Analyzing Non-deterministic Computable Aggregations

Luis Garmendia[1(✉)], Daniel Gómez[2(✉)], Luis Magdalena[3(✉)], and Javier Montero[4(✉)]

[1] Facultad de Informática, Universidad Complutense de Madrid, Madrid, Spain
lgarmend@fdi.ucm.es
[2] Facultad de Estudios Estadísticos, Universidad Complutense de Madrid, Madrid, Spain
dagomez@estad.ucm.es
[3] ETSI Informáticos, Universidad Politécnica de Madrid, Madrid, Spain
luis.magdalena@upm.es
[4] Facultad de Ciencias Matemáticas, Universidad Complutense de Madrid, Madrid, Spain
monty@mat.ucm.es

Abstract. Traditionally, the term aggregation is associated with an aggregation function, implicitly assuming that any aggregation process can be represented by a function. However, the concept of computable aggregation considers that the core of the aggregation processes is the program that enables it. This new concept of aggregation introduces the scenario where the aggregation can even be non-deterministic. In this work, this new class of aggregation is formally defined, and some desirable properties related with consistency, robustness and monotonicity are proposed.

Keywords: Aggregation · Computable aggregation · Nondeterministic aggregation

1 Introduction

Aggregation is a fundamental part of any decision, compression or summarization process of complex information [1–4]. For many years, aggregation has become one of the most relevant topics in soft computing, with multiple applications to decision making, artificial intelligence, data science, and image processing among many others. Aggregation processes have been associated in literature with aggregation functions. An aggregation process was usually represented by means of a function or a family of functions so that the aggregation result associated to a vector of elements was obtained through the image of the vector by the function.

Supported by PGC2018-096509-B-I00 research national project.

M.-J. Lesot et al. (Eds.): IPMU 2020, CCIS 1238, pp. 551–564, 2020.
https://doi.org/10.1007/978-3-030-50143-3_43

However, this association between aggregation processes and functions was broken in [5] with the definition of computable aggregations. In that paper, the authors put the emphasis in the programs enabling the aggregation processes, not necessarily being expressed in funct terms of functions. In that sense, the core of an aggregation process is the program allowing its computation, and not only the function that describes it. Given a function that describes an aggregation process, it can be implemented in many ways and the way in which this process is carried out is relevant. This new idea of aggregation, allow us to classify classical aggregation operators according to their algorithmic complexity [5], or to classify aggregation functions according to new ideas of recursivity [6,7] in terms of the programs instead of the functions.

The rupture between functions and aggregation operators opens the domain of aggregation processes to a field not yet analyzed in this discipline: nondeterministic computable aggregations. Aggregation processes where the same input can produce different outputs. This type of aggregation is very common in statistics, where, due to the volume of information to be processed, it is frequent to choose a representative sample on which the aggregation is operated. Obviously, replicating the process does not imply obtaining the same sample and consequently the result can change. Obviously, these types of aggregation processes can never be modeled by functions, as a result of the intrinsic definition of function. But leaving the well known arena of functions, studying the desirable properties for these new concept of aggregations represents a significant challenge.

The present paper focuses on the definition of properties related to robustness, stability, boundary conditions, and monotony, on nondeterministic computable aggregations.

2 Preliminaries

2.1 Aggregation Operators

Aggregation is a fundamental part of science. The process of aggregating the information is a key tool for most knowledge based systems. In general, we can say that aggregation has the aim of merging different pieces of information to come to a conclusion or a decision. Several research communities consider this kind of tools, such as multi-criteria community, decision-sensor fusion community, decision making community, data mining community, among many others.

Although this is not a necessary assumption, aggregation operators [4,8–11] are associated to the use of membership functions, and this is the reason why they are usually defined as follows:

Definition 1 [12]**.** *An aggregation operator is a mapping* $Ag : [0,1]^n \to [0,1]$ *that satisfies:*

1. $Ag(0,0...,0) = 0$ *and* $Ag(1,1,..,1) = 1$.
2. *Ag is monotonic.*

Let us observed that this definition presents an aggregation operator as a function that is linked to the value of n. There is a different function for each n. Other authors (see [12] for example) present an aggregation operator as a function that considers any cardinality for the set of items to be aggregated, defining Ag as a function $Ag : \cup_n [0,1]^n \rightarrow [0,1]]$. It is also possible (see for example [13–15]) to define the concept of family of aggregation functions as a set $\{Ag_n\}$ assuming some additional constraints for the relations between functions Ag_k, Ag_l of different cardinality.

The original definition of aggregation functions work on the unit interval. This classical definition has been extended to a more general class of situations replacing the lattice $[0,1]$ into a more general scenario $\mathcal{T}$. Another extension of aggregation operators was done by relaxing the monotonicity or boundary conditions. One of these new classes of aggregation operators are the pre-aggregation functions in which the concept of directional monotonicity was introduced [16]. These pre-aggregations could be extended replacing the unit interval for a general T as it is done with the classical aggregation functions. We can find some interesting studies in this line in [17].

It is even possible in some cases to define aggregation processes going beyond functions by considering methods that do not match with the concept of function. To analyze this option let us first remind the concept of Computable aggregation, as well as that of function.

2.2 Computable Aggregation

In [5], it was introduced the concept of computable aggregation focusing on the idea that in aggregation processes we should pay our attention in the program that makes possible the aggregation instead of a generic function that has not yet been implemented.

Would it make sense to talk about an aggregation process where the considered function could not be implemented?. From the actual definition of aggregation operator the answer is yes of course. Or even more, given the same aggregation function, it is not equivalent at all to implement it in one way or another. And therefore, it would make sense to analyze the properties of the implementation (program), although from the functional point of view both implementations coincide. The main contribution in [5] was to separate the strong association that existed between "aggregation processes" and explicit functions.

In order to formally introduce computable aggregations it is necessary first to introduce what we understand by a program, a list and/or an algorithm.

Definition 2. *A list L is an abstract data type (ADT) that represents a sequence of values. A list can be defined by its behavior, and its implementation must provide at least the following operations: test whether a list is empty, add a value, remove a value, and compute the length of a list (number of elements).*

A list can be defined under a template data type. For example, a list $L<[0,1]>$ is a list of values in the space $X = [0,1]$.

Another definition that we need to give here is what we understand by algorithm and computer program. These concepts are necessary to formally define computable aggregations.

Definition 3 [18]**.** *In mathematics and computer science, an* **algorithm** *is a self-contained step-by-step set of operations to be performed.*

Algorithms can be expressed in many kinds of notation, including natural languages, pseudocode, flowcharts, drakon-charts, programming languages or control tables.

Definition 4 [19]**.** *A computer program, or simply a* **program***, is a sequence of instructions written to perform a specified task on a computer.*

Now, we are able to define the concept of a computable aggregation as it was defined in [5].

Definition 5. *Let L<T> be a finite and non-empty list of n elements with type T. A* **computable aggregation** *is a program P that transform the list L<T> into an element of T.*

3 Non Deterministic Computable Aggregations

It is important to emphasize that the computable aggregation paradigm broadens the set of possible aggregation methods, being in this case not limited to those methods for which there is a function that explains the aggregation process.

In [7], it was demonstrated that any classical aggregation process (aggregation operators, pre-aggregations, fusion functions or extended fusion functions) can be implemented by an algorithm and by a program. However the opposite is not always true.

As was mentioned in [5], given an aggregation operator Ag in any of the previous settings, it is possible to build a program P that for any list L of T, $Ag_{|L|}(L) = P(L)$.

But let us note that the opposite is not always true. Given a program P, there exist some situations in which it is not possible to build a function associated to the program P.

Example 1. Let's consider a population X of size n ($X = \{x_1, \ldots, x_n\}$). Suppose that the objective of the aggregation process is to estimate the average value of X. If n is too large considering the available time for computing the aggregation, a reasonable solution would be to estimate the average value through statistical sampling. In such a situation, k elements of the population will be drawn at random to further calculate the average, i e., given the set $\{x_1, ..x_n\}$, we choose $\{x_{i1}, \ldots, x_{ik}\}$ at random, computing the arithmetic mean of these k elements. Obviously, this aggregation process can't be defined by means of an explicit function since the same input does not always produce the same output. This is a computable aggregation that can't be modeled as an aggregation function, and it is important to integrate such a situation in aggregation processes.

From previous example we can distinguish between those computable aggregations in which there exist an explicit function, from those in which this explicit function does not exist. Formally we can partition the set of computable aggregation programs into two groups. In order to establish this partition we will introduce first the concept of deterministic algorithm and program.

It is important to notice that in this framework, the concept of determinism is approached in two different ways:

- A determinism that only considers input/output relations.
- A determinism that considers the whole process, including the internal states.

With the first approach, a deterministic algorithm will be an algorithm which, given a particular input, will always produce the same output. On the other hand, the second approach assumes also that the underlying machine always passes through the same sequence of states.

From the point of view of computer programs as implementation of aggregation processes, our interest relies on input-output relations. Consequently we will focus on the first conception when defining a deterministic program.

Definition 6. *A program P is deterministic, or repeatable, if it produces the very same output when given the same input no matter how many times it is run.*

It would be possible adding to this definition of deterministic program, a concept similar to the *same sequence of states* considered for the deterministic algorithm. In fact, according to [20], in a deterministic program there is at most one instruction to be executed *next*, so that from a given initial state only one execution sequence is generated. This could be a good option in case we were interested in the verification of a program, being the execution process a key aspect. But it is obviously too restrictive for our purpose, since we are only interested in the result of the Program. Consequently, we will consider Definition 6 as our conception of a deterministic program.

No matter which of the two possible definitions we consider, when applying a deterministic program P we can refer to the output produced for a particular input (a list L), as $P(L)$. That is because the same input (L) will always produce the same output ($P(L)$), generating a mapping.

Obviously, a program or an algorithm are non-deterministic when they do not match with the previous definitions.

It is clear according to the definitions that (when having the same approach to determinism) a non-deterministic algorithm will always produce a non-deterministic program. Consequently, when analyzing if a computable aggregation is deterministic or non-deterministic we should simply consider the corresponding Program.

Definition 7. *A computable aggregation P over the set T is non deterministic if and only if the program P is non deterministic.*

From this definition we will say that a computable aggregation P is deterministic when the program P is deterministic, implying that the underlying algorithm is also deterministic.

Let us denote by $\mathcal{P}_{\mathcal{D}}$ the class of deterministic computable aggregations and by $\mathcal{P}_{\mathcal{ND}}$ the set of non deterministic computable aggregations.

From now on, we will analyze the case in $T = [0, 1]$.

3.1 Some Non Deterministic Computable Aggregations

It is obvious that in many cases the non-deterministic behavior of a program relates to an inappropriate coding that generates an unexpected problem. This is usually the case when we have a deterministic algorithm that being wrongly programed produces a non-deterministic output. But this is not the kind of non-determinism we are interested in. Our interest relies on programs describing aggregation processes that are intrinsically non-deterministic, processes where, as an example, random or probabilistic decisions are involved.

In this subsection, we will define some interesting cases of non deterministic computable aggregations (NDCAs).

Definition 8. *Given a value $p \in (0, 1]$, and given a family of aggregation operators $\{Ag_n : [0, 1]^n \longrightarrow [0, 1],\ n \geq 2\}$, let us define the computable aggregation $P_{Ag,p}$ as the two steps program that for a given list $l = (x_1, \ldots, x_n) \in L<[0, 1]>$ performs the following actions:*

- **Step 1.** *To reduce the list l into another list l_p of lower (or equal) dimension by randomly erasing the elements of the list with probability $1 - p$.*
- **Step 2.** *To return the value $Ag_{|l_p|}(l_p)$ if $|l_p| \geq 2$ and 0 otherwise.*

Note that the computable aggregation $P_{Ag,p=1}$ coincides with the program associated with the family of aggregation operators $\{Ag_n,\ n \geq 2\}$ and is deterministic since the list $l_p = l$ when $p = 1$.

Obviously, $P_{Ag,p}$ as described in Definition 8 is not only a computational aggregation, it is in fact a generic approach that induces as many different computable aggregation as the possible families of aggregation operators $\{Ag_n\}$. In particular, we will analyze in this paper three cases: the arithmetic mean, maximum and minimum aggregation operators. From now on, we will denote the three of them as: $P_{M,p}$, $P_{Max,p}$ and $P_{Min,p}$ described as the sample mean or average, the sample maximum and the sample minimum respectively.

Another way to build a class of Non deterministic computable aggregations should be to fix a value k, and randomly select k elements from the list, as those to be aggregated. Given a list of n elements of $[0, 1]$, $l = \{x_1, \ldots, x_m\} \in <[0, 1]>$, let us denote by Sel_k a program that randomly chooses a sample (without resampling) of k elements of the list if $m \geq k$, and maintains the same list in other case.

Definition 9. *Given a family of aggregation operators $\{Ag_n\ n \geq 2\}$, the computable aggregation $P_{Ag,k}$, with $k \leq n$, is defined as the program that for a given*

list l of m elements, first applies the procedure $l' = Sel_k(l)$, and then computes the value $Ag_{|l'|}(l')$.

It is very easy to see that the computable aggregation $P_{Ag,k}$ is also a non-deterministic computable aggregation (being deterministic when $k \geq m$). The main difference with the previous computable aggregation is that here the dimension of the list to be aggregated is upper bounded by k, consequently the family of aggregation operators is also bounded in dimension, no matter the dimensions of the list to be aggregated.

Definition 10. *Given a normal distribution $N(\mu, \sigma)$ and a family of aggregation operators $\{Ag_n : [0,1]^n \longrightarrow [0,1],\ n > 2\}$, we define the* **noisy computable aggregation** *$P_{N(\mu,\sigma),Ag}$ as the program that for any list $l \in <[0,1]>$, with $|l| \leq n$, returns $P_{N(\mu,\sigma),Ag} = Ag(l_N)$, where l_N is the list generated by replacing each element in l with the same element after being modified by adding noise generated by the normal distribution (truncated to 0 or 1 in case that the resulting value was out of the $[0,1]$ interval).*

Definition 11. *The pure random computable aggregation P_{PR} is a program that for any list $l \in <[0,1]>$ returns a random value in $[0,1]$.*

Definition 12. *The bounded random computable aggregation P_{BR} is a program that for any list $l \in <[0,1]>$ returns a random value in the interval $[Min(l), Max(l)]$.*

Proposition 1. *The computable aggregation operators: $P_{M,p}$, $P_{Max,p}$, $P_{Min,p}$ $P_{Ag,k}$, $P_{N(\mu,\sigma),Ag}$, P_{PR}, P_{BR} are non deterministic computable aggregation operators if $\sigma > 0$ and $p < 1$.*

An example of C++ program P implementing $P_{M,p}$ and $P_{Max,p}$ is described in Fig. 1.

```
double aggSampleAVG(PopulationData d, double prob) {
    int n = 0;
    double sum = 0;
    for (int i = 0; i < PopulationSize; i++)
        if (generateUniform(0,1) < prob) {
            sum += d[i];
            n++;
        }
    return sum / n;
}
double aggSampleMax(PopulationData d, double prob) {
    double max = 0;
    for (int i = 0; i < PopulationSize; i++)
        if (generateUniform(0, 1) < prob)
            if(max< d[i]) max= d[i];
    return max;
}
```

Fig. 1. An implementation of $P_{M,p}$ and $P_{Max,p}$ in C++.

4 Exploring Non Deterministic Computable Aggregations

Given a computable aggregation P that aggregates a list $l \in L<T>$, let us denote by $\mathcal{P}(l)$ the theoretical distribution after all possible realizations of the program P over the fixed list l. Obviously, if $P \in \mathcal{P}_{\mathcal{D}}$, the associate $\mathcal{P}(l)$ for any list will be a single value. For non deterministic programs, we will have here a probability distribution $\mathcal{P}(l)$ for each fixed value of l.

In general, given a non deterministic computable aggregation P, it is not possible to know the theoretical distribution $\mathcal{P}(l)$. Nevertheless, we could try to approximate it by making many realizations of $P(l)$. In the following definition we distinguish between the empirical and theoretical distribution.

Definition 13. *Given a computable aggregation P and given a list $l \in L<T>$, the distribution of results obtained after n executions of the program P over the list l, will be referred us the empirical distribution with size n of the program P over the list l, represented by $DP_{n,l}$.*

Proposition 2. *Given a deterministic computable aggregation P, the following holds:*

$$DP_{n,l} = \mathcal{P}(l)$$

To analyze the aggregation process previously described and implemented by an NDCA we have several components to consider. There is a list $l \in L<T>$ of elements to be aggregated. There is also a family of aggregation operators $(\{Ag_n\})$ underlying in the non deterministic aggregation process. Finally, two distributions describe the aggregation process: the theoretical distribution $\mathcal{P}(l)$ and the empirical distribution $(DP_{n,l})$. It is obvious that the interactions and relations among these elements will describe and characterize an NDCA.

Some of the questions to be considered are obvious, as the relations between the properties of both distributions (the theoretical and the empirical). But it could also be interesting to consider potential relations between some properties of the list l (analyzed as a distribution) and the corresponding properties of the theoretical/empirical distribution. Another important question could be to compare the result produced by the deterministic underlying aggregation, that is $Ag_{|l|}(l)$, and some properties of the distributions (mean, median, etc) generated by the NDCA.

As said before, it is in general not possible to know the theoretical distribution generated by a non deterministic computable aggregation P when applied on a list l $(\mathcal{P}(l))$. We will replace it with an empirical distribution $(DP_{n,l})$, and we need both to have similar properties.

In that sense we can define the concept of robustness of an NDCA as a measure of the similarity of both distributions (theoretical and empirical).

Definition 14 Robust. *A non deterministic computable aggregation P is said to be Robust, if and only if, for any $l \in L<[0,1]>$, for any $t \in [0,1]$, and for*

any $\epsilon > 0$*, there exists* n_0 *such that the absolute difference between the empirical distribution function*

$$\widehat{DP_{l,n}}(t) = \frac{number\ of\ elements\ in DP_{l,n} \leq t}{n}$$

and the real distribution function $F_{P(l)}(t)$ *is lower than* ϵ *for* $n \geq n_0$.

4.1 Empirically Exploring Non Deterministic Computable Aggregations

Given a non deterministic computable aggregation $P = agg$ and a list l, Fig. 2 presents a $C{+}{+}$ program generating the empirical distribution $DP_{n,l}$ for a list l with length 1000.

```
//Population size
const int PopulationSize = 10000;
// Array (List) of population data
typedef double PopulationData[PopulationSize];
// Times to execute an aggregation to a Population Data
const int AggExecutionTimes = 1000;
// Array (List) to store the ExecutionTimes outputs
typedef double ExecutionsOutputData[AggExecutionTimes];
//Aggregation: List<[0, 1]> -> [0, 1]
double agg(PopulationData d);
// Input: Population Data  Output: ExecutionsOutputData
void generateAggExecutions(PopulationData l,ExecutionsOutputData o){
    for (int i = 0; i < AggExecutionTimes; i++) {
        o[i] = agg(l);
    }
}
```

Fig. 2. An implementation of $Dagg_{l,n}$ in C++.

Figure 3 presents the empirical distribution ($DP_{n,l}$) for some of the previously defined NDCAs[1], with $n = 1000$ and being l a list of 10000 random values[2] generated either using a uniform(0,1) distribution or a Normal(0.5,0.1) bounded in [0, 1][3].

Note that the sample and the noisy mean NDCA have a normal distribution and the pure random and bounded pure random NDCA have uniform distribution. Note also that the sample min and the sample max NDCA have more dispersion in the $N(0.5, 0.1$ population generated list that with the $U(0, 1)$ population generated list.

[1] The C++ program could be downloaded from https://github.com/lgarmend/NonDeterministicComputableAggregations/.

[2] The generated population is available in file GeneratedPopulationData.txt, at https://github.com/lgarmend/NonDeterministicComputableAggregations/blob/master/GeneratedPopulationData.txt.

[3] The executions lists are saved in file GeneratedExecutionsList.txt for the cases of uniform and Normal population distribution.

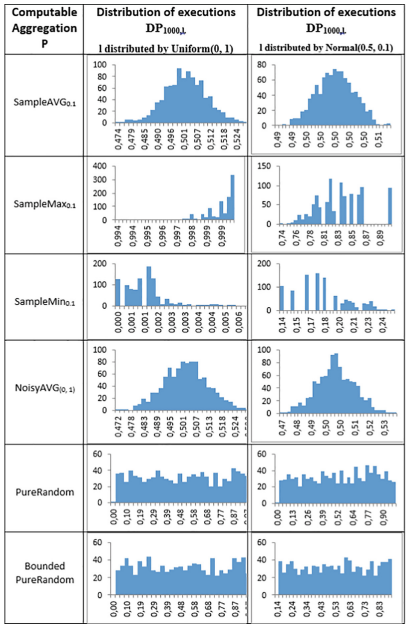

Fig. 3. Empirical distribution for some NDCAs

5 Characterizing Non Deterministic Computable Aggregations

Three properties were initially considered in aggregation operators theory: two boundary conditions plus monotonicity property. Taking into account now that for a non-deterministic computable aggregation the output for a fixed input does not necessarily have to be a single-point value, in this section we will try to extend (or at least give possible extensions) of how these concepts could be generalized in the context of non deterministic computable aggregations.

In the following definition, we present the pure case in which a computable aggregation satisfies the boundary conditions and also the idempotent property. Firs at all, let us denote by l_a a list with all elements equal to a, i.e $l_a = (a, \ldots, a)$.

Definition 15 *A computable aggregation P over the set $T = [0, 1]$ is idempotent if and only $P(l_a) = a$ for all $a \in [0, 1]$.*

Definition 16 *A computable aggregation P over the set $T = [0, 1]$ satisfies the two classical boundary conditions if and only the following holds:*

- $P(l_0) = 0$.
- $P(l_1) = 1$.

Let us note that if a computable aggregation is idempotent then it satisfies the two boundary condition.

Proposition 3 *If the family* $\{Ag_n\}$ *of aggregation operators is idempotent, then the following holds:*

- *The computable aggregation* $P_{Ag,p}$ *is idempotent.*
- *The computable aggregation* $P_{Ag,k}$ *is idempotent.*

As a consequence of the previous proposition the following corollary holds.

Corollary 1. *The computable aggregations* $P_{M,p}$, $P_{Max,p}$, $P_{Min,p}$, $P_{M,k}$, $P_{Max,k}$, $P_{Mix,k}$ *are idempotent.*

Proposition 4. *If the family* $\{Ag_n\}$ *of aggregation operators satisfies the boundary conditions then the following holds:*

- *The computable aggregation* $P_{Ag,p}$ *satisfies the boundary conditions.*
- *The computable aggregation* $P_{Ag,k}$ *satisfies the boundary conditions.*

As a consequence of the previous proposition the following corollary holds.

Corollary 2. *The computable aggregations* $P_{M,p}$, $P_{Max,p}$, $P_{Min,p}$, $P_{M,k}$, $P_{Max,k}$, $P_{Mix,k}$ *satisfy the boundary conditions.*

Proposition 5. *The computable aggregations* P_{BR}, P_{PR}, $P_{N(\mu,\sigma)}$ *are non idempotent and do not satisfy the boundary conditions.*

In previous section we introduced robustness as the way to establish that both the theoretical and the empirical distributions are similar. This idea will allow us to work on the basis of $DP_{l,n}$, considering that in most cases the theoretical distribution is unknown.

We will introduce now the idea of consistency of an NDCA, as the property that considers how close is the behavior of P, with that of Ag, the underlying aggregation process considered by the NDCA.

Definition 17 Consistent in ϕ**.** *A non deterministic computable aggregation* P *is said to be robust in* ϕ*, if and only if, for any* $l \in L{<}[0,1]{>}$*, and for any* $\epsilon > 0$*, there exists* n_0 *such that the absolute difference between* $\phi(\pi(DP_{l,n}))$ *and* $\phi(l)$ *is lower than* ϵ *for any* $n \geq n_0$*, where* $(\pi(DP_{l,n}))$ *is the vector representation of the set* $DP_{l,n}$*.*

Proposition 6 *If* $p \in (0,1]$*, the following holds:*

- $P_{M,p}$ *is Consistent in mean.*
- $P_{Max,p}$ *is Consistent in Max.*
- $P_{Min,p}$ *is Consistent in Min.*

Proof. For random sample theory, it can be seen that the convergence of $M(\pi(DP_{l,n})$ is $M(l)$, $Max(\pi(DP_{l,n})$ is $Max(l)$ and $Min(\pi(DP_{l,n})$ is $Min(l)$.

It would be also important to consider how the dispersion of $DP_{l,n}$ evolves with n. Ideally we would like the dispersion to reduce when n increases.

Definition 18 Concentrate. *A non deterministic computable aggregation P is said to concentrate when the dispersion of $DP_{l,n}$ decreases (is non increasing) with n.*

Now let us define what we understand as monotonicity in non deterministic computable aggregation, since for the deterministic the definition can be reproduced in the same way. A definition of monotony for any function $f : X \longrightarrow Y$ required first the definition of an order in the spaces X and Y, in such a way if we have two inputs $l, l' \in X$ with $l \leq_X l'$ then monotonicity implies that $f(l) \leq_Y f(l')$.

Taking into account this, if we want to define some class of monotonicity in the case of non-deterministic computable aggregations we have to define first an order in the input space (the set of possible lists $L<[0,1]>$) and also an order relation in the space of finite subsets of $[0,1]$. Let us denote by $\widetilde{\leq_{L<[0,1]>}}$ an order between lists and let us denote by $\widetilde{\leq_{P_F[0,1]}}$ an order between finite sets contained in $[0,1]$. Once these two order relations be explicitly defined, the monotony definition is fixed as follow:

Definition 19 *Let $\widetilde{\leq_{L<[0,1]>}}$ be a partial order on the list set and let $\widetilde{\leq_{P_F[0,1]}}$ be a partial order on the sets of finite sets contained in $[0,1]$, then a computable aggregation P is monotone if and only if given any pair of lists l and l' with $l \widetilde{\leq_{L<[0,1]>}} l'$ this implies that $DP_{l,n} \widetilde{\leq_{P_F[0,1]}} DP_{l',n}$ with n large enough.*

Although this definition is perfectly valid, it would be interesting to study in deep different possibilities that will produce different ideas of monotonicity. In particular, in this article we provide a possible order but others could be defined.

The problem of establishing a possible order among the set of ordered lists does not seem very complex if we focus on the case in which the lists presents equal size, since the order relationship coincides with the natural order relationship in $[0,1]^k$, so we will consider that natural order. The case of order over finite sets (even if they have the same size) is much more complex.

In this paper, we propose a possible idea of order that is related with a majority rule. A finite set $S \leq_T S'$ if and only if the following holds:

$$|\{(x,y) \in S \times S' \ / \ x \leq y\}| \geq |\{(x,y) \in S \times S' \ / \ y \leq x\}|.$$

Just to put an example we have that the set $S = \{0.1, 0.3, 0.5\}$ is lower than $S' = \{0.2, 0.6, 0.7\}$ since from the 9 pairwise comparison the elements of S' win in 7 of the cases. Taking into account this consideration the following monotonicity definition is given:

Definition 20 Tournament monotonicity. *A non deterministic computable aggregation P over the domain T is Tournament monotonic if and only if for any pair of lists l,l' of T with the same cardinal such that $l \leq l'$ there exist n_0 in which the following holds*

$$DP_{n,l} \leq_T DP_{n,l'} \quad for \ n \geq n_0.$$

6 Conclusions

The definition of Computable aggregation implies a rupture between functions and aggregation processes allowing the incorporation of a new class of aggregations: the non deterministic computable aggregation. This new class of computable aggregations is characterized by aggregations where the result of the process could change even if we fix the information that has to be aggregated. Obviously, no function can model this class of aggregation process since by definition the output of a function is always the same for a fixed input value.

It is important to emphasize that there are many real situation in which the information is aggregated in a non deterministic way. For example, any inference based on random sampling is a well-known example of this. If the population is huge, it is very frequent to obtain a sample and operate over it. But this is not the only case, any aggregation process in which randomness appear could be understood as a non deterministic process. Many artificial intelligence or machine learning aggregation process could be classified also as non deterministic.

In this paper, we have tried to define some desirable properties for these new class of aggregation process as robustness, stability, boundary conditions, and monotonicity. This questions open the gate for further research.

Acknowledgment. This research has been partially supported by the Government of Spain (grant PGC2018-096509-B-I00), the Government of Madrid (grant S2013/ICCE-2845), Complutense University (UCM Research Group 910149) and Universidad Politeécnica de Madrid.

References

1. Bouchon-Meunier, B.: Aggregation and Fusion of Imperfect Information, vol. 12. Physica, Heidelberg (2013)
2. Bustince, H., Herrera, F., Montero, J. (eds.): Fuzzy Sets and Their Extensions: Representation, Aggregation and Models. Studies in Fuzziness and Soft Computing, vol. 220. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-73723-0
3. Gómez, D., Montero, J.: A discussion on aggregations operators. Kybernetika **40**, 107–120 (2004)
4. Gómez, D., Rojas, K., Montero, J., Rodríguez, J., Beliakov, G.: Consistency and stability in aggregation operators, an application to missing data problems. Int. J. Comput. Intell. Syst. **7**, 595–604 (2014)
5. Montero, J., del Campo, R.G., Garmendia, L., Gómez, D., Rodríguez, J.: Computable aggregations. Inf. Sci. **460**, 439–449 (2018)
6. González-del-Campo, R., Garmendia, L., Montero, J.: Complexity of increasing phi-recursive computable aggregations. In: XIX Congreso Español sobre Tecnologías y Lógica Fuzzy (CAEPIA-ESTYLF), pp. 187–191 (2018)
7. Magdalena, L., Garmendia, L., Gómez, D., del Campo, R.G., Rodríguez, J.T., Montero, J.: Types of recursive computable aggregations. In: 2019 IEEE International Conference on Fuzzy Systems. FUZZ-IEEE 2019, pp. 1–6. IEEE (2019)

8. Calvo, T., Kolesárová, A., Komorníková, M., Mesiar, R.: Aggregation operators: properties, classes and constructions methods. In: Calvo, T., Mayor, G., Mesiar, R. (eds.) Aggregation Operators. Studies in Fuzziness and Soft Computing, vol. 97, pp. 3–104. Physica, Heidelberg (2002). https://doi.org/10.1007/978-3-7908-1787-4_1
9. Rojas, K., Gómez, D., Montero, J., Rodríguez, J.: Strictly stable families of aggregation operators. Fuzzy Sets Syst. **228**, 44–63 (2013)
10. Beliakov, G., Gómez, D., James, S., Montero, J., Rodríguez, J.: Approaches to learning strictly-stable weights for data missing values. Fuzzy Sets Syst. **325**, 97–113 (2017). https://doi.org/10.1016/j.fss.2017.02.003
11. Olaso, P., Rojas, K., Gómez, D., Montero, J.: A generalization of stability for families of aggregation operators. Fuzzy Sets Syst. **378**, 68–78 (2020)
12. Beliakov, G., Pradera, A., Calvo, T.: Aggregations Functions: A Guide for Practitioners. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73721-6
13. Cutello, V., Montero, J.: Recursive families of OWA operators. In: Proceedings of the IEEE International Conference on Fuzzy Systems (FUZZIEEE 1994), Orlando, USA, pp. 1137–1141 (1994)
14. Cutello, V., Montero, J.: Hierarchical aggregation of owa operators: basic measures and related computational problems. Uncertainty Fuzziness Knowl.-Based Syst. **3**, 17–26 (1995)
15. Cutello, V., Montero, J.: Recursive connective rules. Int. J. Intell. Syst. **14**, 3–20 (1999)
16. Lucca, J.G., Dimuro, G.P., Bedregal, B.R.C., Mesiar, R., Kolesárová, A., Bustince, H.: Preaggregation functions: construction and an application. IEEE Trans. Fuzzy Syst. **24**(2), 260–272 (2016)
17. Magdalena, L., Gómez, D., Montero, J., Cubillo, S., Torres, C.: Generalized pre-aggregations. In: Kearfott, R.B., Batyrshin, I., Reformat, M., Ceberio, M., Kreinovich, V. (eds.) IFSA/NAFIPS 2019 2019. AISC, vol. 1000, pp. 362–370. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21920-8_33
18. Minsky, M.: Computation: Finite and Infinite Machines. Prentice-Hall, Englewood Cliffs (1967)
19. Wirth, N.: Algorithms + Data Structures = Programs. Prentice-Hall, Englewood Cliffs (1976)
20. Apt, K.R., Olderog, E.-R.: Deterministic programs. In: Apt, K.R., Olderog, E.-R. (eds.) Verification of Sequential and Concurrent Programs. Graduate Texts in Computer Science, pp. 47–99. Springer, New York (1997). https://doi.org/10.1007/978-1-4757-2714-2_3