algorithm is described in Fig. 1.

The most common conflict resolution strategies combine one or more of the following three elements [2]:

- *Refraction* which prevents a rule instance from firing (being selected by the conflict resolution algorithm) again unless its condition clause has been reset.
- *Priority* which is a kind of partial order on rules, leading of course to a partial order on rule instances.
- *Recency* which orders rule instances in decreasing order of continued validity duration (when rule instances are created at run time, it is often expressed as increasing order of rule instance creation time).

These elements can be simulated by the basic interpreter by adding more types, variables or rules, in broad strokes:

- *Refraction* results in the use of an additional boolean variable $needsReset$ per rule per variable permutation (ie per rule instance), an additional test clause in each rule and an additional rule (**if** $needsReset_{x,r} = true \wedge T_r(\alpha, x) = false$ **then** $needsReset_{x,r} = false$) per rule r.
- *Priority* results in one additional integer variable p, an additional test clause in each rule ($p = \pi_r$), an additional action clause in each rule ($p \leftarrow p_{\max}$) , and two additional rules that come dead last in the predefined order on rules (**if** $p > 0$ **then** $p = p - 1$; **if** $p = 0$ **then** Stop).
- *Recency* is the most complicated. A possible simulation could involve an additional integer variable $validityStart$ per rule instance, an additional integer variable $timer$, an additional action clause in each rule ($timer \leftarrow timer + 1$) and a similar setup to the one suggested for *priority*, using an integer variable $priorValidity$ which would this time start at 0 and end at $timer$.

The Rules (in order)
R_1 :
if $(\alpha_1 \geq 1 \wedge \alpha_2 = 2)$
then $(\alpha_1 \leftarrow 0, \alpha_2 \leftarrow 0)$
R_2 :
if $(\alpha_3 = 1)$
then $(\alpha_3 \leftarrow \alpha_3 + 1)$

The typed Variables (in order)
int x $\leftarrow 1$
int age $\leftarrow 90$

The typed Meta-variables
int α_1
int α_2
int α_3

The Rule Instances
In the total order considered by the algorithm, they are:
r_1 :
if (x $\geq 1 \wedge$ age $= 2$)
then (x $\leftarrow 0$,age $\leftarrow 0$)
r_1' :
if (age $\geq 1 \wedge$ x $= 2$)
then (age $\leftarrow 0$,x $\leftarrow 0$)
r_2 :
if (x $= 1$)
then (x $\leftarrow$x+1)
r_2' :
if (age $= 1$)
then (age $\leftarrow$age+1)

The Execution
Iteration 1:
Truth value of conditions:
$t(r_1) =$ False
$t(r_1') =$ False
$t(r_2) =$ True
$t(r_2') =$ False
Rule instances selected (in order): r_2
Rule executed: r_2
Variable values:
x $= 2$
age $= 90$

Iteration 2:
$t(r_1) =$ False
$t(r_1') =$ True
$t(r_2) =$ False
$t(r_2') =$ False
Rules selected: r_1'
Rule executed: r_1'
Variable values:
x $= 0$
age $= 0$

Iteration 3:
$t(r_1) =$ False
$t(r_1') =$ False
$t(r_2) =$ False
$t(r_2') =$ False
Rules selected: None
Rule executed: None

END

Fig. 1. Example illustrating the execution algorithm

3 Turing completeness of business rules

Since a BR program is executed in a loop construct as discussed in Sect. 2.2, which only terminates when all conditions evaluate to False, a fundamental question is whether we can decide if the execution terminates at all. Asking the same question of any programming language amounts to asking whether the HALTING PROBLEM (HALT) can be solved on the class of Turing Machines (TM) that the programming language is able to describe. Since it is well known that HALT cannot be solved for Universal TMs (UTMs), the question is whether BRs can describe a UTM. In this section prove that this is indeed the case. In other words, we prove that BRs are Turing complete.

For our original question, i.e. whether there is an algorithm $\mathcal{A}$ for arbitrarily changing a BR program so it statistically behaves according to a given target, Turing completeness shows that we cannot hope to ever find an algorithm $\mathcal{A}$ which works on *all* BR programs. At the very least, we shall have to limit our attention to all *terminating* BR programs.

Our proof of Turing completeness for BR programs consists in spelling out a BR program $\mathcal{U}$ which takes as input any TM description with its own input. When executed $\mathcal{U}$, simulates the TM acting on its input.

A UTM is a TM which can simulate any other TM on arbitrary input [8,9]. It does that by taking as input a description T of any TM as well as its input x. We use the usual definition of a Turing Machine [4]. We note the states of a TM $q_1, \ldots, q_Q$, its tape symbols $s_1, \ldots, s_S$, its blank symbol s_b, its transition function $(q^i, s^i) \rightarrow (q^f, s^f, \mathsf{act})$ where $\mathsf{act} \in \{$"left", "right", "stay"$\}$, its initial state q^0, and its accepting states $T_e r$. An initial tape T_0 is said to be accepted by a TM if the TM reading this tape stops and has a final state in $T_e r$.

A BR program which simulates a UTM by being able to simulate any TM is described in Fig. 2. It uses the same notations, with initial values of $q = q^0$; of T a truncated T^0 containing a finite number of symbols, containing T_0^0 and containing all non-blank symbols of T^0; of $l = \text{size}(T)$; and of $p = 0$.

We suppose the variables include the following:
- many (static) state objects of type "state": $q_1, \ldots, q_Q$
- many (static) symbol objects of type "symbol": $s_1, \ldots, s_S$
- a (static) finite set of terminal states of type "terminal": T_{er}
- a (static) blank symbol of type "symbol": s_b
- a (static) set of Turing rules of type "rules", of the form $(\text{state}_{\text{initial}}, \text{symbol}_{\text{initial}}, \text{right}|\text{left}|\text{stay}, \text{state}_{\text{next}}, \text{symbol}_{\text{written}})$:
  $\mathcal{R} = \{(q_r^i, s_r^i, \mathsf{act}_r, q_r^f, s_r^f) \mid \mathsf{act}_r \in \{\text{"left"}, \text{"right"}, \text{"stay"}\}\}_r$
- the current state of type "state": q
- the length of the visible tape data, of type "length": l
- the current visible tape data of type "tape": $T = \{(i, s_i) \mid i \in \mathbb{N}, 0 \leq i \leq l-1\}$ where l is the length of the visible tape data
- the current place on the tape of type "position": p

We use the following meta-variables in the BR program that simulates a UTM:
- α_{q^f} of type "state"
- α_{s^f} of type "symbol"

The rule set to simulate a UTM is then written in a compact form:

R_1:
if
  $(q, T(p), \mathsf{act}, \alpha_{q^f}, \alpha_{s^f}) \in \mathcal{R}$
then
  $q \leftarrow \alpha_{q^f}$
  $T \leftarrow (T \setminus \{(p, T(p))\}) \cup \{(p, \alpha_{s^f})\}$
  $p \leftarrow p \pm 1$(Depending on the value of act)
  $l \leftarrow l \pm 1$(Depending on the respective values of act, p and l)

R_2:
if
  $(q \in T_{er})$
then
  Stop;

Fig. 2. A BR program which describes a UTM.

Some simplifications have been made for the sake of clarity: R_1 should clearly be at least three different rules each replacing act with one of "left", "right", "stay". The complete formally correct form would in fact have two more rules, in order to increase the length of the tape as needed, using the variable s_b.

Theorem 1. *the BR program described in Fig.2 simulates any TM given an accepted tape. The final value of the tape and the final symbol of the TM will be identical to the final values of T and q.*

Proof. We prove that the simulation is correct by induction over the number of steps n taken by the TM.

Before the TM takes any step ($n = 0$), its tape is identical to the value of T in the BR program before it executes any rule, as that tape is given to the BR as an input value. Similarly, the place on the tape at that point is the value of p and the state of the TM is equal to the value of q.

Assume that the tape, the position on it, and the state of the TM after step n are accurately represented by the BR program after n rule executions. We call T^{TM} the sequence representing the tape, p^{TM} the current place on the tape, and q^{TM} the current state of the TM.

If the TM halts, that means $\forall(\mathsf{act}, q^f, s^f), (q^{TM}, T^{TM}_{p^{TM}}, \mathsf{act}, q^f, s^f) \notin \mathcal{R}$. Thus, the BR does not execute R_1. Further, as the initial tape is accepted by the TM, we have $q^{TM} \in T_e r$, which fulfills the condition for R_2. The BR program terminates at the same time as the TM, and its output is correct as R_2 does not modify values.

Otherwise, the TM will follow a rule in $\mathcal{R}$. Let us call it

$$r = (q^{TM}, T^{TM}_{p^{TM}}, \mathsf{act}, q^f, s^f).$$

In this case, the next BR executed will be R_1, and the only member of $\mathcal{R}$ to match will be r. In other words, the only relevant rule instance is:

R_1:

if	$(q, T(p), \mathsf{act}, q^f, s^f) \in \mathcal{R}$
then	$q \leftarrow q^f$
	$T \leftarrow (T \setminus \{(p, T(p))\}) \cup \{(p, s^f)\}$
	$p \leftarrow p \pm 1$(Depending on the value of act)
	$l \leftarrow l \pm 1$(Depending on the respective values of act, p and l)

because $q = q^{TM}$ and $T(p) = T^{TM}_{p^{TM}}$. As the action on this BR instance corresponds exactly to the modifications to the state and tape of the TM, the values stored on the tape of the TM after $n + 1$ steps will again be the same as the values in the BR program. □

4 Unlearnability

Let us go back once more to the original question: is there an algorithm for arbitrarily changing a BR program so it statistically behaves according to a

given target? By Sect. 3, we know that this algorithm cannot exist in the most general terms, since BR programs might not even terminate in finite time. But what if we just look at those BR programs which *do* terminate? The question can be considered as a learnability problem: Does there exist an algorithm $\mathcal{A}$ which taking a class $\mathscr{P}_p$ of terminating BR programs parametrized over p, a data distribution $\mathcal{D}$ over its input domain X and a goal g for the value of the average output $\mathbb{E}_{\mathcal{D}}(\mathscr{P}_p)$, efficiently and weakly learns p? In other words, is p learnable in this context?

We limit ourselves to the well-studied and powerful family of algorithms known as Probably Approximately Correct (PAC) learning algorithms introduced in [10]. In this section we use a class of pseudorandom functions to provide a negative answer to the question, when posed in these very general terms. Of course, it may still be possible to achieve our stated purpose for less general, but still useful classes of BR programs.

4.1 Background

Pseudorandom functions (PRF), introduced by Goldreich, Goldwasser and Micali ([3]), are indexed families of functions F_p for which there exists a polynomial-time algorithm to evaluate $F_p(x)$, but no probabilistic polynomial-time algorithm can distinguish the function from a truly random function F_{rand} without knowing p, even if allowed access to an oracle.

A PAC learning algorithm identifies a concept (i.e. a function $X \to \{0,1\}$) among a concept class $\mathscr{C}$ (i.e. a family of concepts). For a concept $f \in \mathscr{C}$ and a list S of data points in X of length λ, an algorithm $\mathcal{A}$ is an (ϵ, δ)-PAC learning algorithm for $\mathscr{C}$ if for all sufficiently large λ:

$$\mathbb{P}[\mathcal{A}(f) = h \mid h \text{ is an } \epsilon\text{-approximation to } f] \geq 1 - \delta$$

where $\mathcal{A}$ has access to an oracle for f.

- $\mathcal{A}$ is said to be efficient if the time complexity of $\mathcal{A}$ and h are polynomial in $1/\epsilon$; $1/\delta$; and λ.
- $\mathcal{A}$ is said to weakly learn $\mathscr{C}$ if there exist some polynomials $p_\epsilon(\lambda)$; $p_\delta(\lambda)$ for which $\epsilon \leq \frac{1}{2} - \frac{1}{p_\epsilon(\lambda)}$ and $\delta \leq 1 - \frac{1}{p_\delta(\lambda)}$.
- We say a concept class is PAC learnable if it is both efficiently and weakly learnable. Otherwise, it is unlearnable.

It is known that PRF are unlearnable using PAC algorithms ([3,6]). In the rest of this section, we consider F_p such a PRF, and note $\text{Eval}_{p,x}(F_p(x))$ the complexity of evaluating $F_p(x)$.

4.2 Unlearnability Result

We call $(\mathscr{P}_p)_{p\in\pi}$ a class of terminating BR programs indexed by p, S a list of items from the input domain X with $|S| = \lambda$, and g a goal for the value of the average output $\mathbb{E}_S(\mathscr{P}_p)$. We consider C the concept class whose members are $f : (\mathscr{P}_p)_{p\in\pi} \to \{0,1\}$.

Theorem 2. *The concept class C is unlearnable: specifically, the concept $h \in C$ defined as $h(p) = 1$ iff $\mathbb{E}_S(\mathscr{P}_p) = g$ cannot be learned using a PAC learning algorithm in the general case.*

In other words, there is no practically viable algorithm that can learn a BR program out of a class of BR programs in the general case, even with access to a perfect oracle. This is a consequence of both the Turing-completeness of BR programs and the unlearnability of PRF.

Proof. As BR programs are Turing-complete, we choose the family $(\mathscr{P}_p)_{p\in\pi}$ to be a PRF. Any algorithm that leans C also learns $(\mathbf{1}_f(p))_p \subset C$, where $\mathbf{1}_f(p) = 1$ iff $\mathscr{P}_p = f$. Learning the latter is trivially the same as learning a PRF, which is proven to be impossible. □

The specific example mentioned in the theorem answers our original question. Even if the concept we wish to learn is described more broadly than by providing an oracle for a specific BR program, it is impossible to adjust the statistical behavior of BR programs according to a predefined goal.

4.3 Complete Unlearnability

We have used the fact that PRFs are not PAC learnable in the sense that no PAC algorithm can efficiently and weakly learn a PRF. We now demonstrate an example of a concept class that cannot be learned by PAC algorithms at all. This example is based on the intuition that chaos cannot be predicted, and so cannot be learned.

We use a known chaotic map, the logistic map $f_{n+1}(x) = af_n(x)(1 - f_n(x))$, $f_0(x) = x$, with the parameter $a = 4$. Some of its properties are presented by Berliner [1]. We call $C_n(x)$ the concept class such that $C_n(x) = 1$ iff $f_n(x) \geq 0.5$ and $C_x(n) = 0$ otherwise, where $x \in [0, 1]$ follows the arcsine distribution, i.e. the probability density function is $p(x) = \frac{1}{\pi\sqrt{x(1-x)}}$, and with $n \in \mathbb{N}$ following the uniform distribution.

Theorem 3. *The concept class $C_n(x)$ cannot be learned with any accuracy. To be precise, for all algorithms $\mathcal{A}$ calling the oracle $C_n(x)$ a finite number of times, we have:*

$$\mathbb{P}_{n\in\mathbb{N}}(\mathbb{P}_{x\in X}(\mathcal{A}(C_n)(x) \neq C_n(x)) = 0.5) = 1$$

Proof. The proof relies heavily on Berliner's paper [1]. From it, we know that as the logistic map is chaotic, each sequence $(f_n(x))_n$ is either eventually periodic or is dense in $[0, 1]$. We also know that as X follows the arcsine distribution, the $C_n(X)$ are i.i.d. Bernoulli random variables, such that $\mathbb{P}_{x\in X}(C_n(x) = 1) = 0.5$.

Suppose $\mathcal{A}$ calls $C_n(x)$ for values of $x \in \{x_1, \ldots, x_k\}$. We call n^0 the value such that $\mathcal{A}(C_n) = C_{n^0}$. As $\mathbb{P}_{x\in X}(C_{n^0}(x) = 1) = 0.5$ does not depend on n^0, and the $C_n(X)$ are i.i.d., we have $\mathbb{P}_{x\in X}(C_{n^0}(x) \neq C_n(x)) = 0.5$ iff $n^0 \neq n$ and $\mathbb{P}_{x\in X}(C_{n^0}(x) \neq C_n(x)) = 0$ otherwise. The theorem is thus the same as saying that $\mathcal{A}$ almost certainly (in the probabilistic sense) cannot match n^0 to

the exact value of n. We now prove that there almost always exists $n^1 \neq n$ which is indistinguishable from n by $\mathcal{A}$, i.e. $C_{n^1}(x_1) = C_n(x_1), \ldots, C_{n^1}(x_k) = C_n(x_k)$.

Let us call $Y_i^1 = C_i(x_1), \ldots, Y_i^k = C_i(x_l)$ with $i \in \mathbb{N}$. Some of the sequences Y^j are periodic after some rank, and some are not. Without loss of generality, we assume $Y^1, \ldots, Y^{k_1}$ are periodic, and $Y^{k_1+1}, \ldots, Y^k$ are not. Almost certainly, $(Y^1)_{i \geq n}, \ldots, (Y^{k_1})_{i \geq n}$ are periodic (n is big enough). Using $P \in \mathbb{N}$ to denote the smaller common multiple of those sequences' periods, we notice that $C_{n+Pi}(x^1) = C_n(x^1), \ldots, C_{n+Pi}(x^{k_1}) = C_n(x^{k_1})$. We note $y_i = n + Pi$.

As the sequences $Y^{k_1+1}, \ldots, Y^k$ are not eventually periodic, we know that each sequence $(f_n(x^{k_1+1}))_n, \ldots, (f_n(x^k))_n$ is dense in $[0, 1]$. Consequently, for any sequence of $k - k_1$ bits, there exists a countable number of $n^1 \in (Y_i)_{i \in \mathbb{N}}$ such that it is equal to $Y^{k_1+1}, \ldots, Y^k$. In particular, if this sequence is $C_n(x^{k_1+1}), \ldots, C_n(x^k)$, any of those n^1 different from n proves the theorem. □

It must be noted that no practical application would ever try to learn this type of concept class. A key part of the proof is allowing the concept class to be infinite and indexed by a natural number, without bounding that index at all. This is unlikely to happen for computational reasons, the usual way to represent a natural number being with integer or long typed variables. Another difficulty is representing and computing real numbers, which can be done using Real RAM machines, to compute $f_n(x^1), \ldots, f_n(x^l)$. In the case of the logistic map with parameter $a = 4$, the task is made slightly easier by the existence of an exact solution, but other chaotic maps would require expensive recursive computations.

The existence of such extreme cases of unlearnability is nevertheless something to be careful of. It must be noted that none of the aforementioned computational difficulties are impossibilities, and that such unlearnable concepts are thus possible problems for BR programs, among other Turing-Complete programming languages.

5 Conclusion

Business Rules seem simple enough, repeatedly treating data according to a simple algorithm. The complexity of BR programs actually comes from the interpreters. In particular, almost any interpreter that uses a looping algorithm can make BR programs Turing complete, as is the case with the simplistic algorithm we have presented in this paper. The proof of such is simple, yet it is a result that has been overlooked so far (to the best of our knowledge). The Turing completeness of BR programs can have important theoretical implications: it links the usual Rules research on Inference Rules and ontologies with more traditional research on programming languages and computability.

The second part of our paper studies shows another theoretical implication of this result. As a Turing-complete language, no PAC algorithm can adjust the parameters in a BR program for a specific statistical behavior, even with a perfect oracle. This impossibility has practical implications. In particular, BR programs

are often used to model business processes, which companies might want to optimize over an average output. That there exists no such algorithm in the general case means that algorithms working on a specific class of BR programs are their only way of automatically modifying a BR program with such an aim. Furthermore, we demonstrate that learning at all is not always possible, and so general learning algorithms for Turing-Complete programming languages cannot exist as such. Learning heuristics are thus not only computationally efficient, but necessary for trying to treat the full generality of the learning problem.

Acknowledgments

The first author (OW) is supported by an IBM France/ANRT CIFRE Ph.D. thesis award.

References

1. L. Berliner. Statistics, probability and chaos. *Statistical Science*, 7(1):69–90, 1992.
2. C. de Sainte Marie, G. Hallmark, and A. Paschke. Rif production rule dialect (second edition). W3C Recommendation, 2013.
3. O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, 1986.
4. J. Hopcroft and J. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, Reading, MA, 1979.
5. IBM Corporation. *Operational Decision Manager 8.8*, 2015.
6. M. Kearns and L. Valiant. Cryptographic limitations on learning boolean formulae and finite automata. *Journal of the ACM*, 41(1):67–95, 1994.
7. L. Liberti and F. Marinelli. Mathematical programming: Turing completeness and applications to software analysis. *Journal of Combinatorial Optimization*, 28(1):82–104, 2014.
8. C. Shannon. A universal Turing machine with two internal states. In C. Shannon and J. McCarthy, editors, *Automata Studies*, volume 34 of *Annals of Mathematics Studies*, pages 157–165, Princeton, 1956. Princeton University Press.
9. A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(1):230–265, 1937.
10. L. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.
11. O. Wang, C. Ke, L. Liberti, and C. de Sainte Marie. Controlling the average behaviour of business rules programs. Technical report, Ecole Polytechnique and IBM France, 2016.