Diplomarbeit

Confident Iterative Learning in Computational Learning Theory

Ausgeführt am Institut für

Diskrete Mathematik und Geometrie
der Technischen Universität Wien

unter der Anleitung von

Univ.Ass. Dipl.-Ing. Mag. Dr.techn. Ekaterina Fokina

durch

Vanja Doskoč

Wien, am 23.8.2017

Acknowledgements

Here, I would like to take the time to spread a few words of gratitude. Generally, these words are addressed to all the people who encouraged and supported me during my studies. However, I do wish to express a few particularly addressed words, too.

The first of these are certainly directed towards Professor Ekaterina Fokina. Not only did she introduce me towards the field of algorithmic learning theory, but she also provided me with much useful information, not only concerning this thesis. She also proved to be a very patient, humorous and friendly supervisor.
Also, she enabled me to have a first glimpse into the daily work of a researcher, by granting me the chance do some research on my own.
I would like to thank Professor Frank Stephan as well, for suggesting the topic of this thesis and for providing some useful facts and helpful remarks.

I would also like to thank my mother for granting me the opportunity of studying. Also, one big kiss of gratitude is directed towards my girlfriend. I am really thankful for both of their never ending love and support.
A last, but certainly not least, act of gratitude is directed towards my friends and my family. By supporting me, they truly influenced that experience of studying of mine.

Thank you.

Abstract

Let us assume that we have a class of sets. Now, if we have a machine that, fed information on one of these sets, tells us which set the information belonged to, then the machine underwent some sort of learning. This act of learning can happen in various forms. The aim of this thesis is to motivate, introduce and investigate some possible ways of learning. Firstly, we will motivate the basic ideas of computability theory and algorithmic learning theory. Concerning the latter theory, we will get to know some widely used learning types, the most prominent being the explanatory and behaviourally correct learning.
The main aim, however, is to investigate a new type of learning, the confident iterative learning. The idea here is to merge two known concepts, namely that of the confident and iterative learner. Additionally to learning the sets of the class correctly, the first learner is required to make some, not necessarily true, guess on any other set, too. Instead of having all the information of the set at hand at every time, the second learner may only use its last hypothesis as memory on the previous calculations and information. So, we restrict its memory.
Observing it, we will provide some negative as well as positive examples. We will also prove some properties the confident iterative learner possesses. This will peak at the classification theorem, where we provide a classification for certain types of classes.
As last act, we will consider an even more advanced idea, namely that of the very and strongly confident learner. Here, we will focus on the behaviour on sets not belonging to the class. We will try to detect them, in one form or another.
Lastly, we will focus on the possible hypotheses. We will investigate the behaviour of the confident iterative learning when choosing special kinds of hypothesis spaces.

Contents

1 Overview

1.1 Motivation

Most living creatures have experienced some form of learning at one point or another. With the computer revolution at the last midcentury, and the improvements ever since, it only seems natural to test whether one can teach a computer to learn and, if so, ask to what extend this is possible.
Given a class and a set in it, learning is considered the process of identifying the set while getting more and more information, i.e. elements, of that set. Since the initialization[1] of this idea, learning has been very intensively studied. With time, more and more ideas emerged. Most of them rely on some natural process of learning, i.e. the process of learning a language.
For example, adding memory bounds, which can be considered as a pretty natural bound, or learning, even if confronted with some unrelated information, lead to the implementation of the iterative and confident learner, respectively.
In this thesis, we will contribute to the efforts done so far, by investigating the combination of the two mentioned learning paradigms, leading to the confident iterative learner.

1.2 Preview

In order to do so, in the **second chapter** we will fix some notations as well as swipe through the basic concepts of computability theory and recapitulate the, for our purpose, relevant results.

In the **third chapter** we will introduce the basic concepts of algorithmic learning theory. To get to know the materia, we will investigate two prominent concepts of learning types, namely the explanatory and behaviourally correct one. Also, to get a better feeling for the materia, we will consider some alteration of these. Comparing those will provide us a better understanding. Altogether, this chapter will serve as an introduction into algorithmic learning theory.

In the **fourth chapter** we will introduce the object of desire, the confident iterative learner. After introducing the idea, we will investigate what kind of classes can or cannot be learnt using a confident iterative learner. While doing so, we will also provide some properties. This will culminate in the classification theorem of special types of classes.

[1] One of the first, see [1], [6] or [16], formalizations can be found in [7].

As an excursion into some further topics, in the **fifth chapter** we will refine the idea of the confident learner. We will try to detect sets not belonging to the class. In order to do so, we will introduce two, rather powerful, ways, and then combine these with the iterative learner. This will lead us to some notable results.
Lastly, we will focus our attention on the hypotheses themselves. We will restrict the possible hypotheses, thus receive a hypothesis space. Learning classes with respect to some hypothesis space, obviously, depends on the chosen hypothesis space. Considering the confident iterative learner once again, we will see three different types of learning here.
At the very end, we will also propose some rough ideas on what the future research concerning this topic may consist of.

2 Introduction

Based on the course notes [5], but also locatable in [4], in this chapter we are going to fix some notations and recapitulate some important results. Most of them will remain without proof here, as we assume them to be well known. For some more details, readers are forwarded to the mentioned sources.

2.1 Notations

The start marks an enhanced notation of quantifiers.

Notation 2.1.1: Let $A(x)$ be a statement.
Then, by writing $\forall_y^\infty x : A(x)$ we mean $\exists y\ \forall x \geq y : A(x)$. If we do not need to emphasize y, we will write $\forall^\infty x : A(x)$.
Also, if we write $\exists^\infty x : A(x)$, we mean that $\forall y\ \exists x \geq y : A(x)$.

As we will be handling functions, let us fix some notations for them, too.

Notation 2.1.2: Consider a function $f\colon I \subseteq \omega^{n_f} \to \omega$. Here, n_f is the *arity* of f, and if it is clear from the context, we will use n.
Given a tuple $\overline{x} = (x_1, \ldots, x_n)$, we write $f(\overline{x}) \downarrow = y$ if f is defined on $\overline{x}$ and equal to y. We write $f(\overline{x}) \uparrow$ if it is not defined on $\overline{x}$.

Also, the equality of functions will be of some importance.

Notation 2.1.3: Let f, g be two functions, then we write $f(x_1, \ldots, x_{n_f}) = g(y_1, \ldots, y_{n_g})$ iff either both values are simultaneously defined and equal, or both are not defined.

Later on, we will need an extension of the natural numbers. So, let us enlarge them with a biggest element.

Notation 2.1.4: We denote $\mathbb{N}_* := \mathbb{N} \cup \{*\}$. The ordering $(\mathbb{N}_*, \leq_{\mathbb{N}_*})$ is defined as

$$\forall a, b \in \mathbb{N} : a \leq_{\mathbb{N}_*} b \Leftrightarrow a \leq_{\mathbb{N}} b,$$
$$\forall a \in \mathbb{N}_* : a \leq_{\mathbb{N}_*} *.$$

As we are speaking of a rather natural extension here, we will write $\leq = \leq_{\mathbb{N}_*}$.

Basically, the $*$ will stand for "finitely many". With this at our hands, we can introduce the term "almost equal".

Notation 2.1.5: Let $a \in \mathbb{N}$. Two functions f and g are said to be equivalent on all but

a many arguments, written $f =^a g$, if

$$\exists y_1, \dots y_a \; \forall x, x \notin \{y_1, \dots y_a\} : f(x) = g(x).$$

Two functions f and g are said to be equivalent on all but finitely many arguments, written $f =^* g$, if

$$\exists n \in \mathbb{N} : f(x) =^n g(x).$$

Remark 2.1.6: For any function $f(x)$ and any $a \in \mathbb{N}_*$, we have $f =^a f$.

Also, we would like to remind the reader of the following ordering on sequences.

Notation 2.1.7: Let σ, τ be sequences, where $|\sigma| \leq |\tau|$. We write $\sigma \preceq \tau$ if

$$\forall n \leq |\sigma| : \sigma(n) = \tau(n).$$

We call τ an *extension* of σ, and σ an *initial segment* of τ.

2.2 Computable Functions

We will formalize the idea of a function being computable, or effective. One possible approach is the algebraic one. Alongside the basic functions, i.e. $0, o(x) = 0, s(x) = x+1$ and $I^n_m(x_1, \dots, x_n) = x_m, 1 \leq m \leq n, n \in \mathbb{N}$, we have the following operations at our disposal.

- f is the *composition* of $g_1, \dots, g_m, h$ if

$$f(\overline{x}) = h(g_1(\overline{x}), \dots, g_m(\overline{x})).$$

- f is the result of *primitive recursion* of g and h if

$$f(\overline{x}, y) = \begin{cases} g(\overline{x}), & \text{if } y = 0, \\ h(\overline{x}, y-1, f(\overline{x}, y-1)), & \text{else.} \end{cases}$$

- f is the result of *minimization* of g, written $f(\overline{x}) = \mu y[g(\overline{x}, y) = 0]$, if

$$f(\overline{x}) = \begin{cases} y, & g(\overline{x}, 0), \dots, g(\overline{x}, y-1) \downarrow \neq 0 \text{ and } g(\overline{x}, y) \downarrow = 0, \\ \text{not defined}, & \text{else.} \end{cases}$$

Remark 2.2.1: Since we are formalizing computer programs here, it makes sense to compare the operations to their computer counterpart. Composition equals the composition of programs, primitive recursion serves as loop. Lastly, minimization formalizes the idea of an effective search.

With these ideas, we are able to define partial recursive functions.

Definition 2.2.2: A function f is a *partial recursive function* if there is a sequence $f_1, \ldots, f_m$, such that $f_m = f$ and each f_i is either one of the basic functions, or it is obtained by applying composition, primitive recursion or minimization to some of the previous functions, i.e. functions with index $k < i$.

Remark 2.2.3: An everywhere defined, partial recursive function will be called just *recursive function*. If we want to emphasize this property, we will use the term total recursive function.

We provide some examples, as we will need them later on.

Example 2.2.4: The following functions are recursive.

1. $f_1(x) = \overline{\text{sgn}}(x) = \begin{cases} 1, & x = 0, \\ 0, & x \neq 0. \end{cases}$
2. $f_2(x, y) = |x - y|$,
3. $f_3(x, y) = \exp(x, y) = z$, where z is the exponent of the x-th prime number in the prime decomposition of y. We start the effective enumeration of the prime numbers with $p_0 = 2$.

This approach is equivalent to others, like the approach via Turing machines, see [4] or [5]. Altogether, we will believe the following thesis.

Thesis 2.2.5 (Church-Turing Thesis)**:** *The class of all intuitively computable functions coincides with the class of all partial recursive functions.*

In the shine of the Church-Turing Thesis, we will use the term partial computable function from now on.

Notation 2.2.6: We will denote by REC the set of all total, i.e. everywhere defined, computable functions, and PREC all partial computable functions.

Remark 2.2.7: Later, we will consider sets. A set $A \subseteq \omega$ is called *computable* iff its characteristic function χ_A is computable. A set A is called *computably enumerable*, or c.e., iff it is the range or the domain of some partial computable function.

2.3 S_n^m and Recursion Theorem

Here we will provide several classical theorems of computability theory that will be useful for our further considerations. First, we will introduce the following notation.

Notation 2.3.1: With $\varphi_0, \varphi_1, \ldots$ we will denote an acceptable numbering of all partial computable functions. With $W_e = \text{dom}(\varphi_e)$ we will number the c.e. sets. We will refer to such an index e as *Kleene index*.

Notation 2.3.2: Additionally, we introduce the truncated computation $\varphi_{e,s}(x) = y$, also written $(\varphi_e(x))_s = y$. This means that for input x the program e outputs y in no more than s computational steps. Again, we will write $W_{e,s} = \text{dom}(\varphi_{e,s})$.

Example 2.3.3: One example using this numbering is the *halting set* $K = \{x : \varphi_x(x) \downarrow\}$. Recall that it is c.e., but not computable.

Now, we will provide some useful facts concerning that numbering.

Theorem 2.3.4 (Normal Form Theorem)**:** *For every k-ary partial computable f exists $e \in \omega$ such that $f(\overline{x}) = \varphi_e(\overline{x})$.*

In order to deal with those indices, we will need the following two theorems. The first one allows us to "reinterpret" programs and inputs.

Theorem 2.3.5 (S_n^m Theorem)**:** *For every $m, n \geq 1$ there exists a computable function s such that for all $x, y_1, \ldots, y_m$, and all inputs $z_1, \ldots, z_n$*

$$\varphi_x(y_1, \ldots, y_m, z_1, \ldots, z_n) = \varphi_{s(x,y_1,\ldots,y_m)}(z_1, \ldots, z_n).$$

Next comes the Fixed Point Theorem alias Recursion Theorem.

Theorem 2.3.6 (Recursion Theorem)**:** *Let f be a computable function. Then there exists $n \in \omega$ such that $\varphi_n(x) = \varphi_{f(n)}(x)$.*

Additionally, we will need the following corollaries, the latter of them will also be proven.

Corollary 2.3.7: *For every computable $f : \omega^{k+1} \to \omega$ exists some e such that*

$$f(e, \overline{x}) = \varphi_e(\overline{x}).$$

Corollary 2.3.8: *For every $e \in \mathbb{N}$ exists a computable $g \colon \mathbb{N} \to \mathbb{N}$ such that*

$$\varphi_{g(e)}(\overline{y}) = \varphi_e(g(e), \overline{y}).$$

Corollary 2.3.9: *Let f and g be partial computable functions with Kleene indices e_f and e_g, respectively. Then, $h = g \circ f$ is partial computable and we can effectively compute its index using e_f and e_g, i.e. there exists some computable s such that $\varphi_{s(e_g,e_f)} = h$.*

Proof. We will prove the unary case, only. Obviously, h is partial computable.
Define $F(x, y, z) := \varphi_x(\varphi_y(z))$. As composition of partial computable functions, F is partial computable itself. Thus, there exists some e_F such that $F(x, y, z) = \varphi_{e_F}(x, y, z)$. By S_n^m Theorem 2.3.5 there also exists some computable $\tilde{s}$ such that $\varphi_{\tilde{s}(e_F,x,y)}(z) = \varphi_{e_F}(x, y, z)$. Let $s(x, y) = \tilde{s}(e_F, x, y)$. Then, s is computable, too, and

$$\varphi_{s(x,y)}(z) = \varphi_{\tilde{s}(e_F,x,y)}(z) = \varphi_{e_F}(x, y, z) = F(x, y, z).$$

Letting $x = e_g$ and $y = e_f$, we get

$$\varphi_{s(e_g,e_f)}(z) = F(e_g, e_f, z) = \varphi_{e_g}(\varphi_{e_f}(z)) = g(f(z)) = h(z).$$

So, we can compute the index of h by $s(e_g, e_f)$. □

So, when given two partial computable functions, we can also compute the index of their composition.

2.4 m-Reducibility and Arithmetical Hierarchy

In this section, we will introduce a way of comparing the complexity of sets, namely m-reducibility.

Definition 2.4.1: A set A is *m-reducible* to B, written $A \leq_m B$, if there exists a computable f such that for all $x \in \omega$

$$x \in A \Leftrightarrow f(x) \in B.$$

Notice, that $\leq_m$ is reflexive and transitive.
Next, in order to define the jump, we will enhance the idea of computable functions. We will provide them with some oracle, or black box, which provides them some information. For more detail, consider the following definition.

Definition 2.4.2: A partial function f is partial computable relative an oracle B, or partial B-computable, if f can be computed as in Definition 2.2.2, where the basic functions are extended by the characteristic function of B.

Similar as in previous sections, we can get some enumeration of all partial computable functions or sets relative to some oracle A, i.e. φ_e^A and W_e^A, respectively.
Let $\langle .,. \rangle \colon \mathbb{N}^2 \overset{1-1}{\underset{\text{onto}}{\to}} \mathbb{N}$ be some computable pairing function with computable inverse functions. Let $A' := \{\langle x, y\rangle : x \in W_y^A\}$, and $A^{(n+1)} = (A^{(n)})'$ for $n \in \mathbb{N}$, be the *n-th jump* of $A \subseteq \omega$. We can compare jumps using m-reducibility.

Lemma 2.4.3: *Let $n \in \mathbb{N}$ and $A \subseteq \omega$. Then $A^{(n)} \leq_m A^{(n+1)}$ and $A^{(n+1)} \not\leq_m A^{(n)}$.*

One smooth way to compare the complexity of certain sets to jumps of the halting set is via the arithmetical hierarchy.

Definition 2.4.4 (Arithmetical Hierarchy)**:** Recursively define for $n \geq 0$,

1. $\Sigma_0^0 = \Pi_0^0$ is the set of all computable sets or relations.
2. Σ_{n+1}^0 is the set of all relations of the form $\exists \overline{y} R(\overline{x}, \overline{y})$, where $R \in \Pi_n^0$.
3. Π_{n+1}^0 is the set of all relations of the form $\forall \overline{y} R(\overline{x}, \overline{y})$, where $R \in \Sigma_n^0$.

We obtain the following theorem.

Theorem 2.4.5: *For any $A \subseteq \omega$*

$$A \in \Sigma_{n+1}^0 \Leftrightarrow A \leq_m K^{(n)}.$$

3 Learning

In this chapter we will introduce some types of learning. In order to get familiar with the materia, we will take a closer look on explanatory and behaviourally correct learning. This chapter will mainly follow [16], where the reader also can find more results.

3.1 Basic Concept

Basically, learning a function f, or a set, in our terms of speaking will consist of the following steps.

- Receive a datum of the function, i.e. $f(n)$.
- Carry out a computation with the new datum and the data and hypotheses so far.
- Suggest a hypothesis which function could be the sought one.

So, we will receive more and more data, and then hopefully generate a hypothesis that fits the function in some way.

3.2 EX Learning

One type of learning most of us already dealt with is the interpolation of polynomials.

Example 3.2.1: Imagine, over the time you get points, which you try to interpolate with a polynomial. So, with each point $(x_0, y_0), \ldots, (x_n, y_n)$ you estimate which polynomial p_n could be the right one, i.e. for which $\forall i \leq n : p_n(x_i) = y_i$.

If we fix a maximal degree for those polynomials, then this search will actually come to an end. However, if that is not the case, we can learn the polynomial in the limit. The meaning of the latter is that we do not really know when we have found our right polynomial, but from some point onwards, we will not change our mind on which is the right one anymore. With this idea in mind we get the following criterion. Just before that, we will make an **important remark**.

Remark 3.2.2: In what follows, also throughout this whole chapter, we will only consider total computable functions as input. If, in some cases, we allow other functions, too, we will stress that explicitly.
One could consider an analogous attempt with partial computable functions, as it is described in [2].

Definition 3.2.3 (EX)**:** A class $\mathcal{S}$ of total computable functions is *explanatorily learnable* if there exists a computable machine $\mathtt{M}$ such that, for every $f \in \mathcal{S}$, $\mathtt{M}(f(0)f(1)\dots f(n))$ converges to a fixed e, which then is the code for a program of f, i.e.

$$\exists \text{total comp. } \mathtt{M}\ \forall f \in \mathcal{S}\ \exists e\ \forall^{\infty} n : \mathtt{M}(f(0)\dots f(n)) = e \wedge \varphi_e = f.$$

Notation 3.2.4: For any learning criterion, let LC be the respective abbreviation. If $\mathcal{S}$ is learnable using that learning criterion, i.e. if $\mathcal{S}$ is LC-learnable, we will write $\mathcal{S} \in \mathrm{LC}$ or $\mathcal{S}$ is LC.

Remark 3.2.5: In EX the learner provides a code in the limit for a program which explains the function.

Example 3.2.6: To conclude Example 3.2.1, let $\mathcal{S} = \{p : p \text{ is a polynomial}\}$. A process of learning could look like this

Data	Hypotheses
$(0,0)$	$p(x) = 0$
$(1,1)$	$p(x) = x$
$(2,4)$	$p(x) = x^2$
$(3,9)$	$p(x) = x^2$
$(4,4)$	$p(x) = x^2 - \frac{x(x-1)(x-2)(x-3)}{2}$
$\vdots$	$\vdots$

We see that we change the hypothesis when needed. Once we have the right polynomial, and the right degree, the sequence will converge to a certain $p(x)$.

In order to give some other examples let us prove the following.

Lemma 3.2.7: *The class* $\mathcal{S}_0 := \{f : f = \varphi_{f(0)}\}$ *is not empty.*

Proof. Consider the function $g(x,y,z) := \varphi_{\varphi_x(x)}(z)$.
Using the Normal Form Theorem 2.3.4 $(*)$ and the S_n^m Theorem 2.3.5 $(**)$ we get

$$g(x,y,z) \overset{(*)}{=} \varphi_e(x,y,z) \overset{(**)}{=} \varphi_{s(e,x,y)}(z) \overset{(*)}{=} \varphi_{\varphi_m(x,y)}(z) = \tag{3.1}$$

$$\overset{(**)}{=} \varphi_{\varphi_{\tilde{s}(m,x)}(y)}(z) \overset{(*)}{=} \varphi_{\varphi_{\varphi_k(x)}(y)}(z). \tag{3.2}$$

So, we have $g(x,y,z) = \varphi_{\varphi_{\varphi_k(x)}(y)}(z)$. Evaluating g on $x = k$, $y = 0$ we get

$$g(k,0,z) = \varphi_{\varphi_k(k)}(z) \text{ by definition,}$$
$$g(k,0,z) = \varphi_{\varphi_{\varphi_k(k)}(0)}(z) \text{ with the equation from (3.1) and (3.2).}$$

Letting $f(y) := \varphi_{\varphi_k(k)}(y)$, we get

$$f(z) = \varphi_{f(0)}(z).$$

Thus, $\mathcal{S}_0$ is not empty. □

Remark 3.2.8: Choosing $y = n$ we get that $\{f \mid f = \varphi_{f(n)}\}$ is not empty either.

Example 3.2.9: The classes $\mathcal{S}_0 := \{f \mid f = \varphi_{f(0)}\}$ and $\mathcal{S}_1 := \{f \mid \forall^\infty x : f(x) = 0\}$ are EX.
Both $\mathcal{S}_0$, due to the previous lemma, and $\mathcal{S}_1$, since $o(x)$ is a class member, are non-empty classes.
The machine which would learn the first class would be the constant $\mathtt{M}(f(0)\dots f(n)) = f(0)$ machine. So, $\mathtt{M}$ outputs the first datum, which then is the program for the function. For $f \in \mathcal{S}_1$, let $F := \{(x, f(x)) : f(x) > 0\}$ and $F_n := \{(x, f(x)) \mid x \leq n, f(x) > 0\}$. Also, let N be such that $f(0)\dots f(N)0^\infty$. Then $F = F_k$ for $k \geq N$. One can easily see that

$$f(x) = \begin{cases} y, (x, y) \in F, \\ 0, \text{else.} \end{cases}$$

We can choose its Kleene index to be dependent on F only. So, for some computable s, we get $f(x) = \varphi_{s(F)}(x)$. Then, the machine $\mathtt{M}(f(0)\dots f(n)) = s(F_n)$ actually outputs the right code $s(F)$. Notice, that the fixed F is quite important, as the index does not depend on n anymore. So it cannot produce any semantically equivalent but syntactically different code.

However, the union of those two is not EX, see also [2] or [3].

Theorem 3.2.10 (Blum and Blum's Non-Union Theorem)**:** *There exists* $\mathcal{C}_1, \mathcal{C}_2 \in$ EX *such that* $\mathcal{C}_1 \cup \mathcal{C}_2 \notin$ EX.

Proof. We will show that $\mathcal{S}_0$ and $\mathcal{S}_1$ from Example 3.2.9 are the sought classes. Assume $\mathtt{M}$ learns $\mathcal{S}_1$. We will define a function

$$\tilde{f}(x, y) = x\tau_1\tau_2\dots\tau_y$$

where τ_i is a finite sequence such that $\mathtt{M}(x\tau_1\dots\tau_{i-1}) \neq \mathtt{M}(x\tau_1\dots\tau_{i-1}\tau_i), i \in \omega$.
The search for τ_i terminates as $\mathtt{M}$ learns all finite sequences, i.e. $\sigma y 0^\infty$, and almost all of them have different codes. Also notice that no τ_i is the empty string. Since $\mathtt{M}$ is computable, we can computably find τ_i. So, $\tilde{f}$ is computable and total. By Corollary 2.3.7 there exists n such that $\tilde{f}(n, y) = \varphi_n(y)$, thus $\tilde{f}$ looks like

$$\tilde{f}(n, 0) = n, \tilde{f}(n, 1) = n\tau_1, \tilde{f}(n, 2) = n\tau_1\tau_2, \dots, \tilde{f}(n, y) = n\tau_1\tau_2\dots\tau_y.$$

If we define $f(y) := \tilde{f}(n, y) = \varphi_n(y)$, then $f(y) = \varphi_{f(0)}(y)$, as $f(0) = \tilde{f}(n, 0) = n$. So, $f \in \mathcal{S}_0$. However, $\mathtt{M}$ does not halt when trying to learn f. □

With that theorem in mind, the next result follows easily.

Corollary 3.2.11: REC $\notin$ EX.

Proof. As $\mathcal{S}_0 \cup \mathcal{S}_1 \subseteq$ REC, no machine can learn REC in EX as it would have to learn $\mathcal{S}_0 \cup \mathcal{S}_1$, too. □

We can try to expand this mechanism of learning. For example, in EX we have allowed finitely many "wrong" hypotheses. Now, we could try to restrict this, say we allow only one guess and no guess until then.

Definition 3.2.12 (FIN)**:** A class $\mathcal{S}$ is *finitely learnable* if there exists a computable machine $\mathtt{M}$ which either outputs "?" or exactly one correct hypothesis, i.e.

$$\exists \text{comp. } \mathtt{M}\ \forall f \in \mathcal{S}\ \exists e\ \forall^{\infty}_{N} n : \mathtt{M}(f(0)\dots f(n)) = \begin{cases} ?, & n < N, \\ e, & n \geq N. \end{cases} \wedge f = \varphi_e.$$

Remark 3.2.13: The symbol "?" is used to signal that there is no guess, i.e. the information so far is not enough to make that one right guess.

Remark 3.2.14: The machines in FIN are in some sense total. They may not output a hypothesis, however, they have to output the information that they are not sure which hypothesis to output yet.

Considering EX and FIN, we could also request that all the hypotheses made are programs of total functions. Additionally to that, we will request that even if we input some non-computable, total function which is not in the class, we still get some possibly wrong hypothesis. These ideas applied to the learning criteria we have so far, are gathered in PEX and PFIN[1]. For what follows, let $\mathcal{T}$ be the class of all total functions.

Definition 3.2.15 (PEX)**:** A class $\mathcal{S}$ is *Popperianly explanatorily learnable* if there exists a computable machine $\mathtt{M}$ such that, for every $f \in \mathcal{S}$, $\mathtt{M}(f(0)f(1)\dots f(n))$ converges to a fixed e, which then is the code for a program of f and on its way there it only outputs hypotheses for total functions. We also request $\mathtt{M}$ to output total hypotheses for $f \in \mathcal{T} \setminus \mathcal{S}$. I.e.

$$\exists \text{total comp. } \mathtt{M}\Big[\big[\forall f \in \mathcal{S}\ \exists e\ \forall^{\infty} n : \mathtt{M}(f(0)\dots f(n)) = e \wedge \varphi_e = f\big] \wedge \\ \wedge \big[\forall f \in \mathcal{T}\ \forall n\ \exists e_n : \mathtt{M}(f(0)\dots f(n)) = e_n \wedge \varphi_{e_n} \text{ is total}\big]\Big].$$

Definition 3.2.16 (PFIN)**:** A class $\mathcal{S}$ is *Popperianly finitely learnable* if there exists a computable machine $\mathtt{M}$ which either outputs "?" or exactly one correct hypothesis, which is total. We also request $\mathtt{M}$ to output total hypotheses for $f \in \mathcal{T} \setminus \mathcal{S}$. I.e.

$$\exists \text{comp. } \mathtt{M}\Big[\big[\forall f \in \mathcal{S}\ \exists e\ \forall^{\infty}_{N} n : \mathtt{M}(f(0)\dots f(n)) = \begin{cases} ?, & n < N, \\ e, & n \geq N. \end{cases} \wedge f = \varphi_e\big] \wedge \\ \wedge \big[\forall f \in \mathcal{T}\ \forall^{\infty}_{N} n\ \exists e_n : \mathtt{M}(f(0)\dots f(n)) = \begin{cases} ?, & n < N, \\ e_n, & n \geq N. \end{cases} \wedge \varphi_{e_n} \text{ is total}\big]\Big].$$

In order to better compare these learning criteria, let us sum up their main features in the following table.

[1]Here, the "P" stands for "Popperian", named after the philosopher Popper.

Name	Criterion	Additional condition	Additional functions
EX	Finitely many hypotheses, and last one is correct.	None.	None.
FIN	One correct hypothesis, and "?" for no hypothesis yet.	None.	None.
PEX	Finitely many hypotheses, and last one is correct.	All hypotheses are total functions.	$\mathcal{T}$
PFIN	One correct hypothesis, and "?" for no hypothesis yet.	All hypotheses are total functions.	$\mathcal{T}$

Remark 3.2.17: The slot "additional functions" states which types of functions may be also input, and for which we also request some output. However, the output for the latter ones does not have to be correct in any way.

Our aim will be to compare these criteria. Before we do that, we will prove that PEX-learning is actually the same as being able to predict the next value of a function of the class, for example see [3] or [16].

Theorem 3.2.18 (van Leeuwen and Barzdin)**:** *A class $\mathcal{S}$ is* PEX *iff there exists a total computable machine* $\mathtt{M}$ *which for every $f \in \mathcal{S}$ predicts the next value of f almost always, i.e.*

$$\exists \textit{total comp. } \mathtt{M} \ \forall f \in \mathcal{S}\ \forall^{\infty} n : f(n+1) = \mathtt{M}(f(0)\dots f(n)).$$

Proof. We will prove each direction separately.

$\Rightarrow$: Assume that $\mathcal{S}$ is PEX. Let $\mathtt{N}$ learn $\mathcal{S}$, and let $f \in \mathcal{S}$. Define

$$\mathtt{M}(f(0)\dots f(n)) := \varphi_{\mathtt{N}(f(0)\dots f(n))}(n+1).$$

First of all note that $\mathtt{N}$ always outputs a hypothesis for a total function. Thus, $\mathtt{M}$ is total too. Secondly, as $\mathtt{N}$ learns f, there exists $e \in \omega$ such that $\forall_r^{\infty} n : \mathtt{N}(f(0)\dots f(n)) = e$ and $f = \varphi_e$. So,

$$\forall_r^{\infty} n : \mathtt{M}(f(0)\dots f(n)) = \varphi_{\mathtt{N}(f(0)\dots f(n))}(n+1) = \varphi_e(n+1) = f(n+1).$$

$\Leftarrow$: Let now $\mathtt{M}$ be a total computable machine which predicts the functions in $\mathcal{S}$, i.e. $\forall f \in \mathcal{S}\ \forall^{\infty} n : f(n+1) = \mathtt{M}(f(0)\dots f(n))$.
Let $f \in \mathcal{S}$. For a finite sequence σ define

$$F_\sigma(n) = \begin{cases} \sigma(n), & n \in \mathrm{dom}(\sigma), \\ \mathtt{M}(f(0)\dots f(n-1)), & \text{else.} \end{cases}$$

As σ is finite and $\mathtt{M}$ is total and computable, F_σ is total and computable.
As $\mathtt{M}$ predicts f for almost all n, we get $F_{f(0)\dots f(n)}(x) = f(x)$ for almost all n and for all x.

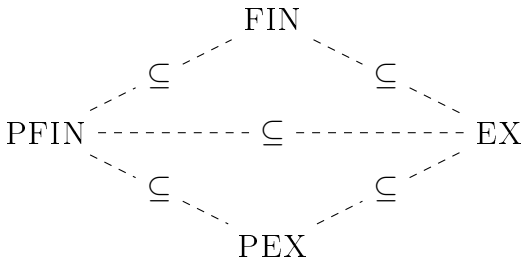
Now we will explain N, the machine to learn $\mathcal{S}$. Given the input $f(0)\dots f(n)$ our machine N searches for the least $m \leq n$ such that

$$\forall k \leq n : F_{f(0)\dots f(m)}(k) = f(k).$$

Basically, we search for the shortest starting sequence such that together with the predicting machine M we already know the whole function f, i.e. we search for the point, where we can start predicting the function. Now, if we have such a point, N outputs some index[2] $s(f(0)\dots f(m))$ of $F_{f(0)\dots f(m)}$. Notice, how this index is only depending on m and the fixed first few values $f(0)\dots f(m)$. Also, notice how m is actually important, as otherwise the computation for the code may not stop.
As F_σ is total, all indices which N outputs must be indices of total functions. And as for the found m the equation $F_{f(0)\dots f(m)}(n) = f(n)$ holds, the index $s(f(0)\dots f(m))$ is an index of f. □

As promised, we will compare the learning criteria discussed so far.

Theorem 3.2.19: *The following inclusions hold*



Proof. All the inclusions follow directly from the definitions. So, we will only prove that FIN and PEX are incomparable. In order to do so, remember $\mathcal{S}_0 = \{f \mid f = \varphi_{f(0)}\}$ and $\mathcal{S}_1 = \{f \mid \forall^\infty x : f(x) = 0\}$ from Example 3.2.9.
First, we will show that $\mathcal{S}_0 \in \text{FIN} \setminus \text{PEX}$.
As the machine to learn $\mathcal{S}_0$ only outputs one hypothesis $f(0)$, which then is the correct one, it follows that $\mathcal{S}_0 \in \text{FIN}$.
Assume now that $\mathcal{S}_0 \in \text{PEX}$. Then by Theorem 3.2.18 there exists a machine M such that $\forall f \in \mathcal{S}_0\ \forall^\infty n : f(n+1) = \mathtt{M}(f(0)\dots f(n))$. Now define

$$\tilde{f}(x,n) = \begin{cases} x, & n = 0, \\ 1 + \mathtt{M}(f(0)\dots f(n-1)), & n > 0. \end{cases}$$

Now, by Corollary 2.3.7 there exists $e \in \omega$ such that $\varphi_e(n) = \tilde{f}(e,n)$. So, if we define $f(n) := \tilde{f}(e,n)$, then $f(n) = \varphi_{f(0)}(n)$. Thus, $f \in \mathcal{S}_0$. However, $\mathtt{M}(f(0)\dots f(n)) = f(n+1) = 1 + \mathtt{M}(f(0)\dots f(n))$, so M does not predict f. A contradiction.
It remains to prove $\mathcal{S}_1 \in \text{PEX} \setminus \text{FIN}$.

[2] We get this by the S_n^m Theorem 2.3.5. Let $\tilde{F}(\sigma,n) := F_\sigma(n)$. As F is computable, $\tilde{F}$ is, too. So there exists e such that $\tilde{F}(\sigma,n) = \varphi_e(\sigma,n)$. By S_n^m Theorem $F_\sigma(n) = \tilde{F}(\sigma,n) = \varphi_e(\sigma,n) = \varphi_{s(e,\sigma)}(n)$. So we get an index of F by computing $s(e,\sigma)$.

Remember that all hypotheses were total in Example 3.2.9. Thus, $\mathcal{S}_1 \in \text{PEX}$.
Now assume $\mathcal{S}_1 \in \text{FIN}$ via a machine $\mathtt{M}$. For some finite input σ the machine $\mathtt{M}$ has to make its choice for a code, i.e. $\mathtt{M}(\sigma) \neq ?$. But then for $\sigma 10^\infty$ and $\sigma 20^\infty$, which both are in $\mathcal{S}_1$, $\mathtt{M}$ cannot change its mind anymore, thus outputs the same code for both. So, one of them cannot be learned by $\mathtt{M}$. We have a contradiction again. □

To close this section, we will consider another example of a class which can be learnt by one of the criteria considered this far.

Example 3.2.20: Let $\mathcal{S}$ be a finite class of recursive functions. We will show that $\mathcal{S} \in \text{FIN}$. To see that, let $\mathcal{S} = \{f_0, \ldots, f_m\}$. As the elements of $\mathcal{S}$ are recursive, all of them have Kleene indices, so $\mathcal{S} = \{\varphi_{e_0}, \ldots, \varphi_{e_m}\}$. The functions are different, so there exist points where they differ from each other, i.e.

$$\forall i, j, 1 \leq i, j \leq m, i \neq j \ \exists \text{min. } n_{i,j} : \varphi_{e_i}(n_{i,j}) \neq \varphi_{e_j}(n_{i,j}).$$

Let $n_0 = \max_{1 \leq i,j \leq m, i \neq j}\{n_{i,j}\}$, i.e. all the functions differ on at least one point $n \leq n_0$. The machine then is

$$\mathtt{M}(f(0)\ldots f(n)) = \begin{cases} ?, & n < n_0, \\ e_i, & n \geq n_0 \wedge \forall x \leq n_0 : f(x) = f_i(x) = \varphi_{e_i}(x). \end{cases}$$

Note that $\mathtt{M}$ is computable as finitely many computable case distinctions are computable. For given f the machine makes no choice until it hits n_0. Then it compares all the data so far, i.e. for all e_i it checks $\forall x \leq n_0 : f(x) = f_i(x) = \varphi_{e_i}(x)$. Since the functions are sure to differ on at least one point $x \leq n_0$, exactly one e_i can have that property.
Thus, $\mathtt{M}$ learns $\mathcal{S}$.

3.3 BC Learning

Recall the EX learning criterion from the previous section. When given a class and a function out of it, we tried to find one code for the function and then stack to it. We can change this approach and allow the codes themselves to be different but still codes for the function. We introduce the behaviourally correct learning.

Definition 3.3.1 (BC)**:** A class $\mathcal{S}$ is *behaviourally correctly learnable* if there exists a computable machine such that for almost every initial input σ of f there exists some code e such that $\mathtt{M}(\sigma) = e$, i.e.

$$\exists \text{comp. } \mathtt{M} \ \forall f \in \mathcal{S} \ \forall^\infty \sigma \preceq f \ \exists e : \mathtt{M}(\sigma) = e \wedge \varphi_e = f.$$

Remark 3.3.2: Notice the difference to EX. While we search for **one fixed code** for the function in explanatory learning, we do not do so in behaviourally correct learning. In the latter one we search for codes that are semantically equivalent, but we do not require them to be syntactically equivalent.

We can easily compare EX and BC.

Corollary 3.3.3: *If* $\mathcal{S} \in \mathrm{EX}$ *then* $\mathcal{S} \in \mathrm{BC}$, *i.e.* $\mathrm{EX} \subseteq \mathrm{BC}$.

However, we will try to refine this inclusion. Consider the class

$$\mathcal{S}_{0,1} := \{f \mid \exists y \, \forall x \neq y : \varphi_{f(0)}(x) \downarrow = f(x)\},$$

i.e. the class of all functions f that disagree with the program $f(0)$ on no more than one argument. This class is not EX as shows the following example, following the idea in [3].

Example 3.3.4: For any given machine $\mathtt{M}$ we will find a fitting $f \in \mathcal{S}_{0,1}$ which spoils $\mathtt{M}$ as an EX-learner of $\mathcal{S}_{0,1}$.
We will execute an algorithm in order to compute a function φ_e step by step. This function will serve as a guidance later on. For this purpose, let

e: program, which can be chosen like in previous proofs to output itself first,

x: input,

α: marker, keeping track of the position of the possible anomaly,

α^s: position of the marker α at the beginning of stage s,

φ_e^s: the finite part of φ_e defined at the beginning of stage s,

x^s: the minimum of $\omega \setminus \big(\{\alpha^s\} \cup \mathrm{dom}(\varphi_e^s)\big)$,

σ^s: largest initial segment of φ_e^s.

For the initial stage $s = 0$, let

- $\alpha^0 = 1$,
- $\varphi_e^s(0) = e$ and undefined elsewhere.

At stage $s \geq 0$ execute the following.
Calculate $x^s = \min\{\omega \setminus \big(\{\alpha^s\} \cup \mathrm{dom}(\varphi_e^s)\big)\}$ first.

(i) If

$$\exists \sigma : \sigma^s \prec \sigma \preceq \big(\varphi_e^s \cup \{(\alpha^s, 0)\}\big) \wedge \mathtt{M}(\sigma^s) \neq \mathtt{M}(\sigma), \tag{3.3}$$

then

- $\varphi_e^{s+1} := \varphi_e^s \cup \{(\alpha^s, 0)\}$,
- $\alpha^{s+1} := x^s$.

(ii) If

$$\text{not (3.3) and } \varphi_{\mathtt{M}(\sigma^s),s}(\alpha^s) \downarrow, \tag{3.4}$$

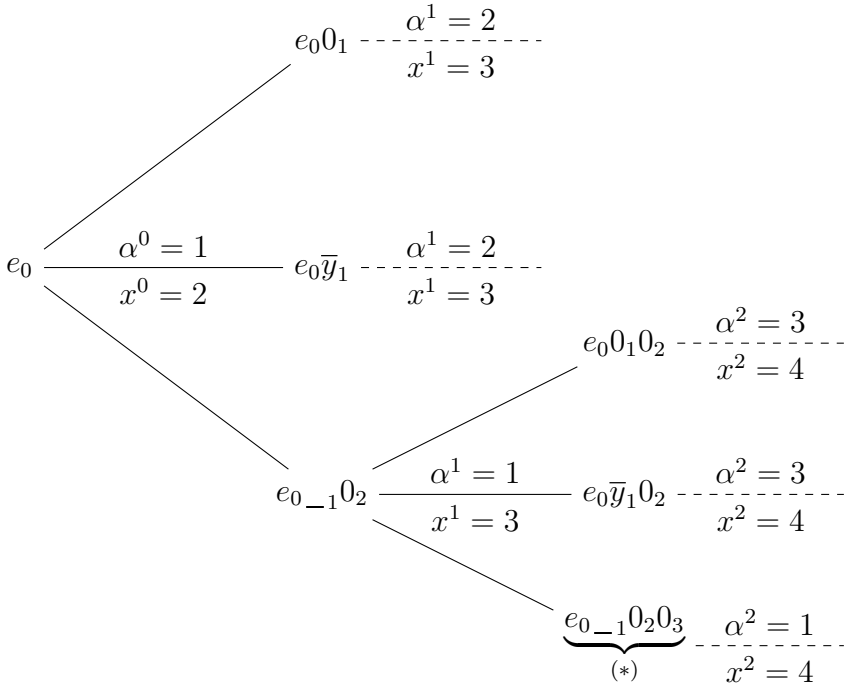
then

- $\varphi_e^{s+1} := \varphi_e^s \cup \left\{\left(\alpha^s, \overline{\mathrm{sgn}}\left(\varphi_{\mathtt{M}(\sigma^s),s}(\alpha^s)\right)\right)\right\}$,
- $\alpha^{s+1} := x^s$.

(iii) If not (3.3) and not (3.4), then

- $\varphi_e^{s+1} := \varphi_e^s \cup \{(x^s, 0)\}$,
- $\alpha^{s+1} := \alpha^s$.

In the algorithm above all steps are computable. The only case where this is not obvious is finding σ in (3.3). In order to reason why this is computable, too, let us execute the algorithm for the first few steps and thereby argue, why we only search over finitely many σ.



We will explain the upper diagram.

x_y: means that the y-th position of φ_e is defined as x. We write $x = \_$ if we know that it is undefined,

$\overline{y}$: is an abbreviation for the term $\overline{\mathrm{sgn}}(\varphi_{\mathtt{M}(\sigma^s),s}(\alpha^s))$,

lines: the information over and under the middle line are the current values of α^s and x^s, respectively, and the dashed lines just mean that it goes on like that.

(∗): We will take a closer look on the situation of σ^s and $\varphi_e^s \cup \{(\alpha^s, 0)\}$ here. We have $s = 2$, and $\sigma^2 = e_0$. Also, as $\alpha^2 = 1$, we get $\varphi_e^2 \cup \{(1, 0)\} = e_0 0_1 0_2 0_3$. So, for σ there are only three possibilities, namely $e_0 0_1, e_0 0_1 0_2$ and $e_0 0_1 0_2 0_3$.

As we are convinced that the algorithm above is computable, we will try to find the spoiling f. We have to distinguish two cases.

1.C.: $\lim_{s\to\infty} \alpha^s = \infty$.
In this case, $f := \varphi_e$ is a total computable function. As $f(0) = \varphi_e(0) = e$ and $\forall x : \varphi_e(x) = f(x)$, $f \in \mathcal{S}_{0,1}$. Now, assume that $\mathtt{M}$ learns f and that the program is some p. Then, $\mathtt{M}$ stops changing its mind, i.e.

$$\exists s\ \forall \sigma : \sigma^s \prec \sigma \prec f \Rightarrow \mathtt{M}(\sigma^s) = \mathtt{M}(\sigma) = p.$$

Past stage s, we come infinitely often to case (ii), as otherwise $\mathtt{M}$ would still change its mind. However, $\varphi_p(x)$ converges for infinitely many x. But, due to the construction,

$$\exists^\infty x : \varphi_p(x) = \varphi_{\mathtt{M}(\sigma^s)}(x) \neq \overline{\mathrm{sgn}}(\varphi_{\mathtt{M}(\sigma^s)}(x)) = f(x).$$

So, f and φ_p are different on infinitely many inputs. Thus, $\mathtt{M}$ does not learn some $f \in \mathcal{S}_{0,1}$.

2.C.: $\lim_{s\to\infty} \alpha^s = a < \infty$.
Here, $\mathrm{dom}(\varphi_e) = \omega \setminus \{a\}$. Now define

$$f(x) = \begin{cases} \varphi_e(x), & x \in \mathrm{dom}(\varphi_e), \\ 0, & \text{else.} \end{cases}$$

f is a total recursive function, which is 0 almost everywhere. Again, $f(0) = \varphi_e(0) = e$ and $\forall x \neq a : f(x) = \varphi_e(x)$, so $f \in \mathcal{S}_{0,1}$. Let s be large enough that $\alpha^s = a$. Then, $\forall s' > s : \sigma^s = \sigma^{s'}$. Past stage s, we always end up in case (iii). So,

$$\forall \sigma : \sigma^s \prec \sigma \prec f \Rightarrow \mathtt{M}(\sigma^s) = \mathtt{M}(\sigma) = p,$$

where p is a program that $\mathtt{M}$ outputs when fed f. However, $\mathtt{M}(\sigma^s)$ diverges on input a. So, $\mathtt{M}(\sigma^s)$ does not compute f.
Again, $\mathtt{M}$ does not learn some $f \in \mathcal{S}_{0,1}$.

So, for any machine $\mathtt{M}$ we have found some $f \in \mathcal{S}_{0,1}$ which cannot be learnt. Thus, $\mathcal{S}_{0,1} \notin \mathrm{EX}$.

We see, if we may have some mistake, we cannot learn explanatory correct. In order to circumvent this state, we introduce the learning with mistakes.

Definition 3.3.5: Let $\mathcal{S}$ be a class, SLC be any suitable learning criterion and $a \in \mathbb{N}_*$. Then $\mathcal{S} \in \mathrm{SLC}^a$ if there exists a machine $\mathtt{M}$ which, for every $f \in \mathcal{S}$, outputs a code e according to SLC, such that $\varphi_e =^a f$, i.e. the program e computes f on all but a many positions.

As $f(0)$ is a code for $f \in \mathcal{S}_{0,1}$ with at most one mistake, we see that $\mathcal{S}_{0,1} \in \mathrm{EX}^1$.
As we did in the previous section, we will provide some inclusions here, too.

Theorem 3.3.6: *The following inclusions hold*

$$\mathrm{EX} \subseteq \mathrm{EX}^1 \subseteq \mathrm{EX}^2 \subseteq \cdots \subseteq \mathrm{EX}^* \subseteq \mathrm{BC} \subseteq \mathrm{BC}^1 \subseteq \mathrm{BC}^2 \subseteq \cdots \subseteq \mathrm{BC}^*.$$

Proof. For $a \leq b$, the inclusions $\mathrm{EX}^a \subseteq \mathrm{EX}^b, \mathrm{BC}^a \subseteq \mathrm{BC}^b$ are rather obvious. What remains to prove is $\mathrm{EX}^* \subseteq \mathrm{BC}$.
Let $\mathcal{S} \in \mathrm{EX}^*$ via $\mathtt{M}$. Consider the following function

$$g(\sigma, x) = \begin{cases} \sigma(x), & x \in \mathrm{dom}(\sigma), \\ \varphi_{\mathtt{M}(\sigma)}(x), & \text{else.} \end{cases}$$

Let $f \in \mathcal{S}$. At some point x_0, $\mathtt{M}$ converges to some program e of f. Let y be the point of the last mistake φ_e makes when compared to f. Now, for $n > \max\{x_0, y\}$ we have $\mathtt{M}(f(0)\dots f(n)) = e$ and, with $\sigma = f(0)\dots f(n)$,

$$g(\sigma, x) = \begin{cases} \sigma(x), & x \in \mathrm{dom}(\sigma), \\ \varphi_{\mathtt{M}(\sigma)}(x), & \text{else.} \end{cases} = \begin{cases} \sigma(x), & x \in \mathrm{dom}(\sigma), \\ \varphi_e(x), & \text{else.} \end{cases} = f(x).$$

By Normal Form Theorem 2.3.4 and S_n^m Theorem 2.3.5, there exists some computable s such that

$$\varphi_{s(\sigma)}(x) = g(\sigma, x).$$

Let $\mathtt{N}(\sigma)$ output $s(\sigma)$, then $\mathtt{N}$ learns f for almost all inputs, namely for all $f(0)\dots f(n)$, $n > \max\{x_0, y\}$. □

We have witnessed that allowing mistakes leads to more learnable classes. However, if we replace "at most one mistake" by "exactly one mistake", as it is suggested in [3], the learnable classes stay the same. We will show this, following the idea in [3].

Lemma 3.3.7: *Let $\mathcal{S} \in \mathrm{EX}^{=1}$ if $\mathcal{S}$ is learned explanatory, i.e.* EX, *but with exactly one mistake. Then,* $\mathrm{EX}^{=1} = \mathrm{EX}$.

Proof. We will show each inclusion separately.

$\subseteq$: Let $\mathcal{S} \in \mathrm{EX}^{=1}$ via $\mathtt{M}'$. We will give an algorithm for $\mathtt{M}$. Let $s \geq 0$ be the stage and $\sigma_s := f(0)\dots f(s)$ be the input at stage s. Also, let $i = 0$ be some counting variable. Also, introduce some repair function

$$g(z, y, x) = f(y)\overline{\mathrm{sgn}}(|x - y|) + \varphi_z(x)\,\mathrm{sgn}(|x - y|).$$

One can easily see that

$$g(z, y, x) = \begin{cases} f(y), & x = y, \\ \varphi_z(x), & \text{else.} \end{cases}$$

1. While $i \leq s$, check whether $f(i) = \varphi_{\mathtt{M}'(\sigma_s),s}(i)$.
   i. If so, raise i, i.e. $i := i + 1$.
   ii. If not, consider $g(\mathtt{M}'(\sigma_s), i, x)$. As composition of computable functions, it is computable itself. So, by Normal Form Theorem 2.3.4 and S_n^m Theorem 2.3.5 there exists some computable c such that

   $$g(z, y, x) = \varphi_{c(z,y)}(x).$$

   Output $c(\mathtt{M}'(\sigma_s), i)$ and reset i to zero.

2. If $i = s + 1$, then output $\mathtt{M}'(\sigma_s)$ and reset i to zero.

We need to explain why the algorithm works as desired. In (1.) we check whether we have found the anomaly or not, which is computable, as we only check what $\varphi_{\mathtt{M}'(\sigma)}$ outputs after finitely many, namely s, steps. Now, if they are equal, then we land at (1.i.), raise i and repeat the question. If this case happens all the time, i.e. $s + 1$ times, we land at (2.). Thus, we have not found any mistake, so we can output the same as $\mathtt{M}'$ does, as the mistake is yet to come.
However, once we are in (1.ii.), we found an, at least temporary, anomaly. We fix that by using g. In more detail, we use $g(\mathtt{M}'(\sigma_s), i, x)$, which is $\varphi_{\mathtt{M}'(\sigma)}$ on all input except for i, where we manually tell it to be $f(i)$. In this case, we output $c(\mathtt{M}'(\sigma), i)$. Since we know that there is some mistake, the case (1.ii.) will be hit at some point, and from some point, we will always hit the mistake. From some stage s, $\mathtt{M}'(\sigma_s)$ halts, i.e. $\forall_s^\infty s' : \mathtt{M}'(\sigma_{s'}) = \mathtt{M}'(\sigma_s)$. Thus, $\mathtt{M}$ halts, too, and outputs $\forall_s^\infty s' : c(\mathtt{M}'(\sigma_s), i)$.

$\supseteq$: Let $\mathcal{S} \in \mathrm{EX}$ via $\mathtt{M}$. This time, we try to destroy our function at some point. Therefore, introduce the destroyer function

$$h(z, y, x) = (f(y) + 1)\overline{\operatorname{sgn}}(|x - y|) + \varphi_z(x)\operatorname{sgn}(|x - y|).$$

One can easily see that

$$h(z, y, x) = \begin{cases} f(y) + 1, & x = y, \\ \varphi_z(x), & \text{else.} \end{cases}$$

Now, given $\sigma_s := f(0) \dots f(s)$, $\mathtt{M}'$ simply outputs the code of $h(\mathtt{M}(\sigma_s), 0, x)$ which is some $c(\mathtt{M}(\sigma_s), 0)$, where c is computable. As $\mathtt{M}$ halts to some $e_{\mathtt{M}}$, the machine $\mathtt{M}'$ halts to some $c(e_{\mathtt{M}}, 0) = e_{\mathtt{M}'}$, too, but errs only at 0, as

$$h(e_{\mathtt{M}'}, 0, x) = \begin{cases} f(0) + 1, & x = 0, \\ \varphi_{e_{\mathtt{M}'}}(x), & \text{else.} \end{cases} = \begin{cases} f(0) + 1, & x = 0, \\ f(x), & \text{else.} \end{cases}$$

□

One property we want to show is that we can learn REC in BC^*, for example see [3] or [16].

Theorem 3.3.8 (Leo Harrington)**:** $\mathrm{REC} \in \mathrm{BC}^*$.

Proof. For some input σ define the following computable function ψ_σ

$$\varphi_{s(\sigma)}(x) = \psi_\sigma(x) := \begin{cases} \varphi_e(x), & \text{for the smallest } e \leq x \text{ such that} \\ & \varphi_{e,x}(y) \downarrow = \sigma(y) \text{ for all } y \in \operatorname{dom}(\sigma), \\ 0, & \text{else.} \end{cases}$$

Let now $f \in \mathrm{REC}$. Then f has an index e which can be chosen to be minimal[3]. As e is the minimal index of f, all programs $i < e$ have to be different from f at some point

[3]Notice, that this is a theoretical attempt here. Finding a minimal index cannot be done effectively. Attempts in that direction can be found for example in [17].

m. As we are only speaking about finitely many programs, we can take the maximum of those m. So there is some n such that all φ_i, where $i < e$, are different at some point $m \leq n$ from f, i.e. either both are defined and different, or φ_i is not defined at all.
Now, if $\sigma = f(0) \dots f(k), k \geq n$ and $\varphi_{j,x}(y) \downarrow = \sigma(y)$ for all $y \in \mathrm{dom}(\sigma)$, then the index j has to be greater than e, so $j \geq e$.
For $k \geq n$ and $\sigma = f(0) \dots f(k)$, in order to compute all $\varphi_{e,x}(y), y \in \mathrm{dom}(\sigma)$ we only need finitely many steps, say x_0 many. So, we have $\varphi_{e,x}(y) \downarrow = \sigma(y)$ for all $y \in \mathrm{dom}(\sigma)$ and all sufficiently big x, namely $x \geq x_0$.
Summing up, we get, for $\sigma = f(0) \dots f(k), k \geq n$,

(i) if $\varphi_{j,x}(y) \downarrow = \sigma(y)$ for all $y \in \mathrm{dom}(\sigma)$, then $j \geq e$,

(ii) $\varphi_{e,x}(y) \downarrow = \sigma(y)$ for all $y \in \mathrm{dom}(\sigma)$ and all sufficiently big x.

Yet again, let $\sigma = f(0) \dots f(k), k \geq n$. Then, for almost all x, we have $\psi_\sigma(x) \downarrow \overset{\text{(ii)}}{=} \varphi_e(x) = f(x)$.
Summing up, we have $\psi_\sigma(x) = f(x)$ for almost all $\sigma \preceq f$ and almost all x, i.e.

$$\forall^\infty \sigma \preceq f \; \forall^\infty x \; \exists s(\sigma) : \varphi_{s(\sigma)}(x) = f(x).$$

Thus, the machine to learn REC in BC* will output $s(\sigma)$. □

Now we know that we can behaviourally correctly, but with finitely many mistakes, learn all computable functions.
Remembering Theorem 3.2.18 for the case PEX, we can try to obtain something similar for BC.

Theorem 3.3.9: *Let* $\mathcal{S} \subseteq$ REC.
$\mathcal{S}$ *is* BC *if and only if there is a partial computable machine* $\mathtt{N}$ *which predicts every* $f \in \mathcal{S}$ *almost everywhere, i.e.*

$$\exists \textit{part. comp. } \mathtt{N} \; \forall f \in \mathcal{S} \; \forall^\infty x : \mathtt{N}(f(0) \dots f(x-1)) \downarrow = f(x).$$

Proof. Again, we will prove each direction separately.

⇒: Let $\mathcal{S} \in$ BC via a machine $\mathtt{M}$. Define $\mathtt{N}(\sigma) := \varphi_{\mathtt{M}(\sigma)}(|\sigma|)$. Once $\mathtt{M}$ learns f, we have that, for almost all $\sigma \preceq f$, $\mathtt{N}(\sigma)$ is defined and

$$\mathtt{N}(\sigma) = \varphi_{\mathtt{M}(\sigma)}(|\sigma|) = f(|\sigma|).$$

Now, if $\sigma = f(0) \dots f(x-1)$ is big enough, we get

$$\mathtt{N}(f(0) \dots f(x-1)) \downarrow = \mathtt{N}(\sigma) = \varphi_{\mathtt{M}(\sigma)}(|\sigma|) = f(|\sigma|) = f(x).$$

⇐: Let $\mathtt{N}$ be the predicting machine. Consider the partial computable function

$$f(y, \sigma, x) = \begin{cases} \sigma(x), & x \in \mathrm{dom}(\sigma), \\ \mathtt{N}(\varphi_{\varphi_y(\sigma)}(0) \dots \varphi_{\varphi_y(\sigma)}(x-1)), & \text{else.} \end{cases}$$

By Normal Form Theorem 2.3.4 and S_n^m Theorem 2.3.5 there exists a computable $s = \varphi_n$ such that

$$f(y, \sigma, x) = \varphi_e(y, \sigma, x) = \varphi_{\tilde{s}(e,y,\sigma)}(x) = \varphi_{s(y,\sigma)}(x) = \varphi_{\varphi_n(y,\sigma)}(x). \tag{3.5}$$

Now, by Corollary 2.3.8, there exists some computable g such that $\varphi_{g(n)}(\sigma) = \varphi_n(g(n), \sigma)$. Using that and letting $y = g(n)$ in (3.5), we obtain

$$\varphi_{\varphi_{g(n)}(\sigma)}(x) = \varphi_{\varphi_n(g(n),\sigma)}(x) =$$
$$= f(g(n), \sigma, x) = \begin{cases} \sigma(x), & x \in \mathrm{dom}(\sigma), \\ \mathtt{N}(\varphi_{\varphi_{f(n)}(\sigma)}(0) \dots \varphi_{\varphi_{f(n)}(\sigma)}(x-1)), & \text{else.} \end{cases}$$

Let $\mathtt{M}(\sigma) = \varphi_{g(n)}(\sigma)$. Notice that $\mathtt{M}$ is computable. Then,

$$\varphi_{\mathtt{M}(\sigma)}(x) = \begin{cases} \sigma(x), & x \in \mathrm{dom}(\sigma), \\ \mathtt{N}(\varphi_{\mathtt{M}(\sigma)}(0) \dots \varphi_{\mathtt{M}(\sigma)}(x-1)), & \text{else.} \end{cases}$$

Let $f \in \mathcal{S}$. Now, if this inductive definition is undefined at one point x because $\mathtt{N}$ is undefined there, then $\varphi_{s(\sigma)}(y)$ is undefined, too, for all $y \geq x$. However, once σ is long enough for $\mathtt{N}$ to predict f for all $x \geq |\sigma|$, then $f(\sigma, x) = g(x)$ and thus $\varphi_{s(\sigma)}(x) = g(x)$. So, $s(\sigma)$ is a program for g.
So, almost all hypotheses are correct and $\mathtt{M}$ learns f in BC. □

4 Confident Iterative Learning

By now, we are familiar with some basic learning criteria. In order to advance, we will investigate a new type of learning, the confident iterative one.

4.1 Main definitions

Opposite to what we have done so far, we will now infer languages, i.e. **non-empty c.e. sets**, rather than functions. One natural form of learning languages is receiving the positive information of the language, i.e. getting to know more and more words which are in the language. As suggested in [16], we can compare that to some archaeologist coming across some ruins. There, the archaeologist will find words of the language, rather than words that are not. This kind of language learning is called text-learning, see [16]. It is obvious that the order of the input in this kind of learning can be arbitrary. Thus, we need to fix the following notation.

Notation 4.1.1: Let $D \subseteq \mathbb{N}$ be some set. If $d_0, d_1, \dots$ is some *input sequence of* D, i.e. if $\{d_0, d_1, \dots\} = D$, we will write $(d_n)_n \in D$.

Remark 4.1.2: Sometimes, one allows the use of a pause symbol #. This is done to be able to represent the empty set, namely by inputting $\#, \#, \dots$. However, since we are not interested in learning the empty set, and since none of the classes here contain the empty set, we will omit using the pause symbol.

Now, we need to clarify what it means to learn a set.

Definition 4.1.3: Let $D \subseteq \mathbb{N}$ be some set. We call e a program of D iff $W_e = D$.

Remark 4.1.4: Intuitively, a program of a set S should provide us with the information which elements are in the set. For example, if we know the characteristic function χ_S, we can compute the set. However, that is not consistent with our definition of a program of S. Since $\chi_S = \varphi_e$ is everywhere defined, $W_e = \mathbb{N}$. To circumvent that, we will allow reinterpretations of programs, i.e. e is a program of S iff $W_{s(e)} = S$ for some computable s. We will call such s a *translation*.
Using that and $f(p, m, x) = \varphi_p(x + m)$ if latter is 1, and undefined else, we can prove that having the shifted characteristic function χ_S allows us to compute the set. For fixed m, let e be such that $\varphi_e(x + m) = \chi_S(x)$. Then, using S_n^m Theorem 2.3.5 on $f = \varphi_n$, we obtain

$$\varphi_{s(n,p,m)}(x) = \varphi_n(p, m, x) = f(p, m, x) = \begin{cases} 1, & \varphi_p(x + m) = 1, \\ \uparrow, & \text{else.} \end{cases}$$

Then, $W_{s(n,e,m)} = S$.
Using, for example, this shifted version of the characteristic function allows us to use the first m positions as some memory. A more detailed look on this can be found in [11].

We will omit mentioning the use of the previous remark, however, we will provide the way of recomputing. Also, to make future algorithms more readable, we will use the abbreviations discussed in the next remark.

Remark 4.1.5: Later on, the presented algorithms will use the terms "output x" and "return 0" in their codes.
While the first means that the algorithm outputs x as its current hypothesis and then requests the next input, the second implies that the computation could actually be aborted here. Technically speaking, the algorithm would start outputting some dummy code, say 0, implying that the computation is over.

So far, the learning machines had full access to all input data. However, we will restrict that might by quite a bit. In the next type of learning, the machine will get the new datum as input as well as its own last hypothesis. Thus, any memory on previous computations can only be stored into the output itself. This type of learning is known as iterative learning. We will take the definition from [10] and [13] and adapt it to fit our needs.

Definition 4.1.6 (IT)**:** A class $\mathcal{S}$ is *iteratively learnable* iff there exists a machine which converges in the process of mapping the old hypothesis p_i and the current datum d_i to a new hypothesis p_{i+1}, i.e.

$$\exists \text{comp. } \mathtt{it}(.,.)\ \forall D \in \mathcal{S}\ \forall (d_n)_n \in D\ \exists p\ \forall^\infty i : \mathtt{it}(p_i, d_i) = p_{i+1} = p \wedge W_p = D.$$

Such a learner $\mathtt{it}(.,.)$ is called *iterative learner*.

Remark 4.1.7: One can choose the first input program p_0 arbitrarily.

Next, we will enhance the idea of the Popperian learner from the last chapter. In addition to the regular input, i.e. input that belongs to some set in the class, we will allow any arbitrary input as well. However, the learner has to converge on both types of inputs, but only has to be correct on the regular one. To be able to capture that idea formally well, let us widen notation 4.1.1.

Notation 4.1.8: Let $\mathcal{S}$ be a class. If we have some input sequence $d_0, d_1, \ldots$, we write $(d_n)_n \in D_\mathcal{S}$ if the input belongs to some set $D_\mathcal{S}$ in $\mathcal{S}$, i.e. if $\exists D_\mathcal{S} \in \mathcal{S} : (d_n)_n \in D_\mathcal{S}$. We will call such an input regular.
Otherwise, or if we allow both cases, we simply write $(d_n)_n$.

Now, we can formalize the idea above, thus introducing the confident learning machine, as it is explained in [6] or [8].

Definition 4.1.9 (CFD)**:** A class $\mathcal{S}$ is *confidently learnable* iff there exists a machine which converges on any input sequence and is correct on the regular input, i.e.

$$\exists \text{comp. } \mathtt{cfd}\ \forall (d_n)_n\ \exists e : \Big(\big(\forall^\infty i : \mathtt{cfd}(d_0, d_1, \ldots, d_i) = e\big) \wedge \big((d_n)_n \in D_\mathcal{S} \Rightarrow W_e = D_\mathcal{S}\big) \Big).$$

Such a learner cfd is called *confident learner*.

Our main focus will lay on the combination of these two, the confident iterative learner.

Definition 4.1.10 (CI)**:** A class $\mathcal{S}$ is *confidently iteratively learnable* iff there is a learner $\texttt{ci}(.,.)$ learning $\mathcal{S}$ confidently and iteratively.
Such a learner $\texttt{ci}(.,.)$ is called *confident iterative learner*.

4.2 First efforts

Naturally, we will investigate some classes in order to check whether they are CI. The start marks the class of all finite sets.

Example 4.2.1: Let $\mathcal{P} := \mathcal{P}_{fin}(\omega) := \{A \subseteq \omega : |A| < \infty\}$, i.e. the class of all finite subsets of ω.
First, we will show that this class is iteratively learnable. Let $(d_n)_n \in A_{\mathcal{P}}$ be the input for the following algorithm. Let p_0 be a program of the zero function. Then

- for inputs p_i and d_i, where p_i is the old hypothesis,
  1. if $\varphi_{p_i}(d_i) = 0$, then p_{i+1} is the program of $f(x) = \varphi_{p_i}(x) + \overline{\text{sgn}}(|x - d_i|)$,
  2. otherwise, $p_{i+1} = p_i$.

As composition of such, $f(x)$ is a total computable function. By Corollary 2.3.9 we can compute its index effectively via p_i. So, all steps in the algorithm are effective.
To see that the algorithm works properly, let $(d_n)_n \in A_{\mathcal{P}}$ be some input sequence. For input d_i and p_i, the algorithm checks whether or not d_i is some new datum, i.e. $\varphi_{p_i}(d_i) = 0$. If so, we change that value to 1, using f. Then, p_{i+1} is the program of the current characteristic function. Since the sets in the class are finite, there exists some stage I when all data has occurred in the input sequence at least once, i.e. $\forall_I^\infty i : \{d_0, \ldots, d_I\} = \{d_0, \ldots, d_i\} = A_{\mathcal{P}}$. From this point onwards, the algorithm will not change its mind anymore. Thus, we have learned the class iteratively.
Next, we will argue that this class is not in CI. For the full technical proof, consider the proof of Lemma 4.2.3. Now, assume the class is in CI. Then it has some confident iterative learner $\texttt{ci}(.,.)$. Take some infinite ascending chain $A_0 \subsetneq A_1 \subsetneq \cdots$ in the class. As $\texttt{ci}(.,.)$ learns the class, it has to converge on every of these A_i. So, we start inputting elements from A_0 until $\texttt{ci}(.,.)$ converges, and then continue to input elements from A_1 until it converges again, just to go on with elements from A_2, etc. Since all of these sets have different codes, $\texttt{ci}(.,.)$ will not converge on that input in $A := \bigcup_i A_i$.

This example already provides some useful facts.

Lemma 4.2.2: *Any subclass $\mathcal{S}$ of $\mathcal{P}_{fin}(\omega)$ can be learned iteratively.*

Proof. Let $\mathcal{S}$ be a subclass of $\mathcal{P}_{fin}(\omega)$ and $(d_n)_n \in A_{\mathcal{S}}$ the input for the algorithm below. Let p_0 be the program of the zero function. Then

- for inputs p_i and d_i, where p_i is the old hypothesis,

1. if $\varphi_{p_i}(d_i) = 0$, then p_{i+1} is the program of $f(x) = \varphi_{p_i}(x) + \overline{\text{sgn}}(|x - d_i|)$,
2. otherwise, $p_{i+1} = p_i$.

In the light of Corollary 2.3.9 and since $f(x)$ is computable, we can get its code effectively via p_i. Thus, the algorithm works effectively.
Let $(d_n)_n \in A_{\mathcal{S}}$ be some input sequence. For any input d_i the algorithm checks, whether d_i is a new datum, i.e. $\varphi_{p_i}(d_i) = 0$, or not. In the first case, it changes the characteristic function on this argument to 1, leading to $f(x)$. Then it outputs a code of $f(x)$. Otherwise, it outputs the last hypothesis.
At some point I, all elements of $A_{\mathcal{S}}$ have appeared in the input, i.e. $\forall_I^\infty i : \{d_0, \ldots, d_I\} = \{d_0, \ldots, d_i\} = A_{\mathcal{S}}$. Now, the algorithm will always proceed with case (2.), as there are no yet unmentioned elements anymore.
So, the algorithm will converge, and output a code of the characteristic function of $A_{\mathcal{S}}$. □

Lemma 4.2.3: *If a class $\mathcal{S}$ is* CI*, then $\mathcal{S}$ cannot contain an infinite ascending chain.*

Proof. Let $\mathcal{S}$ have a CI-learner $\texttt{ci}(.,.)$. Assume, that $\mathcal{S}$ contains an infinite ascending chain $A_0 \subsetneq A_1 \subsetneq \cdots$. Then

A_0: for some input $d_{0,0}, d_{0,1}, \ldots$ of A_0 the CI-learner will eventually converge, so

$$\exists p_0 \; \forall_{I_0}^\infty j : \texttt{ci}(p_{0,j}, d_{0,j}) = p_{0,j+1} = p_0.$$

A_1: Starting with the previous input $d_{0,0}, d_{0,1}, \ldots, d_{0,I_0}$, and then continuing with some $d_{1,I_0+1}, d_{1,I_0+2}, \ldots$ of $A_1 \setminus A_0$ the CI-learner will eventually converge, so

$$\exists p_1 \; \forall_{I_1}^\infty j : \texttt{ci}(p_{*,j}, d_{*,j}) = p_{*,j+1} = p_1.$$

A_i: Starting with the input $d_{0,0}, d_{0,1}, \ldots, d_{0,I_0}, d_{1,I_0+1}, \ldots, d_{1,I_1}, d_{2,I_1+1}, \ldots, d_{i-1,I_{i-1}}$, and then continuing with some $d_{i,I_{i-1}+1}, d_{i,I_{i-1}+2}, \ldots$ of $A_i \setminus A_{i-1}$ the CI-learner will eventually converge, so

$$\exists p_i \; \forall_{I_i}^\infty j : \texttt{ci}(p_{*,j}, d_{*,j}) = p_{*,j+1} = p_i.$$

So, we have an input sequence

$$d_{0,0}, d_{0,1}, \ldots, d_{0,I_0}, d_{1,I_0+1}, \ldots, d_{1,I_1}, d_{2,I_1+1}, \ldots, d_{i-1,I_{i-1}}, d_{i,I_{i-1}+1}, d_{i,I_{i-1}+2}, \ldots$$

where $\texttt{ci}(.,.)$ outputs the codes $\ldots, p_0, \ldots, p_1, \ldots, p_i, \ldots$. As all of these codes are different, $\texttt{ci}(.,.)$ does not converge on this particular input of $A := \bigcup_i A_i$, a contradiction. □

One may assume that the infinite ascending chain is the only reason that bugs the existence of the CI-learner. However, this is not true, as shows the next example.

Example 4.2.4: Let $\mathbb{N}_e$ be the set of all even natural numbers, and let $\mathbb{N}_o$ be the set of all odd natural numbers. Consider the class

$$\mathcal{C} := \{E \mid \exists n : 2n+1 \in E \wedge |\mathbb{N}_e \cap E| = 2n \wedge |\mathbb{N}_o \cap E| = 1\}.$$

The sets of this class contain one odd element $2n+1$ and $2n$ many even elements.
Since all of the sets of this class are finite, it is iteratively learnable by Lemma 4.2.2.
Next, we will provide a confident learner of $\mathcal{C}$. For any input $(d_n)_n$, execute the following algorithm.

1. If $\{d_0, \ldots, d_i\}$ contains no odd elements, output 0.
2. If $\{d_0, \ldots, d_i\}$ contains one odd element $m = 2n+1$ and $2n$ many even elements, output a code of the characteristic function of $\{d_0, \ldots, d_i\}$.
3. Else, output 0.

To see that the algorithm works properly, let $(d_n)_n$ be some input sequence. The idea of the algorithm is simple, we count the odd elements, leading us into three cases on $\{d_0, \ldots, d_i\}$.

1. While we have no odd elements, we do not have a set in the class, yet, so we output any dummy code, i.e. 0.
2. When we have exactly one odd element $m = 2n+1$ and $2n$ many even elements, we output a code of the characteristic function of $\{d_0, \ldots, d_i\}$, as it is a set in the class.
3. Else, output 0.

Since the algorithm does only change its mind when changing from case (1.) or (3.) to (2.) or from (2.) to (3.), which can only happen finitely many times, we see that the algorithm works properly.
Lastly, we will prove that $\mathcal{C}$ is not CI. Assume there exists a CI-learner $\texttt{ci}(.,.)$. Consider the set of all even numbers $\mathbb{N}_e$ and some input sequence $(e_n)_n \in \mathbb{N}_e$. The learner $\texttt{ci}(.,.)$ has to converge on that input at some point, so

$$\exists p \; \forall^{\infty}_{I-1} j : \texttt{ci}(p_j, e_j) = p_{j+1} = p.$$

Now consider the set $L_I = \{e_0, e_1, \ldots, e_I\}$. Without loss of generality, we can assume that $|L_I| = 2k$ for some $k \in \omega$.
Now, take four different even numbers that have not appeared in the input, yet, i.e. $n_1, \ldots, n_4 \in \mathbb{N}_e \setminus L_I$. Then, the two sets $N_1 := L_I \cup \{n_1, n_2\}$ and $N_2 := L_I \cup \{n_3, n_4\}$ still have the same code p, as the learner $\texttt{ci}(.,.)$ will not change its mind on even numbers anymore. Furthermore, $|N_1| = |N_2| = 2(k+1)$. So, if we add the element $m = 2(k+1)+1$ to the sets N_1 and N_2, the sets appear to be in $\mathcal{C}$. However, these two sets, and in fact all sets of this form, have the same code $\texttt{ci}(p, m) = p'$. A contradiction.

So, we observed two things. As already mentioned, we cannot invert the lemma above. Secondly, having an iterative and a confident learner does not suffice to have a confident iterative learner. To see another example, we need the following property of CI classes.

Lemma 4.2.5: *Let $\mathcal{C} \in$ CI. Then, for every finite set $F \in \mathcal{C}$ we can compute its code, i.e. for certain input $(d_n)_n \in F$ one can find I effectively such that for some p*

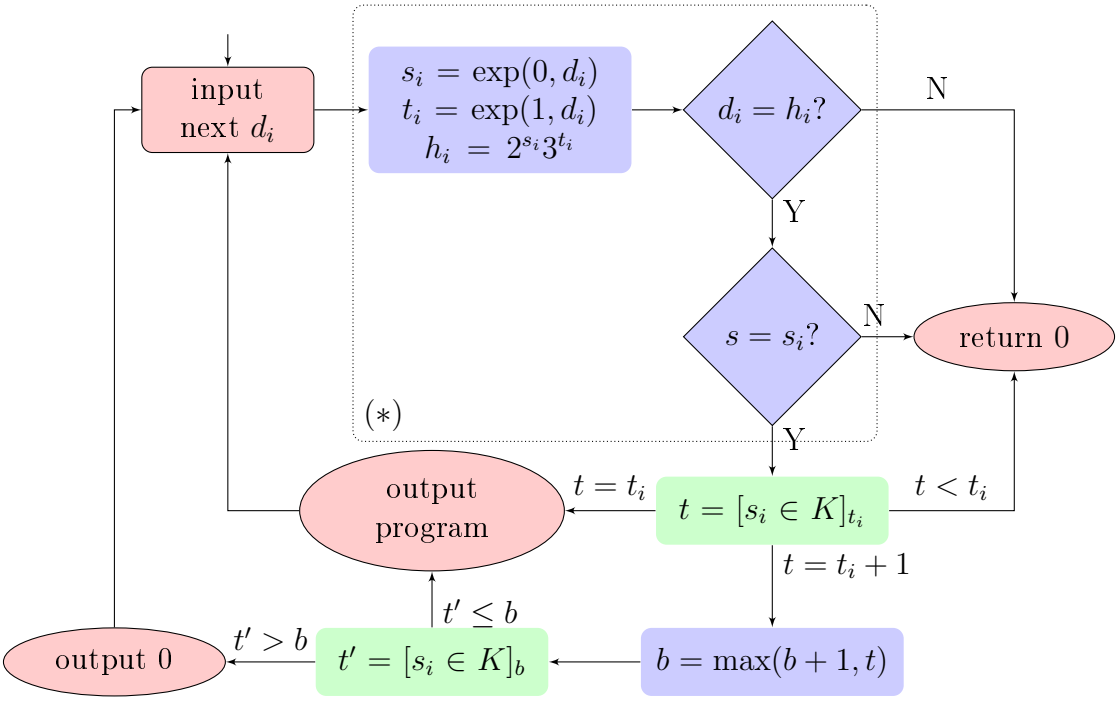
$$\forall i > I : \texttt{ci}(p_i, d_i) = p_{i+1} = p.$$

Proof. Let $\mathcal{C}$ be a CI class, learned by $\texttt{ci}(.,.)$, and $F = \{f_0, \ldots, f_n\} \in \mathcal{C}$. Now, we will feed $\texttt{ci}(.,.)$ the input $f_0, f_1, \ldots, f_n, f_n, f_n, \ldots$, abbreviated by $(f^*_m)_m$.
As the class is CI, $\texttt{ci}(.,.)$ will learn the set on this input, i.e. there exists some p and I such that $\texttt{ci}(p_j, f^*_j) = p_{j+1} = p$ for all $j > I$. Without loss of generality, let $I > n$. Since the elements of the input sequence do not change anymore, once $\texttt{ci}(.,.)$ repeats its output, it actually learned the set, since the next calculation is the same as the previous. I.e. at some point $I > n$ the computation outputs $\texttt{ci}(p_I, f^*_I) = \texttt{ci}(p_I, f_n) = p_I$, thus receives the same input again, namely p_I and f_n. Thus, $\forall j > I : \texttt{ci}(p_j, f^*_j) = \texttt{ci}(p_j, f_n) = p_j$. That I is the sought index. □

Example 4.2.6: Take the class $\mathcal{M} = \{A \mid \exists x, y : A \subseteq M_{x,y}\}$, where

$$M_{x,y} := \{2^x, 2^x 3, \ldots, 2^x 3^y : x \text{ is enumerated into } K \text{ in exactly } y \text{ steps}\}.$$

Again, by Lemma 4.2.2 the class is iteratively learnable.
In order to show that the class is confidently learnable, we will provide an explicit algorithm. To that attempt, recall the computable function $\exp(x, y)$ from Example 2.2.4. Also, let $[x \in K]_y$ output the number of steps $\leq y$ needed for x to be enumerated into K. If x is not enumerated into K in y steps, let that function output $y + 1$.
Now, let $(d_n)_n$ be some input for the algorithm. Upon starting, fix two global variables, namely $b = 0$ and $s = \exp(0, d_0)$.



We will prove that the algorithm works as supposed and with that explain it.
Let $(d_n)_n$ be some input. As already mentioned, we have two global variables. First, we have a counter b which will come to use later. Secondly, we have $s = \exp(0, d_0)$, which is the pivot program.
Now, once we receive the next d_i as input, we compute $h_i = 2^{\exp(0,d_i)}3^{\exp(1,d_i)}$ and check whether d_i is equal to that, i.e. whether d_i has the correct form $2^{\exp(0,d_i)}3^{\exp(1,d_i)}$. If not, we know that the input cannot belong to a set in the class, so we may as well as stop here and return 0.
If we do have the sought form, we ask whether or not the corresponding program s_i matches the program s. If not, we again are surely outside the class, so we return 0.
Otherwise, we compute $t = [s_i \in K]_{t_i}$ and then proceed depending on the outcome.

1. $t < t_i$. In this case, the amount of steps extends the minimal amount, so we do not have a subset of some $M_{x,y}$, rather a superset. So, we are surely not in the class and thus return 0.

2. $t = t_i$. In this case, we have a candidate for a class member. So, we output a program of the set $\{d_0, d_1, \ldots, d_i\}$, i.e. the set of input that we have gathered so far.

3. $t = t_i+1$. When we get to this case, we may have one of the two following situations. We may have a candidate, namely a proper subset of some $M_{x,y}$, or we may have a program that actually is never enumerated into K. Here, the counter comes to use. We set the counter b to the maximal value of the counters last value plus one, i.e. $b + 1$, or t. Then we ask, whether or not the program was enumerated into K in b steps. By doing so, we make sure to check the next, yet unchecked, step, to see whether or not we enumerate s into K in some later step, namely b. If so, we can output the program of $\{d_0, \ldots, d_i\}$, as we may have hit some proper subset of some $M_{x,y}$. In the other case, we output 0 and request the next input. By doing so, we ensure the machine to converge if the input does not belong to a set of the class.

We see that the algorithm works properly. Thus, the class is confidently learnable.
Lastly, we will show that this class is not CI. Assume it has a CI-learner $\texttt{ci}(.,.)$. For some $x \in \mathbb{N}$, consider the following computation. Let $(x \in K)_y$ be the abbreviation for the computable question, whether x is enumerated into K in at most y steps.

- Input $2^x, \ldots, 2^x$ until $\texttt{ci}(.,.)$ computably converges, see Lemma 4.2.5, to some p_1. Then, check if $(x \in K)_0$. If so, then $x \in K$. Otherwise, proceed with the next step.

- Continue to input $2^x3, \ldots, 2^x3$ until $\texttt{ci}(.,.)$ computably converges to some p_2. Then, check if $p_1 = p_2 \vee (x \in K)_1$. In the second case, again $x \in K$. In the first case, $W_{p_1} = W_{p_2}$. So, if $x \in K$, then the class members $\{2^x\}$ and $\{2^x, 2^x3\}$ would have the same code. A contradiction. Thus, $x \notin K$. Otherwise, proceed with the next step.

- Continue to input $2^x3^i, \ldots, 2^x3^i$ until $\texttt{ci}(.,.)$ computably converges to some p_{i+1}. Then, check if $p_i = p_{i+1} \vee (x \in K)_i$. If not, then again $x \notin K$ or $x \in K$, respectively. Otherwise, proceed with the next step.

This computation has to stop, as otherwise, as $p_i = p_{i-1}$ is one of the requirements, $\texttt{ci}(.,.)$ would not converge on the input sequence

$$2^x, \ldots, 2^x, 2^x 3, \ldots, 2^x 3, 2^x 3^2, \ldots, 2^x 3^{i-1}, 2^x 3^i, \ldots.$$

Thus, the computation stops for all x, and therefore we can solve the halting problem, a contradiction.

Remark 4.2.7: In further algorithms, we will reuse the flowchart in Example 4.2.6, and refer to the box $(*)$ by simply calling it "compute and check the form of d_i". Then, for some input d_i, the values s_i, t_i and h_i are defined in the same way, if not stated otherwise.

4.3 Some positive examples

So far, we have only seen classes that are not CI. To get rid of this peculiar situation, we will provide some examples of CI classes.

Example 4.3.1: For $n \in \mathbb{N}_{>0}$ let

$$\mathcal{C}_n := \{A : |A| \leq n\}.$$

We will show that $\mathcal{C}_n$ is CI. For some input $(d_n)_n$, the following algorithm will do the trick. Let p_0 be the program of the zero function. Then,

1. given d_i and p_i, check whether $\varphi_{p_i}(0) < n$ and if so, do
   a) if $\varphi_{p_i}(d_i + 1) = 0$, then p_{i+1} is the program of
   $$f(x) = \varphi_{p_i}(x) + \overline{\text{sgn}}(x) + \overline{\text{sgn}}(|x - (d_i + 1)|),$$
   b) else, $p_{i+1} = p_i$,
2. else, $p_{i+1} = p_i$.

The idea is simple. Position 0 of the program counts the different data that occurred so far. Once it reaches n, i.e. (1.) is not true anymore, the computation will converge, as either we have met some set in the class correctly, or we expanded it. Otherwise, while we do not have all the elements, in (1.a.) we check whether the datum is really new, i.e. $\varphi_{p_i}(d_i + 1) = 0$. If so, we add one to the counter, i.e. $f(0) = \varphi_{p_i}(0) + 1$, and mark the element as seen, i.e. $f(d_i + 1) = 1$. Then the algorithm outputs a code of this current characteristic function. Since this case can only occur finitely often, the algorithm will converge here, too.
In the end, we can rebuild the characteristic function of $S_{\mathcal{C}_n}$ via $\chi_{S_{\mathcal{C}_n}}(x) = \varphi_p(x + 1)$.

Although we have a restriction on the memory, we still are capable of keeping some in the first few positions of the code, which we may change only finitely many times.
The last example is of some finite nature, as all the sets do not expand some bound. However, we can find examples with a set of every size.

Example 4.3.2: Consider the class $\mathcal{C} := \{S \mid \exists x : S \subseteq S_x\}$ where

$$S_x := \{2^x, 2^x 3, \ldots, 2^x 3^y : y \leq x\}.$$

Let $(d_n)_n$ be some input sequence. Let p_0 be a program of the function f, where $f(0) = \exp(0, d_0)$ and $f(x+1) = 0$ for $x \geq 0$. So, we book the zeroth position for the pivot program and book the rest of the function for the characteristic function. Then, execute the following algorithm.

1. For some p_i and d_i, calculate and check the form of d_i. If it is valid, i.e. $d_i = 2^{s_i}3^{t_i}$, then
2. for $s = \varphi_{p_i}(0)$, check whether $s = s_i$,
3. if so, check whether $\varphi_{p_i}(d_i + 1) = 0$,
4. if so, check whether $t_i \leq s_i$,
5. if so, let p_{i+1} be a program of $f(x) = \varphi_{p_i}(x) + \overline{\text{sgn}}(|x - (d_i + 1)|)$.
6. In the else case of (3.) output $p_{i+1} = p_i$. In all remaining else cases, return 0.

To check its functionality, let $(d_n)_n$ be any input. For some input p_i and d_i, we first check the form and whether $s = s_i$. Then we check whether the datum is new and whether the condition $t_i \leq s_i$ is met. If all of this is true, then output a code of the corresponding characteristic function f. Else, one can return 0, as the element cannot belong to some set in the class. The only exception here is when the element is already seen, i.e. case (3.). Here the algorithm outputs the last hypothesis. Again, mind changes can only happen finitely often, thus the algorithm converges.
Again, we can recompute the set $S_{\mathcal{C}}$ via $\chi_{S_{\mathcal{C}}}(x) = \varphi_p(x+1)$.

The previous example can be considered in a more general setting.

Lemma 4.3.3: *Let $R(x, y)$ be a two place computable relation, such that for every x*

$$S_{R(x)} := \{2^x 3^y : R(x, y), y \in \mathbb{N}\}$$

is finite. Then the class $\mathcal{C}_R := \{S \mid \exists x : S \subseteq S_{R(x)}\}$ is CI.

Proof. Let $R(x, y)$ be a two place computable relation, such that all $S_{R(x)}$ are finite. Then, for some input sequence $(d_n)_n$ the following algorithm will learn the class $\mathcal{C}_R$ confidently iteratively. Let p_0 be a program of the function f, where $f(0) = \exp(0, d_0)$ and $f(x+1) = 0$ for $x \geq 0$. Then, execute the following algorithm.

1. For input d_i and p_i, calculate and check the form of d_i. If it is valid, i.e. $d_i = 2^{s_i}3^{t_i}$, then
2. for $s = \varphi_{p_i}(0)$, check whether $s = s_i$,
3. if so, check whether $\varphi_{p_i}(d_i + 1) = 0$,

4. if so, check whether $R(s_i, t_i)$,

5. if so, let p_{i+1} be a program of $f(x) = \varphi_{p_i}(x) + \overline{\text{sgn}}(|x - (d_i + 1)|)$.

6. In the else case of (3.) output $p_{i+1} = p_i$. In all remaining else cases, return 0.

For some input p_i and d_i, we first check the form and whether $s = s_i$. Then we check whether the datum is new and whether the condition $R(s_i, t_i)$ is met. If all of this is true, then output a code of the corresponding characteristic function f. Else, one can return 0, as the input cannot belong to some set in the class. The only exception here is when the element is already seen, i.e. case (3.), where the algorithm outputs the last hypothesis. Since every $S_{R(x)}$ is finite, the algorithm may only come to case (5.) finitely many times. If the input is not regular, at some time the algorithm will return 0. Thus, it works properly.
Again, one can recompute the set $S_{R(x)}$ via $\chi_{S_{R(x)}}(x) = \varphi_p(x + 1)$. □

As last act in this section, we will "repair" the class from Example 4.2.6 to be CI.

Example 4.3.4: Consider the class $\mathcal{M} = \{M_{x,y} : x, y \in \mathbb{N}\}$, where

$$M_{x,y} := \{2^x, 2^x 3, \ldots, 2^x 3^y : x \text{ is enumerated into } K \text{ in exactly } y \text{ steps}\}.$$

So, we do not allow any subsets of $M_{x,y}$. The following algorithm learns this class confidently iteratively. Let $(d_n)_n$ be some input sequence and let $p_0 = 1$ be some fixed dummy code.

1. For p_i and d_i, calculate and check the form of d_i. If it is valid, then

2. if $p_i = 1$, then, check whether $s_i = \exp(0, d_i)$ is enumerated into K in exactly $t_i = \exp(1, d_i)$ steps,

   a) if so, then p_{i+1} is a program of $\{2^{s_i}, 2^{s_i} 3, \ldots, 2^{s_i} 3^{t_i}\}$,

   b) else, $p_{i+1} = p_i$.

3. Else, $p_{i+1} = p_i$.

The algorithm simply waits for some input $2^x 3^y$, where x is enumerated into K in exactly y steps. Then it outputs some code of $\{2^x, 2^x 3, \ldots, 2^x 3^y\}$. The only memory used here, is whether $p_i = 1$, meaning that no such occurrence has been witnessed, yet.
To show that the algorithm works properly, let $(d_n)_n$ be an input sequence. Let $p_0 = 1$. At stage i, the algorithm receives p_i and d_i. If $p_1 = 1$, we check the form of d_i and whether $s_i = \exp(0, d_i)$ is enumerated into K in exactly $t_i = \exp(1, c_i)$ steps. If it turns out to be true, we output a program of the characteristic function of $\{2^{s_i}, 2^{s_i} 3, \ldots, 2^{s_i} 3^{t_i}\}$. Thus, we change the program and from now on, the algorithm does nothing. Otherwise, $p_{i+1} = p_i$.
In both cases the algorithm converges to some output, being correct on the regular input.

4.4 Classification Theorem

So far, we have gathered some examples of CI classes and such that are not. Next, we aim to find a classification of some confident iterative classes. To do so, we need an effective enumeration of the finite sets.

Definition 4.4.1: For $x \in \mathbb{N}$ consider the prime decomposition of x : $x = 2^{k_0}3^{k_1}\cdots p_n^{k_n}$. If $k_0 < k_1 < \cdots < k_n$, then the *x-th finite set* is $D_x := \{k_0, k_1, \ldots, k_n\}$. Otherwise, $D_x = \emptyset$.

Without further ado, we state the following theorem.

Theorem 4.4.2 (Classification Theorem)**:** *Let $\mathcal{C}$ be a class such that*

1. *$\mathcal{C}$ is a class of finite sets,*
2. *$\mathcal{C}$ is closed under subsets,*
3. *there exists an effective procedure which tells whether D_x is in $\mathcal{C}$ or not.*

Then, $\mathcal{C}$ is CI *iff $\mathcal{C}$ does not contain an infinite ascending chain.*

Proof. We will prove each direction separately.

$\Rightarrow$: By Lemma 4.2.3, if a class has a CI-learner, it cannot contain an infinite ascending chain.

$\Leftarrow$: Let $\mathcal{C}$ have no infinite ascending chain. Also, let $(d_n)_n$ be any input sequence. Then, the following algorithm will serve as a CI-learner. Let p_0 be a program of the zero function. For some input d_i and p_i, do

1. if $\varphi_{p_i}(d_i + 1) = 0$,
   a) collect all $\varphi_{p_i}(0) = n$ elements that appeared so far, i.e. $O_i = \{x : \varphi_{p_i}(x + 1) = 1\}$ with $|O_i| = n$,
   b) then check whether $O_i \cup \{d_i\} = D_{x'} \in \mathcal{C}$.
      i. If so, let p_{i+1} be a program of
      $$f(x) = \varphi_{p_i}(x)\,\mathrm{sgn}(x) + (\varphi_{p_i}(x) + 1)\overline{\mathrm{sgn}}(x) + \overline{\mathrm{sgn}}(|x - (d_i + 1)|).$$
      ii. else, return $p_{i+1} = 0$.
2. Else, $p_{i+1} = p_i$.

The only step that is not obviously computable, is finding O_i. Since the algorithm knows how many elements O_i has to contain, one can conduct the search effectively. Thus, all steps in the algorithm can be done effectively.
For some input $(d_n)_n$, the algorithm checks whether d_i is some new input, i.e. $\varphi_{p_i}(d_i + 1) = 0$. If so, it collects all the information so far, and then checks whether the corresponding set $D_{x'}$ belongs to the class. If it does, it outputs a program of its characteristic function, with an additional update on the first argument. Else,

i.e. if the algorithm witnesses $D_{x'}$ not belonging to the class, it returns 0.
So, the only case where the machine could possibly change its mind infinitely many times, is when confirming $D_x \in \mathcal{C}$ and thus change the program. However, since the class has no infinite ascending chain, this cannot happen. □

5 Further Topics

In order to close this thesis off, we will investigate some more concepts connected to the confident iterative learner.

5.1 Stronger Confidence

In this section we will expand the idea of the confident learner. As discussed, the confident learner earns its name from confidently outputting some code, even if the input does not belong to any set in the class. However, we can request it to output some fixed dummy code, i.e. -1, if the learner detects such a case. With that idea in mind, we introduce the very confident learner. A similar attempt can be found in [12].

Definition 5.1.1 (vC)**:** A class $\mathcal{C}$ is *very confidently learnable* iff there exists a machine which converges on any input sequence, is correct on the regular one and outputs some dummy code if the input does not belong to any set in $\mathcal{C}$, i.e.

$$\exists \text{comp. } \mathtt{vc}\ \forall (d_n)_n : \exists e \Big(\big(\forall^\infty i : \mathtt{vc}(d_0, \dots, d_i) = e\big) \wedge \big((d_n)_n \in D_{\mathcal{C}} \Rightarrow W_e = D_{\mathcal{C}}\big) \Big) \wedge \\ \forall j \Big(\big(\forall S \in \mathcal{C} : \{d_0, \dots, d_j\} \neq S\big) \Rightarrow \big(\mathtt{vc}(d_0, \dots, d_j) = -1\big) \Big).$$

Such a learner is called *very confident learner*.

Remark 5.1.2: Classes in vC cannot have any infinite sets. Let $\mathcal{U}$ be a class with an infinite set $U = \{u_0, u_1, \dots\}$. Notice that confident classes cannot contain any infinite ascending chain, as otherwise one could conduct a proof similar to that of Lemma 4.2.3. So, for some n, $\forall_n^\infty m : \{u_0, \dots, u_m\} \notin \mathcal{U}$. Then, $\forall_n^\infty i : \mathtt{vc}(u_0, \dots, u_i) = -1$, however, vc should converge to some code of U.

We can take this even a step further, by not only requesting the learner to output some dummy code, but rather to actually stop once it detects some irregular input. So, we introduce the strongly confident learner.

Definition 5.1.3 (sC)**:** A class $\mathcal{C}$ is *strongly confidently learnable* iff there exists a machine which converges on any input sequence and is correct on the regular input and halts when it detects some input not belonging to any set in $\mathcal{C}$, i.e.

$$\exists \text{comp. } \mathtt{sc}\ \forall (d_n)_n : \exists e \Big(\big(\forall^\infty i : \mathtt{sc}(d_0, \dots, d_i) = e\big) \wedge \big((d_n)_n \in D_{\mathcal{C}} \Rightarrow W_e = D_{\mathcal{C}}\big) \Big) \wedge \\ \forall J \Big(\big(\forall S \in \mathcal{C} : \{d_0, \dots, d_J\} \neq S\big) \Rightarrow \big(\forall j \geq J : \mathtt{sc}(d_0, \dots, d_j) = -1\big) \Big).$$

Such a learner is called *strongly confident learner*.

The definitions directly imply the following corollary.

Corollary 5.1.4: sC $\subseteq$ vC $\subseteq$ CFD.

To see that the inclusions are proper, consider the following two examples.

Example 5.1.5: With K'' being the second jump of the halting set K, let

$$\mathcal{C} = \{\{x\} : x \in \mathbb{N} \wedge x \in K''\}.$$

Then, $\mathcal{C}$ is obviously in CFD, via the confident learner $\texttt{cfd}(d_0, \ldots, d_i)$ which outputs the code of $\{d_0\}$.
However, assume $\mathcal{C} \in$ vC. Then there exists some $\texttt{vc}(.)$ learning this class. For some $x \in \mathbb{N}$, consider the input sequence $x, x, x, \ldots$. Then,

$$x \notin K'' \Rightarrow \forall N : \texttt{vc}(x^N) = -1 \Rightarrow \forall N\ \exists n : (n > N \wedge \texttt{vc}(x^n) = -1),$$
$$x \in K'' \Rightarrow \exists N\ \forall n : (n > N \Rightarrow \texttt{vc}(x^n) \neq -1).$$

Thus, $x \in K'' \Leftrightarrow \exists N\ \forall n : (n > N \Rightarrow \texttt{vc}(x^n) \neq -1)$. As the latter one is in Σ_{1+1}, by Theorem 2.4.5, we get $K'' \leq_m K'$. A contradiction.

Example 5.1.6: For two different numbers $x, y \in \mathbb{N}$, let $\mathcal{C} = \{\{x, y\}\}$. $\mathcal{C}$ is obviously in vC. However, it is not in sC. Assume the opposite, with a learner $\texttt{sc}$. Then for some input sequence $x, y, y, \ldots$ the machine $\texttt{sc}$ will see that $\{x\}$ is not in the class, thus output -1 for everything that is to come, i.e. for $J = 0$

$$\{x\} \neq \{x, y\} \Rightarrow \forall j : \texttt{sc}(x, y^j) = -1.$$

This is obviously the wrong behaviour, since $\{x, y\}$ is in the class.

Altogether, we get a stronger version of Corollary 5.1.4.

Corollary 5.1.7: sC $\subsetneq$ vC $\subsetneq$ CFD.

The strongly confident learner may seem awfully weak compared to the very confident one. However, its weakness in the last example originates from the fact, that the class was not closed under subsets. When combining those attempts with the iterative learner, we will see some astonishing facts. Therefore, let us combine those two in the obvious way.

Definition 5.1.8 (vCI)**:** A class $\mathcal{S}$ is *very confidently iteratively learnable* iff there is a learner $\texttt{vci}(.,.)$ learning $\mathcal{S}$ very confidently and iteratively.
Such a learner $\texttt{vci}(.,.)$ is called *very confident iterative learner*.

Definition 5.1.9 (sCI)**:** A class $\mathcal{S}$ is *strongly confidently iteratively learnable* iff there is a learner $\texttt{sci}(.,.)$ learning $\mathcal{S}$ strongly confidently and iteratively.
Such a learner $\texttt{sci}(.,.)$ is called *strongly confident iterative learner*.

Immediately, we get the following counterpart of Corollary 5.1.4.

Corollary 5.1.10: sCI $\subseteq$ vCI $\subseteq$ CI.

However, these classes do behave slightly different, as we will show next. To do so, we need some auxiliary corollary.

Corollary 5.1.11: *Let $\mathcal{C} \in$ vCI via some $\texttt{vci}(.,.)$, and $(d_n)_n$ be some input. If $\exists i : \texttt{vci}(p_i, d_i) = -1$, then $\forall j \geq i : \texttt{vci}(p_j, d_j) = -1$.*

Proof. Let $\mathcal{C} \in$ vCI via $\texttt{vci}(.,.)$, $(d_n)_n$ be some input and let i such that $\texttt{vci}(p_i, d_i) = -1$. Assume that $\exists j > i : \texttt{vci}(p_j, d_j) \neq -1$. Then,

$$S_1 = \{d_0, \ldots, d_i\} \subsetneq \{d_0, \ldots, d_j\} = S_2$$

as $S_1 \notin \mathcal{C}$, while $S_2 \in \mathcal{C}$. As $\mathcal{C} \in$ vCI $\subseteq$ CI, it cannot contain any infinite ascending chain. So, there is some finite $S_3 \supsetneq S_2$ such that $S_3 \notin \mathcal{C}$. Now, consider the following two input sequences

$$d_0, \ldots, d_i, d_{i+1}, \ldots, d_j, d^*_{j+1}, \ldots,$$
$$e_0, \ldots, e_m, d_{i+1}, \ldots, d_j, d^*_{j+1}, \ldots,$$

where $d^*_k \in S_2 \setminus S_1$, such that $\{d^*_{j+1}, \ldots\} = S_2 \setminus S_1$, and $e_k \in S_3$, such that $\{e_0, \ldots, e_m\} = S_3$ and $\texttt{vci}(p_m, e_m) = -1$. Then, as the input is the same after the occurrence of -1, $\texttt{vci}(.,.)$ converges to the same program on both inputs. But, since the first input belongs to $S_2 \in \mathcal{C}$ and the second to $S_3 \notin \mathcal{C}$, this is a contradiction. □

In particular, the proof of the last lemma shows that every vCI class has to be closed under subsets.

Corollary 5.1.12: *Every class $\mathcal{C} \in$ vCI is closed under subsets.*

Lemma 5.1.13: *Let $\mathcal{C}$ be a class. Then, $\mathcal{C} \in$ vCI if and only if $\mathcal{C} \in$ sCI.*

Proof. The right to left direction follows from Corollary 5.1.10.
For the other direction, let $\mathcal{C} \in$ vCI. Then, for some input sequence $(d_n)_n$ and the same starting program p_0, let $\texttt{sci}(p_i, d_i) = \texttt{vci}(p_i, d_i)$.
It only remains to prove that for any J

$$\big(\forall S \in \mathcal{C} : \{d_0, \ldots, d_J\} \neq S\big) \Rightarrow \big(\forall j \geq J : \texttt{sci}(p_j, d_j) = p_{j+1} = -1\big).$$

Via Corollary 5.1.11, for any $J \in \mathbb{N}$,

$$\big(\forall S \in \mathcal{C} : \{d_0, \ldots, d_J\} \neq S\big) \Rightarrow \big(\texttt{vci}(p_J, d_J) = -1\big) \Rightarrow \big(\forall j \geq J : \texttt{vci}(p_j, d_j) = -1\big).$$

Since $\texttt{vci} = \texttt{sci}$, we have the sought learner. □

This lemma is not too surprising. Since the only information on the previous calculations and inputs has to be coded into the output in some form, outputting -1 deletes all that information. Thus, it is the same as stopping.

Theorem 5.1.14: *Let $\mathcal{C}$ be a class of finite sets. Then $\mathcal{C} \in$ vCI if and only if $\mathcal{C} \in$ CI, $\mathcal{C}$ is closed under subsets, and there is a decision procedure telling whether or not $D_x \in \mathcal{C}$.*

Proof. We will prove each direction separately.

⇒: If $\mathcal{C} \in \text{vCI}$, then by Corollary 5.1.10 $\mathcal{C} \in \text{CI}$, and by Corollary 5.1.12 the class has to be closed under subsets.
For $x \in \mathbb{N}$, let $D_x = \{k_0, \ldots, k_n\} \neq \emptyset$. Considering the input $k_0, \ldots, k_n, k_n, \ldots$, we can compute the converging point I effectively. If we look at the output of $\texttt{vci}(p_I, d_I) = p$, we know that $D_x \in \mathcal{C} \Leftrightarrow p \neq -1$.

⇐: For the other direction, let $\mathcal{C} \in \text{CI}$, via $\texttt{ci}(.,.)$ and the starting program $\tilde{p_0}$, as well as some translation s, see Remark 4.1.4. Let $\mathcal{C}$ be closed under subsets and have a decision procedure for $D_x \in \mathcal{C}$. For a finite set A, let $c(A)$ be some coding of finite sets. Then, for the starting program $p_0 = 2^{c(\emptyset)}3^{\tilde{p_0}}$,

$$\texttt{vci}(p_i, d_i) = \begin{cases} 2^{c(\{d_0,\ldots,d_i\})}3^{\texttt{ci}(p_i,d_i)}, & \text{if } \{d_0, \ldots, d_i\} \in \mathcal{C} \wedge p_i \neq -1, \\ -1, & \text{else.} \end{cases}$$

will do the trick, where $s(\exp(1, p))$ is the actual program.
Let $(d_n)_n$ be some input. For input d_i and p_i, the machine checks whether or not $c^{-1}(\exp(0, p_i)) \cup \{d_i\} = \{d_0, \ldots, d_{i-1}, d_i\} \in \mathcal{C}$. If not, it outputs -1, and after that never changes its mind again. Otherwise, it will compute the code that $\texttt{ci}(p_i, d_i)$ would have, and output $2^{c(\{d_0,\ldots,d_i\})}3^{\texttt{ci}(p_i,d_i)}$. As $\texttt{ci}(.,.)$ learns this class, and as it cannot contain an infinite ascending chain, see for the Classification Theorem 4.4.2, the machine may only change its mind finitely often.
Again, as $\texttt{ci}(.,.)$ learns the class correctly, it will converge on any $(d_n)_n$ to some $\tilde{p}$. If $(d_n)_n \in S_{\mathcal{C}}$, then the program $\tilde{p}$ has to be correct, i.e. $W_{s(\tilde{p})} = S_{\mathcal{C}}$. If we compute $s(\exp(1, p)) = s(\tilde{p})$, we get the computable $s(\exp(1, .))$ as translation.
Thus, the algorithm will always converge and behave correctly. □

As we have seen in Example 4.3.4, CI classes do not need to be closed under subsets. Thus, we get the confident iterative counterpart of Corollary 5.1.7.

Theorem 5.1.15: $\text{sCI} = \text{vCI} \subsetneq \text{CI}$.

5.2 Hypothesis Space

Until now, we did not care too much about the hypotheses, as long as they gave some sort of computable information on the set learned. However, it did prove to be inconvenient to code all the information into the hypotheses itself. We also witnessed that for certain problems, certain hypotheses were more comfortable to use than others. In this section we will drill down onto that. To do so, we will equip the classes with some hypothesis spaces. To be capable of that, we need to introduce the notion of uniformly indexed families, see for example [1] or [9].

Definition 5.2.1 (Uniformly Indexed Family)**:** A class $\mathcal{C}$ is given by a uniformly indexed family if there exists a two-place, $\{0, 1\}$-valued, computable function L, such that

1. $L(e,x) = L_e(x) = \begin{cases} 1, & x \in L_e, \\ 0, & x \notin L_e. \end{cases}$
2. $\forall e : L_e \in \mathcal{C}$,
3. $\forall L \in \mathcal{C}\ \exists e : L = L_e$.

We will call such a class $\mathcal{C}$ *indexed class* for short.

Notation 5.2.2: With $\mathcal{C} = \{L_e : e \in \mathbb{N}\}$, where $L_e = \{x : L_e(x) = 1\}$, we will denote indexed classes.

Remark 5.2.3: Since we are only considering classes without the empty set, we additionally demand that none of the L_e is empty.

So, indexed classes provide an effective way of numbering all the sets from the class and checking whether or not $x \in L_e$ for some x, e. With this, we can introduce the idea of the hypothesis space.
Given some indexed class $\mathcal{C}$, we will learn it with respect to some *hypothesis space* $\mathcal{H} = \{H_i : i \in \mathbb{N}\}$, which itself is an indexed class. Learning a set $C \in \mathcal{C}$ here means that the learning machine converges to some i, such that $C = H_i$. This kind of learning can be done in several ways. In [9], we can find three different ideas, how such learning can be done. We can learn $\mathcal{C}$ with respect to $\mathcal{H}$ in the following ways.

- Exactly: $\mathcal{C}$ is *exactly learnable*, if $\mathcal{H} = \mathcal{C}$, using the same numbering.
- Class Preservingly: $\mathcal{C}$ is *class preservingly learnable*, if $\mathcal{H} = \mathcal{C}$.
- Class Comprisingly: $\mathcal{C}$ is *class comprisingly learnable*, if $\mathcal{H} \supseteq \mathcal{C}$.

Contrary to the widely used E, ϵ and C, we will use the prefixes E, CP and CC, respectively.

Remark 5.2.4: Since the hypotheses are restricted now, we will use an "initial state" as first hypothesis p_0, rather than a proper hypothesis, see [10]. However, the learner may never output this state. This state signalises that this is the first step of the computation.

In order to investigate learning with respect to some hypothesis space, let us fix one special hypothesis space.

Definition 5.2.5: Let $[i]_2$ be the binary expression of i. Let $\mathcal{B} = \{B_i : i > 0\}$, where

$$B_i(x) = ([i]_2)(x) = \begin{cases} 1, & \text{if } [i]_2 \text{ is 1 at position } x, \\ 0, & \text{else.} \end{cases}$$

Remark 5.2.6: Technically, we should define $\mathcal{B}$ as $\tilde{\mathcal{B}} = \{B_{i+1} : i \in \mathbb{N}\}$. However, they are the same, but the first version is easier to deal with.

So, $\mathcal{B}$ is an indexation of all finite sets, except for the empty set. With that, we can state the following result.

Lemma 5.2.7: *Let $\mathcal{C} = \{C_i : i \in \mathbb{N}\}$ be an indexed class of finite sets, which is closed under subsets, with a decision procedure for $D_x \in \mathcal{C}$. Then*

$$\mathcal{C} \in \text{ccCI} \Leftrightarrow \mathcal{C} \text{ contains no infinite ascending chain} \Leftrightarrow \mathcal{C} \in \text{CI}.$$

Proof. Since the second equivalence is exactly the Classification Theorem 4.4.2, we only need to show the first one.

$\Rightarrow$: Conducting some similar proof as in the proof of Lemma 4.2.3, we can see that this direction is true.

$\Leftarrow$: By observing the proof of the Classification Theorem 4.4.2, we can see that we could change the output to its respective counterpart in $\mathcal{B}$, and still receive the natural bound on the amount of the elements. Thus, we could conduct the same proof, here. □

Obviously, eCI $\subseteq$ cpCI $\subseteq$ ccCI. As last act in this thesis, we will provide examples witnessing that these inclusions are proper.

Example 5.2.8: Consider the class $\mathcal{C} = \{C_{x,y} : x, y \in \mathbb{N}\}$, where

$$C_{x,y} = \begin{cases} \{p\}, & x \notin K \text{ in } y \text{ steps}, \\ \{x\}, & x \in K \text{ in } y \text{ steps}. \end{cases}$$

where p is a program of some always undefined function.
By outputting 2^{d_0} on the input $(d_n)_n$, we can see that $\mathcal{C} \in$ ccCI with respect to $\mathcal{B}$.
Assume it is in cpCI with respect to some $\mathcal{H}$. For $x \in \mathbb{N}$ consider the input sequence $x, x, \ldots$. At some computable stage N, the learner will converge to some p_x, thus

$$\forall n > N : \texttt{cpci}(p_n, x) = p_n = p_x.$$

Then,

$$x \in K \Rightarrow H_{p_x} = \{x\},$$
$$x \notin K \Rightarrow H_{p_x} = \{p\} \neq \{x\}.$$

The second property originates from the fact, that there is no set in $\mathcal{H}$ representing $\{x\}$, since $x \notin K$. For any i and x, the question whether $H_i = \{x\}$ is computable, as all H_i have exactly one element. Thus, the canonical search for y such that $H_i(y) = 1$ will terminate.
Thus, $x \in K \Leftrightarrow H_{p_x} = \{x\}$, with the latter being computable. A contradiction.

The next example is inspired by an example in [9] and [15].

Example 5.2.9: Consider the class $\mathcal{L} = \{L_{i,j} : i, j \in \mathbb{N}\}$, with

$$L_{i,j} = \begin{cases} \{2^i 3^n : n \in \mathbb{N}\}, & (\varphi_i(i))_j \uparrow, \\ \{2^i 3^n : n \leq k\}, & (\varphi_i(i))_j \downarrow \text{ in } k \text{ steps}. \end{cases}$$

This class is in cpCI via the following algorithm. For input $(d_n)_n$ and for some starting program $p_0 = (-1, 0)$

- for d_i and p_i, check the form of d_i. If it is valid, i.e. $d_i = 2^{s_i}3^{t_i}$, then
- check whether $p_i = (s_i, x)$ or $p_i = (-1, 0)$. If $x \neq 0$, then output $p_{i+1} = p_i$. Else,
- check whether $(\varphi_{s_i}(s_i))_{t_i} \downarrow$, and
  - if not, output $(s_i, 0)$,
  - if so, output (s_i, t_i).

One can easily see that the algorithm works correctly. Let $(d_n)_n$ be some input, and let p_i and d_i be the current input. Then, after checking the form, we check for $p_i = (s_i, x)$. If $x > 0$, then the algorithm already witnessed $\varphi_{s_i}(s_i) \downarrow$, so we output the last hypothesis. Else, even if $p_i = (-1, 0)$, we check whether $(\varphi_{s_i}(s_i))_{t_i} \downarrow$. If so, we output (s_i, t_i) to mark that event as witnessed. If not, we output $(s_i, 0)$.
The algorithm will output $(y, 0)$ as long as it does not witness $\varphi_y(y)$ to be computable. Then, it will change its mind. Thus, it converges correctly.
Assume now, that $\mathcal{L} \in$ ECI via $\texttt{eci}(.,.)$. Let $i \in \mathbb{N}$, and consider some canonical input $(d_n)_n \in L_{i,0}$, i.e. $d_j = 2^i3^j$. On that input, $\texttt{eci}(.,.)$ will converge to some (i, j), i.e. $\forall^\infty_m n : \texttt{eci}(p_n, d_n) = p_n = (i, j)$. Now, we have to distinguish between the following two cases.

1.C.: $2^i3^{j+1} \notin L_{i,j}$. Then, due to the definition, the set $L_{i,j}$ is finite. Thus, $\varphi_i(i) \downarrow$.

2.C.: $2^i3^{j+1} \in L_{i,j}$. In this case, the set is infinite, due to the definition. Now we have to make a case distinction, where we compare j to the converging point m.

2.1.C.: $j \geq m$. We claim that $\varphi_i(i) \uparrow$. Assume the opposite, then there exists some n such that $L_{i,n}$ is finite. But $n > j \geq m$, so the learner converged too early and to the wrong hypothesis. A contradiction.

2.2.C.: $j < m$. Now, test whether $2^i3^{m+1} \in L_{i,m}$. If not, then the set is finite, thus $\varphi_i(i) \downarrow$. Else, the set is infinite again, thus $L_{i,m} = L_{i,0}$. Again, we claim that $\varphi_i(i) \uparrow$. Assume the opposite, then there exists some n such that $L_{i,n}$ is finite. Since $n > m$, the hypothesis would be wrong again. A contradiction.

As all the steps are computable, we can compute the halting problem, a contradiction.

Altogether, we get the following result.

Theorem 5.2.10: ECI $\subsetneq$ CPCI $\subsetneq$ CCCI.

5.3 Further Research

By now, we have obtained some classification for the confident iterative classes. However, we considered only a very restricted type of classes. So, obviously, the next step would be to widen that restriction, and obtain some more general classification. However, this turns out to be quite a hard problem, see [10]. Also, taking classes with possibly infinite sets into consideration would be a natural next attempt. One may even try to capture learning of infinite sets using finite sets only.

Also, since we only considered learning from text, one may be tempted to investigate the behaviour of the very same learner, when learning from an informant. The necessary definitions for this can be found in [16], for example.
On the other hand, it may be worth a try to investigate the idea of the very confident learner further. It does seem like a natural idea to be able to detect unrelated information. Also, combining the confident iterative learner with some more types of learning, i.e. monotonic, consistent or conservative learning, as proposed for example in [10], [13] or [14], or investigating the learning with respect to some hypothesis space more may prove useful. Ideas, what the further investigation of the hypothesis space could look like, can be found for instance in [9] or [14].

Bibliography

[1] D. Angluin, *Inductive inference of formal languages from positive data*. Information and Control 45 (**1980**), 117-135. doi.org/10.1016/S0019-9958(80)90285-5

[2] L. Blum and M. Blum, *Toward a mathematical theory of inductive inference*. Information and Control 28 (**1975**), 125-155. doi.org/10.1016/S0019-9958(75)90261-2

[3] J. Case and C. Smith, *Comparison of identification criteria for machine inductive inference*. Theoretical Computer Science 25 (**1983**), 193-220. doi.org/10.1016/0304-3975(83)90061-0

[4] S. B. Cooper, *Computability Theory*. Chapman & Hall, **2004**.

[5] E. Fokina, Script on *Computability Theory*. **2016**.

[6] Z. Gao and F. Stephan, *Confident and consistent partial learning of recursive functions*. Theoretical Computer Science 558 (**2014**), 5-17.

[7] E. M. Gold, *Language identification in the limit*. Information and Control 10 (**1967**), 447-474. doi.org/10.1016/S0019-9958(67)91165-5

[8] V. S. Harizanov, *Inductive Inference Systems for Learning Classes of Algorithmically Generated Sets and Structures*. Induction, Algorithmic Learning Theory, and Philosophy (**2007**), 27-54.

[9] S. Jain, *Hypothesis spaces for learning*. Information and Computation 209 (**2011**), 513-527. doi.org/10.1016/j.ic.2010.11.016

[10] S. Jain, S. Lange and S. Zilles, *Consistent and conservative iterative learning*. Technical Report **2007**.

[11] S. Jain, S. E. Moelius III and S. Zilles, *Learning without coding*. Theoretical Computer Science 473 (**2013**), 124-148. doi.org/10.1016/j.tcs.2012.10.011

[12] K. P. Jantke, *Reflecting and self-confident inductive inference machines*. Algorithmic Learning Theory 1995 (**1995**), 282-297.

[13] S. Lange and T. Zeugmann, *Incremental learning from positive data*. Journal of Computer and System Sciences 53 (**1996**), 88-103.

[14] S. Lange and T. Zeugmann, *Language learning in dependence on the space of hypotheses*. Proceedings of the Sixth Annual Conference on Computational Learning Theory (**1993**), 127-136.

[15] S. Lange and T. Zeugmann, *Learning recursive languages with bounded mind changes.* International Journal of Foundations of Computer Science (**1993**), 157-178.

[16] F. Stephan, Script on *Einführung in die Lerntheorie.* **2002/2003**.

[17] J. Teutsch and M. Zimand, *On approximate decidability of minimal programs.* Transactions on Computational Theory (**2015**).