Prog Artif Intell (2017) 6:145–158
DOI 10.1007/s13748-017-0112-x

REGULAR PAPER

Exhaustive search algorithms to mine subgroups on Big Data using Apache Spark

F. Padillo[1] · J. M. Luna[1] · S. Ventura[1]

Received: 17 December 2016 / Accepted: 11 January 2017 / Published online: 27 January 2017

Abstract Subgroup discovery is a well-known technique for the extraction of patterns, with respect to a variable of interest in the data. However, the explosion in data gathering has hampered the performance of traditional algorithms to discover interesting relationships between different objects in a set with respect to a specific property which is of interest to the user. In this regard, our goal is to propose a set of efficient techniques to mine subgroups on Big Data by means of Apache Spark. On this matter, AprioriK-SD-OE and PFP-SD-OE are proposed as fast exhaustive search algorithms to discover subgroups on Big Data using Apache Spark. The experimental study includes more than 70 datasets considering search spaces bigger than 10^{15} subgroups. The scalability of our proposals are analyzed by considering datasets with 200 million of instances demonstrating the usefulness of using Spark to tackle Big Data.

Keywords Subgroup discovery · Big Data · Apache Spark

1 Introduction

The increasing innovation in data gathering of the last years has provoked an exponential growth on the quantity of data to deal with. The generation and collection of large datasets has further encouraged the analysis and knowledge extraction process [1]. The discovery of high-level information from raw massive data has become a priority for both academia and industry, driving and motivating the research in improving techniques for data analysis in such massive datasets red. Big Data is the new buzzword more and more used to refer to the techniques used to face up the problems arising from the management and analysis of these huge quantities of data.

Aiming at extracting hidden and interesting information from large quantities of data, many different techniques have been proposed, even though all of them could be cataloged in two different categories [2]. Descriptive learning tasks focus on characterizing and exploring the data, since no information is known previously. On the contrary, the purpose of predictive learning is given a sequence of inputs and outputs, try to predict new outputs as accurate as possible. Notwithstanding, some recent techniques are in halfway between the aforementioned categories. Supervised local pattern mining is one of them, and specifically, Subgroup discovery (SD) has received special attention [3].

SD is a broadly applicable data mining technique aimed at discovering interesting relationships between different objects with respect to a specific property which is of interest to the user (the target variable) [4]. To mine these subgroups, many different techniques have been proposed [3], although almost all of them are based on adaptations from the descriptive tasks. As a matter of example, Apriori [5] or FP-Growth [6] algorithms are some of the most distinguished algorithms from unsupervised learning which have been adapted to extract subgroups. Special attention has received these adaptations of FP-Growth since it enables to speed up the process thank to a highly efficient data structure known as FP-Tree.

Although the algorithms based on FP-Tree speed up the process of mining subgroups, the performance of these algorithms could be hampered on truly Big Data. Some

✉ S. Ventura
sventura@uco.es

F. Padillo
fpadillo@uco.es

J. M. Luna
jmluna@uco.es

[1] Department of Computer Science and Numerical Analysis, University of Cordoba, Rabanales Campus, Córdoba, Spain

researchers have faced up this challenging problem by means of heuristic algorithms reducing the search space [7], or even considering sampling techniques [8]. Although these techniques achieve to reduce the search space, many interesting solutions could be neglected losing interesting subgroups. In this regard, an exhaustive search is required to study the whole set of results. It is worth noting that these kind of algorithms obtains all the possible solutions present in the dataset. However, its main issue is the huge search space that they need to consider. Aiming at reducing the search space without losing any interesting subgroups, optimistic estimates (OEs) have been proposed [9]. These estimators enable to prune the search space achieving a better runtime without losing any accuracy in the results.

At this point, our goal is to propose a new set of algorithms based on traditional approaches but using emerging technologies to tackle Big Data. To avoid the loss of interesting subgroups, only exhaustive search algorithms have been proposed. To face up the aforementioned issues of this kind of algorithms, our proposals consider OEs to prune the search space. Hence, it could be stated that the proposed algorithms extract the best subgroups available. Furthermore, to tackle massive data, Apache Spark has been used since it has proved to obtain excellent results on dealing with massive datasets [10]. Thus, two exhaustive search algorithms to mine subgroups on Big Data are proposed in this paper. ApriorIK-SD-OE (ApriorIK-Subgroup Discovery Optimistic Estimate) as a fast iterative exhaustive search algorithm to discover subgroups on Big Data. Likewise, PFP-SD-OE (Parallel FP-Growth Subgroup Discovery Optimistic Estimate) is an exhaustive search algorithm to mine subgroups on Big Data using a highly efficient data structure. All of them are able to discover subgroups on binary, numeric or nominal targets.

A detailed experimental study has been carried out to prove the efficiency and scalability of our proposals. In the analysis, a comparative study with traditional techniques has also been performed. The results reveal that our proposals outperform with respect to traditional algorithms on Big Data. The experimental study includes more than 60 datasets considering search spaces bigger than 10^{15} subgroups and 2×10^8 instances.

This paper is structured as follows. Section 2 presents the most relevant definitions and related work. Section 3 describes the proposed algorithms. Sect. 4 presents the experimental analysis. Finally, some concluding remarks are stated in Sect. 5.

2 Related work

In this section, the subgroup discovery task is formally defined and some traditional approaches are described. Finally, Apache Spark and its foundation are outlined.

2.1 Subgroup discovery

Subgroup discovery (SD) is a widely applicable data mining technique whose aim is to identify interesting relationships in a population with high generality and distributional unusualness [11]. In where the relations are represented by means of rules with the form $R \equiv A \rightarrow B$, being A a set of independent variables known as subgroup, and B represents a value of the target variable. The number of independent variables in A is the length of the subgroup. In order to quantify the quality of the obtained solutions, tonnes of qualities have been proposed. The aim of these measures is to quantify how interesting a specific subgroup is within a dataset [3]. Thus, the goal is to obtain the j subgroups with a higher value for the quality q. Hence, this task is formally described as follows. Given a database DB, a quality function q and j being the number of subgroups with maximum quality. To return a set of j subgroups descriptions G such that: $\forall$ subgroup description $sd : sd \notin G \Rightarrow q(DB, sd) \leq q*$, where $q* = min_{sd \in G} q(DB, sd)$.

Given that the space of candidate subgroups could be considered as a tree with subgroups of length 1 at the first level, subgroup descriptions of length 2 at the next level and so on. A naive approach could be to mine this tree recursively; however, in this approach, the size of search space is exponential in the number of attributes, and thus, some kind of pruning is required to be run efficiently. Unfortunately, unlike related tasks like association rule mining where state-of-the art algorithms exploit the property of monotonicity [5], in SD this property does not hold: Even if the subgroup description sd does not have a sufficient quality, it is still necessary to consider its refinements [9]. However, if the j best subgroups have been found, and all the refinements sd' of a subgroup sd had a quality that is worse or equal than all j subgroups found so far, this search space could be safely pruned. What is needed to do so is an optimistic estimate (OE) for the refinements sd' of sd. Where an $OE(sd)$ for a given quality function q is a function that satisfies the following $\forall sd, sd'.sd' \succ sd \Rightarrow OE(sd) \geq q(sd')$. Also, tight OEs could be found which are OEs that satisfy the condition, $\forall sd, sd'.sd' \succ sd \Rightarrow OE(sd) = q(sd')$. The tightness the OE, the better the estimation, thus the higher the part of search space which could be safely pruned.

To obtain these j subgroups, many approaches have been used [3]. Pioneering researches were based on adaptations from the association rule mining field, where one of the most distinguished algorithms, known as Apriori [5], have been adapted to the SD task for different kind of targets [12]. Apriori follows a breadth-first search, where patterns are considered as frequent if they have a value of frequency greater than a threshold. It generates candidate itemsets of length s from itemsets of length $s - 1$. Apriori, while historically significant, suffers from a number of inefficiencies

or trade-offs, which have spawned other algorithms. Candidate generation is time-consuming and requires a huge amount of memory, by this reason FP-Growth has been proposed as an approach which not requires the process of generating candidates. FP-Growth uses a compact representation of the database reducing the memory requirements. The core of this method is the usage of a special data structure named FP-Tree, which retains the patterns information. Finally, this data structure is mined in a recursive way. Past proposals are analyzed in function of the kind of considered target.

(a) *Binary* SD-MAP [13] is an algorithm to discover subgroups in binary targets using a highly efficient data structure, known as FP-Tree, which enables to speed up the process. In these targets, there are only two different values (True/Positive or False/Negative). The most typical quality function used in these targets has been proposed by *Piatetsky-Shapiro* [14] as $q_{ps} = n(p - p_0)$. Considering this measure, Grosskreutz et al. [9] proposed a tight OE as $OE_{ps} = np(1 - p_0)$ enabling to prune huge parts of the search space, where n means the size of the subgroup; p is the relative frequency of the target variable in the subgroup; and p_0 is the relative frequency on the target variable in the total population.

(b) *Numeric* some authors proposed to discretize target variable, but this solution is not accurate at all since it involves a loss of information [15]. Unlike it, SD-MAP* [16] mines subgroups in numeric targets. It searches for significant deviations of an aggregation measure among the others. Being the mean one of the most outstanding [17], where the mean value μ_P in the subgroup P is compared to overall mean in the dataset $\mu_\emptyset$. In concrete, one measure based on the generic mean function is the impact quality function defined as $q_{\text{imp}} = q^1_{\text{mean}}(P) = n \cdot (\mu_P - \mu_\emptyset)$. For this measure, an OE [17] has been proposed to prune the search space as $OE_{q_{\text{imp}}} = \sum_{t \in P: T(t) > \mu_\emptyset} T(t) - \mu_\emptyset$. n refers to the size of the subgroup and $T(t)$ means the value for the target variable in the transaction t.

(c) *Nominal* DpSubgroup [18] is one of the most prestigious algorithms where an FP-Tree is used to speed up the task on nominal targets. Gini is considered as one of the most distinguished either in SD and in data mining in general [19]. It is defined as $q_{\text{gini}}(P) = \frac{n}{N-n} \sum_i (p_i - p_{0i})^2$. An OE proposed by Grosskreutz et al. [9] is $OE_{\text{gini}} = \sum \max\{E_i^+, E_i^- E_i^g\}$, where $E_i^- = \frac{n(1-p_i)}{N-n(1-p_i)}(p_{0i})^2$, $E_i^g = \frac{n}{N-n}(p_i - p_{0i})^2$, and $E_i^+ = \frac{np_i}{N-np_i}(1 - p_{0i})^2$, where N means the size of the dataset; n is the size of the subgroup; p_i is the distribution over the class i in the subgroup; and p_{0i} is the distribution over the class i in the dataset.

Despite of these algorithms are highly efficient considering both OEs and data structures, any of them follow a distributed approach. Thus, they only could be scaled up requiring each time a higher quantity of resources. Hampering its application on truly Big Data, where it is required to scale out [20].

2.2 Foundations of Apache Spark: MapReduce

MapReduce [21] is a recent framework of distributed computing, where algorithms are composed of two main stages, map and reduce. In the map phase, each mapper processes a subset of input data and produces a set of $\langle k, v \rangle$ pairs. Finally, the reducer takes this new list as input to produce the final values. Since its origins, many implementations have been proposed [10,22].

Hadoop [22] is the de facto standard for implementation of MapReduce applications. Hadoop implements the MapReduce paradigm efficiently and the Hadoop Distributed File System (HDFS), which replicates file data in multiple storage nodes that can concurrently access to the data. The main drawback of Hadoop is that it imposed an acyclic data flow graphs, and there are applications that cannot be expressed efficiently using this kind of graph such as iterative or interactive analysis. As it has supposed a bottleneck for iterative algorithms which are very common in machine learning, a novel solution has been proposed known as Spark [10]. Spark is designed to cooperate with Hadoop, specially by using HDFS. To solve these downsides, Spark introduces an abstraction called Resilient Distributed Datasets (RDDs). RDD represents a read-only collection of objects partitioned across a set of machines. It allows us to load a dataset in memory one time and read multiple times from executor process without having to load it in each iteration as occurred in Hadoop. Not only introduces Spark an in memory storage but also it provides a thorough set of programming primitives enabling users to implement much more complex distributed applications than those implemented in classic MapReduce. It enables to implement several distributed models like MapReduce, SQL like or even Pregel.

3 Exhaustive search approaches to mine subgroups on big data

With the arrival of Big Data, existing algorithms are not able to run efficiently due to the huge size of the data to deal with. Whereas other researchers have preferred to lose some accuracy on the results introducing heuristic, our proposals are able to work in large datasets using an exhaustive search. It guarantees that the best solutions are obtained. Hence, the aim of this paper is to propose new approaches of discovering subgroups on Big Data by combining pioneering

techniques with new trends. In this regard, a full description of our approaches is given, relying on Apache Spark. Each version obtains the exact same set of results, but the approach followed in the search process is totally different.

3.1 AprioriK-SD-OE

AprioriK-Subgroup Discovery Optimistic Estimate (AprioriK-SD-OE) is an iterative algorithm based on the well-known Apriori [5] algorithm to mine subgroups on Big Data. A really important handicap of Apriori is the extremely high number of items which could produce at the same time ($2^X - 1$ items could be produced for X items) [23]. Moving on to SD field, supposing l attributes where each one has m different values, then $\sum_{i=1}^{l} C_i^l \times m^i$ subgroups could be produced. Thus, to deal with this issue, an option is not to create the whole lattice for each transaction but the l-sized sub-lattice each time, so a set of iterations is required. In this regard, the memory requirements are being reduced, achieving an improvement in the computational time. Additionally, OE allows to reduce the search space enabling to stop the process of discovering subgroups when the j best subgroups have been mined, reducing considerably the computational time.

The proposed AprioriK-SD-OE obtains subgroups on any kind of target value, changing the quality metric depending on the type of target. For binary targets, the previously described q_{ps} measure is used; for numeric and nominal, the q_{imp} and q_{gini} measures are considered, respectively. The quality metrics could be changed depending on the type of problem. To reduce the resulting set, AprioriK-SD-OE only returns the j best subgroups; thus, only the subgroups with a quality value greater than a threshold will be returned. The OE depends on the quality measure used; thus, OE_{ps}, OE_{imp} and OE_{gini} are considered.

To obtain the j best subgroups, this algorithm uses three different processes: driver, mapper and reducer, all of them are fully described in this section. The proposed algorithm works as follows.

Step 1 The driver read the database from disk and loaded in a RDD, which is split in a set of sub-databases. Then, the driver calculates the maximum size of the subgroups, that is, the maximum number of attributes as $maxL$, iterating from $l = 1$ to $maxL$.

Step 2 For each iteration l, a MapReduce phase is needed. In this regard, our proposal works by running a different mapper for each specific sub-database, the results are collected in a reducer phase. Thus, this step is split in two different subparts, both of them are fully depicted in Code 1.

Step 2.1. Map phase Each of these mappers are responsible for mining the complete set of subgroups of size l for its sub-database (line 1 in Code 1). Producing a set of $\langle k, v\rangle$ pairs where k has a set of attributes, that is, the subgroup.

Code 1 AprioriK-SD-OE algorithm

```
function mapper(l, instance, group, metaInfo)
1: subgroups ← getSubgroupsBySize(instance, l)
2: for each subgroup in subgroups do
3:    info ← getInformation(instance, group, metaInfo)
4:    emit(subgroup, info) // Emit ⟨k, v⟩ pair
5: end for
end function
function reducer(subgroup, Iterator<infoAll>, Threshold_quality)
1: globalInfo ← 0
2: for each info in subgroups do
3:    globalInfo += info
4: end for
5: if globalInfo.quality ≥ Threshold_quality then
6:    emit(subgroup, globalInfo) // Emit ⟨k, v⟩ pair
7: end if
end function
function getInformation(instance,group,metaInfo)
1: if group.isBinary() then
2:    return (tp, fp)
3: else if group.isNumeric() then
4:    cond ← (group > metaInfo.meanGlobal ?
      (group − metaInfo.meanGlobal) : 0)
5:    return (group, 1, group − metaInfo.meanGlobal, cond)
6: else
7:    return generateTuple(metaInfo, group)
8: end if
end function
```

And depending on the kind of target, *value* could contains different values as it will be fully described (this functionality is provided by the function *getInformation* depicted in Code 1).

Step 2.2. Reduce phase The overall count for each subgroup is performed by the reduce phase. In this regard, each subgroup is sent to a different reducer, calculating the overall metric results. The reducer returns to the driver only the subgroups with a quality greater than a threshold.

Step 3 After the MapReduce phase, the driver collects the j best subgroups obtained of size l. These subgroups are joined with the best subgroups obtained up to now, selecting the j best. The best OE of the subgroups of size l is compared with the worst quality of the subgroups of whatever size, and if it is lower, the process could be stopped since that any refinements could not be better than the previously obtained. In red another case, l is incremented and it returns to Step 2. In case that l would have the maximum value, the discovery process would be stopped since any bigger subgroups could not be obtained.

For the sake of better understanding, Fig. 1 shows a scheme of three different executions for the first iteration (one for each type of target), where the subgroups with descriptors of size 1 are searched. The quality threshold is fixed to 0, and the number of subgroups to obtain is established to 2. Firstly, the dataset is split in different sub-datasets, where each one

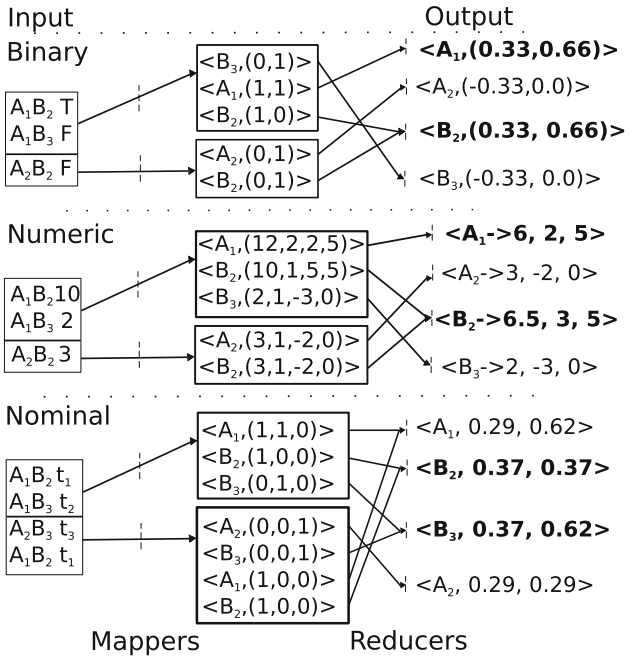

Fig. 1 Example of executions for $l = 1$ in AprioriK-SD-OE. Each execution is performed on different datasets, having each one a different type of target, even though all of them share the same idea being the v in the $\langle k, v\rangle$ pairs the unique difference among them. A quality threshold of 0 and a number of subgroups of 2 are considered. Bold typeface solutions are the best

is processed by a different mapper independently. The aim of the mappers is to mine subgroups of size l. Each mappers produces a set of $\langle k, v\rangle$ pairs, where k is the subgroup and v depends on the kind of target (this functionality is provided by $getInformation$ in Code 1) as it is described.

(a) *Binary* v is a tuple of two longs, where they represent to tp (true positive) and fp (false positive), respectively. For instance, the two key-values $\langle B_2, (1, 0)\rangle$ and $\langle B_2, (0, 1)\rangle$ are produced from different mappers, both of which are aggregated in the reducer phase adding theirs values for tp (cases containing the target variable in the given subgroup) and fp (cases not containing the target variable in the given subgroup), resulting $\langle B_2, (1, 1)\rangle$. Then, the reducer calculates the quality function q_{ps} and OE_{ps} obtaining the values of 0.33 and 0.66, thus the key-value $\langle B_2, 0.33, 0.66\rangle$ is produced. Only the results in bold typeface have been considered in the driver since they are the two best solutions.

(b) *Numeric* The first step in this case is to calculate the mean of the targets' value, thus $meanGlobal = \frac{10+2+3}{3} = 5$. For this kind of target, v contains a tuple of four values. The first one represents to the sum of the target values, the second one represents to the frequency of occurrence, the third one is the addition of $targetValue - meanGlobal$ for each transaction and the last one is the addition of $targetValue - meanGlobal$ for each transaction where the condition $targetValue > meanGlobal$ was satisfied. For instance, two key-values $\langle B_2, (10, 1, 5, 5)\rangle$ and $\langle B_2, (3, 1, -2, 0)\rangle$ are produced from different mappers, both of which are aggregated in the reducer phase adding theirs respective values for v, resulting $\langle B_2, (13, 2, 3, 5)\rangle$. After that, the reducer calculates the mean of the target for this subgroup, dividing the first value of the tuple between the second; thus, a subgroup is produced with the form of $B_2 \rightarrow \frac{13}{2}$ and the quality is the total value contained in the third position of the tuple, the OE is the value in the position four of the tuple, resulting the final key-value: $\langle B_2 \rightarrow \frac{13}{2}, 3, 5\rangle$. Only the results in bold typeface have been considered in the driver since they are the two best solutions.

(c) *Nominal* In this example, a tuple of three elements is used since three different values (t_1, t_2 and t_3) are possible to be included in the target variable. It should be noted that the dimensions of this tuple are determined by the problem, thus it does not have a fixed size. In the example shown in the Fig. 1, the first mapper produces $\langle A_1, (1, 1, 0)\rangle$ where the value is a counter of frequency of each target value as (t_1, t_2, t_3), say, since that A_1 is presented in two different instances having the target values of t_1 and t_2, the tuple is $(1, 1, 0)$. This functionality is provided by *generateTuple* in Code 1. Only the results in bold typeface have been considered in the driver since they are the two best solutions.

3.2 PFP-SD-OE

PFP-SD-OE (Parallel FP-Growth Subgroup Discovery Optimistic Estimate) is an algorithm based on the well-known FP-Growth [6] to discover subgroups on Big Data. It also includes some ideas of the parallel version of FP-Growth [24]. The main advantage of this algorithm is that only two readings of the dataset are required, so huge datasets could be considered. The dataset is read in a distributed way enabling a faster computing, creating a set of independent FP-Trees to represent the current dataset. The number of FP-Trees is determined dynamically by the number of different single values for each attribute, the higher this number the higher the number of FP-Trees. Then, the FP-Trees are mined using a parallel way, where each computer node only has access to one local tree. Moreover, OEs are being considered to reduce the search process; in this way, many branches could be removed guaranteeing that any alluring subgroups could be obtained from them. It considers binary, numeric or nominal targets to be mined, and a different quality measure is used for each one. For binary targets, the aforementioned q_{ps} is considered, whereas for numeric and nominal, the q_{imp} and q_{gini} are used, respectively.

The algorithm works as follows.

Code 2 PFP-SD-OE algorithm

```
function mapperParallelComputing(instance)
 1: for each value in instance do
 2:    emit(value, 1) // Emit ⟨k, v⟩ pair
 3: end for
end function
function reducerParallelComputing(element, Iterator<frequencies>)
 1: globalFrequency ← 0
 2: for each frequency in frequencies do
 3:    globalFrequency += frequency
 4: end for
 5: F−list.append(element, globalFrequency)
end function
function mapperPFP(item, instance, F-list)
 1: a[] ← split(instance)
 2: for j = |instance| - 1 to 0 do
 3:    emit(a[j], a[0]..a[j]) // Emit ⟨k, v⟩ pair
 4: end for
end function
function reducerPFP(item, Iterator<instances>)
 1: LocalFpTree ← null
 2: for each instance in instances do
 3:    insert_build_fp_tree(LocalFpTree, instance)
 4: end for
 5: Mine LocalFpTree recursively
end function
```

Step 1 The driver read the dataset from disk, splitting it in successive parts and loaded in a RDD. In this way, the dataset could be read in a distributed way.

Step 2 A parallel counting is carried out (see Code 2). In this step, the frequency for each value of each attribute is calculated, and the final list will be called as F-list. As the dataset is read completely, some additional operations are also calculated depending on the target value: (*a*)*Binary*: the total value of TP and FP are calculated; (*b*)*Numeric*: the global mean of the target variable is calculated; (*c*)*Nominal*: where the frequency for each value of the target is calculated. These operations are performed because this information will be required to calculate the quality measures. This step requires one read of the dataset, and hence, only one MapReduce phase is required.

Step 3 F-list is sorted by frequency. This step is performed in a single computer in few seconds, since the size of F-list is enough smaller to save in main memory.

Step 4 This step is cornerstone since that it creates the FP-Trees. The mapper phase read the RDD created in Step 1 and the sorted F-list created in Step 3. Each mapper produces a set of $\langle k, v\rangle$ pairs, where k is an item from F-list and v is a item-dependent transaction. For each item of F-list, if it appears in one instance, locate its rightmost appearance, say L, and output a key-value pair with the form $\langle item, instance[1]...instance_i[L]\rangle$. These steps are fully described in the function *mapperPFP* in Code 2. A detailed description about item-dependent transaction and the process carried out in this step is found in [24]. Then, each reducer processes a different item enabling a higher level of parallelism. The reducer creates an independent local FP-Tree and grows its conditional FP-tree recursively. During the recursive process, subgroups are discovered. The quality measure could be calculated in this step since each node has enough information to do it, as well as the OE. When a new node is examined in this recursive process, the OE is compared. If it is lower than the worst subgroup in the collection with the j best subgroups, the recursive step is stopped since that any better subgroups could not be discovered in this branch.

The information contained in each node depends on the type of target value.

(a) *Binary* a single tuple with two values tp (cases containing the target variable in the given subgroup) and fp (cases not containing the target variable in the given subgroup).
(b) *Numeric* one tuple with four elements, having a counter of frequency, an addition of target values for this subgroup, the addition of $targetValue - meanGlobal$ and the addition of $targetValue - meanGlobal$ where the condition $targetValue > MeanGlobal$ was satisfied.
(c) *Nominal* a counter variable for each target is included to keep the frequency. Furthermore, each node independently of the kind of target includes a field with its respective value of OE.

The driver process manages the aforementioned steps, doing as point of coordination. Each reducer returns the j best subgroups after mining its local FP-Tree. Even though it produces $j \times NumberReducers$ subgroups in the driver, when only j subgroups are needed, this is the only way of guaranteeing that the final obtained j best subgroups are returned. In this regard, the driver collects each result from each reducer, sorting and selecting only the final j best subgroups.

For the sake of better understanding, an example of running PFP-SD-OE on a dataset with numeric target is described (See Fig. 2). Supposing a dataset with 4 instances. The Step 1 and Step 2 are performed, where the dataset is split in two different parts, loaded in a RDD and the parallel counting carried out. This parallel counting is developed as a classic MapReduce application, where each mapper extracts subgroups of size 1 from its sub-dataset producing key-value as $\langle subgroup_{size=1}, v\rangle$, where v means the occurrence frequency for this $subgroup_{size=1}$. Then, the aim of the reducer is to collect these key-values with the same $subgroup_{size=1}$, aggregating the occurrence frequency to obtain the overall results. At the end, the frequency of occurrence of each value is obtained with the form *value*(*frequency*): $A_1(3), A_2(1), B_2(2), B_1(2), C_2(3), C_1(1)$. Returning this list to the driver.

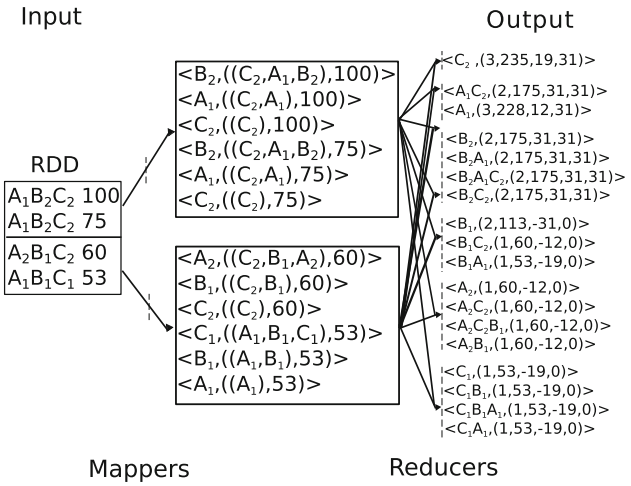

Fig. 2 Example of execution for PFP-SD-OE Step 4. The number of mapper is limited by the RDD previously created, whereas the number of reducer is limited by the number of items in F-list, as this list contains six different items, six reducers have been required

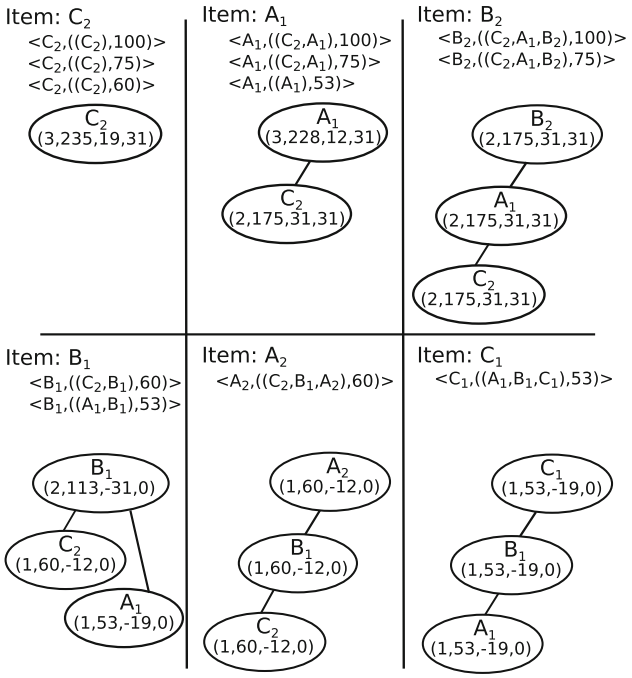

Fig. 3 Example of FP-Tree local created in each reducer. These FP-Trees are built using the item-dependent transactions. As it could be seen, each FP-Tree is totally independent for the rest one; thus, each one could be built and mined separately, enabling a higher level of parallelism

Following the Step 3, this list is sorted by frequency

This step is performed in the driver, since the size of this list is smaller enough to save in main memory.

Now, the Step 4 is performed as shown in Fig. 2. The intermediary $\langle k, v\rangle$ are the form of $\langle item, (tupleItemDependantTransaction, targetValue)\rangle$, which are received in the reducer depending on its $item$ value, and in this way, all the instances with the same item are analyzed in the same reducer. As F-list contains six different values, six different reducers are needed, producing key-values with the form $\langle subgroup, (counterFreq, additionTargetValues, quality, OE)\rangle$. For instance, the key-value $\langle C_2, (3, 235, 19, 31)\rangle$ is produced, since it appears three times in the dataset, 235 is the addition of $100+75+60$, 19 is the quality as $3(\frac{235}{3}-72)$ and 31 is the OE as $(100-72)+(75-72)$. 72 is the overall mean in the dataset.

Each reducer builds a local FP-Tree, as shown in Fig. 3 and then discover subgroups from it using a recursive strategy, considering the OE to remove non-promising branches. For each local FP-Tree, j subgroups are mined, all of them have a quality greater than a threshold established by the user.

4 Experimental study

The aim of this section is to study the scalability and the performance of our proposals considering a varied set of dataset. The purpose of this study is threefold:

- Evaluation of the performance when OEs are considered to prune the search space. The tightness of these OEs are also studied.
- Analysis of the computational time for a varied set of algorithms, comprising both sequential and MapReduce algorithms. This study also includes an evaluation of the performance and scalability of our proposals when truly Big Data is considered.
- Analysis of the endeavor of our algorithms on real-world datasets.

It is worth noting that the runtime obtained for all the algorithms in this experimental analysis is the average runtime obtained for 10 different runs. It has been carried out in this way to alleviate the variations in the runtime derived from the resource management.

All the experiments have been run on a HPC cluster comprising 12 compute nodes. Each of the nodes comprised two Intel Xeon E5645 CPUs with 6 cores at 2.4 GHz and 24 GB DDR memory. Cluster operating system is Rocks cluster 6.1×64 running Spark 2.0.

4.1 Used datasets

This experimental section considers a large number of different datasets. A total of 40 synthetic datasets have been used. Synthetic data have been used since the number of both instances and attributes can be easily changed to illustrate the scalability of the algorithms. In these datasets, the number of instances varies from 10^5 to $2 \cdot 10^8$, whereas the number of attributes ranges from 6 to 16. It is worth mentioning that the search space is exponential in the number of distinct ele-

ments, and these elements are numbered not only by distinct attributes but also by distinct values that each attribute comprises, and thus, this number of attributes produces a search space from $1.17648 \cdot 10^5$ to $1.276 \cdot 10^{15}$ subgroups.

Finally, several executions on 30 real-world datasets have also been considered to prove that our algorithms could work in real data in a efficient way.

Several quality thresholds (α) have been used depending on the quality measure. For binary a value of $\alpha_1 = 8.0 \cdot 10^{-1}$ has been used, whereas for numeric targets and nominal targets values of $\alpha_2 = 8.0 \cdot 10^2$ and $\alpha_3 = 5.0 \cdot 10^{-5}$ have been used, respectively. It is noteworthy to mention that these thresholds are so varied, since each one is applied in a different metric having each one a totally different range of values.

4.2 Optimistic estimates: pruning the search space

The aim of this study is to analyze the effects of considering OEs. Firstly, a comparative study between the same algorithms with and without OE is performed. Then, the tightness of OEs is also studied.

Binary targets In this first analysis, only binary targets are considered. The quality measure is q_{ps} and the tight optimistic estimate is OE_{ps}. Figure 4a shows the results when our proposals are run with and without OE. As it could be seen, the behavior of AprioriK-SD-OE is better than AprioriK-SD, achieving one order of magnitude in runtime better. It is noteworthy to mention that the number of subgroups does not affect. Obviously, when the number of subgroups to extract is close to the number of possible subgroups in the dataset, the performance would be similar, but never worse. The same endeavor happens with PFP-SD-OE with respect to PFP-SD. Both algorithms with OE achieve the best performance. It is also relevant to note that PFP-SD obtains a worse performance than AprioriK-SD, however, when OEs are considered the contrary happens. It is due to the fact that PFP-SD needs to build an FP-Tree where any branches could be pruned being faster extracts directly subgroups.

Numeric targets The quality measures is q_{imp} and the optimistic estimate is OE_{imp}. Figure 4b shows the results when our proposals are run with and without OE. As it could be seen, the same as before occurs. PFP-SD obtains worse results than AprioriK-SD since it needs to build and mines a huge FP-Tree, as any OEs is considered in this algorithm any branch could be pruned, resulting in a larger runtime. When an OE is included, as PFP-SD-OE, the building and mining of an FP-Tree is justified since many branches could be pruned resulting in an considerable improvement in the runtime. It should be highlighted that algorithms based on FP-Tree are meaningless when the number of subgroups is very low, since the building of an FP-Tree is not justified being faster directly extracts the subgroups as AprioriK-SD-OE does.

Nominal targets The quality measure has been q_{gini} and the OE has been OE_{gini}. Figure 4c shows the results when our proposals are run with and without OE. The behavior is the same as the shown previously.

Readers should observe that if Fig. 4a, b is compared, it shows the importance of considering tight OEs. The tightness the OEs, the better the performance of all algorithms, but specially of AprioriK-SD-OE. As OE_{ps} is tight, AprioriK-SD-OE obtains a performance similar to PFP-SD-OE in binary targets, but as OE_{imp} is not tight, there is more difference between PFP-SD-OE and AprioriK-SD-OE in numeric targets. As OE_{gini} is less tight than OE_{imp} (see Fig. 4b, c), the performance of AprioriK-SD-OE is not similar to PFP-SD-OE in nominal targets. It is caused because AprioriK-SD-OE is not able to prune as much as PFP-SD-OE does. Whereas that PFP-SD-OE is able to prune many branches and mines only alluring branches, AprioriK-SD-OE needs to calculate each l and filter after that. Thus, OEs are more effective on algorithms based on FP-Trees.

Continuing the study, the orders of magnitude are analyzed. Demonstrating that the use of OE is recommended (see Table 1). As it could be seen, in binary target, the OE is defined as tight, being the difference always of 1.0 order of magnitude better than algorithms with OEs. When the OE is not tight, as in nominal or numeric, AprioriK-SD does not obtain always one order of magnitude, because this OE is not able to prune as much search space as a tight OE. Being one advantage of PFP-SD-OE that is not as affected as AprioriK-SD-OE by the tightness of the OEs.

To sum up, OEs have proved to prune the search space improving substantially the performance, the tightness the OE, the better. Although, AprioriK-SD-OE is more affected by the non-tightness of the OEs, the use of OEs is justified in all the algorithms because it improves their performances.

4.3 Scalability

In this study, the scalability of our proposals is analyzed. In this regard, a set of synthetic datasets have been used to analyze how the number of both instances and attributes affects the performance. These datasets comprises a search space of $11.76 \cdot 10^4$ to $12.75 \cdot 10^{14}$ subgroups. Only algorithms with OEs have been considered in this study, since it has been proved that these algorithms obtain better performance. These algorithms have been selected since they are able to work in the same kind of target as our proposals. Each of these algorithms has been briefly described as follows:

– *SD-MAP* [13] this algorithm uses a highly efficient data structure to speed up the discovery of subgroups. It only works in binary targets.

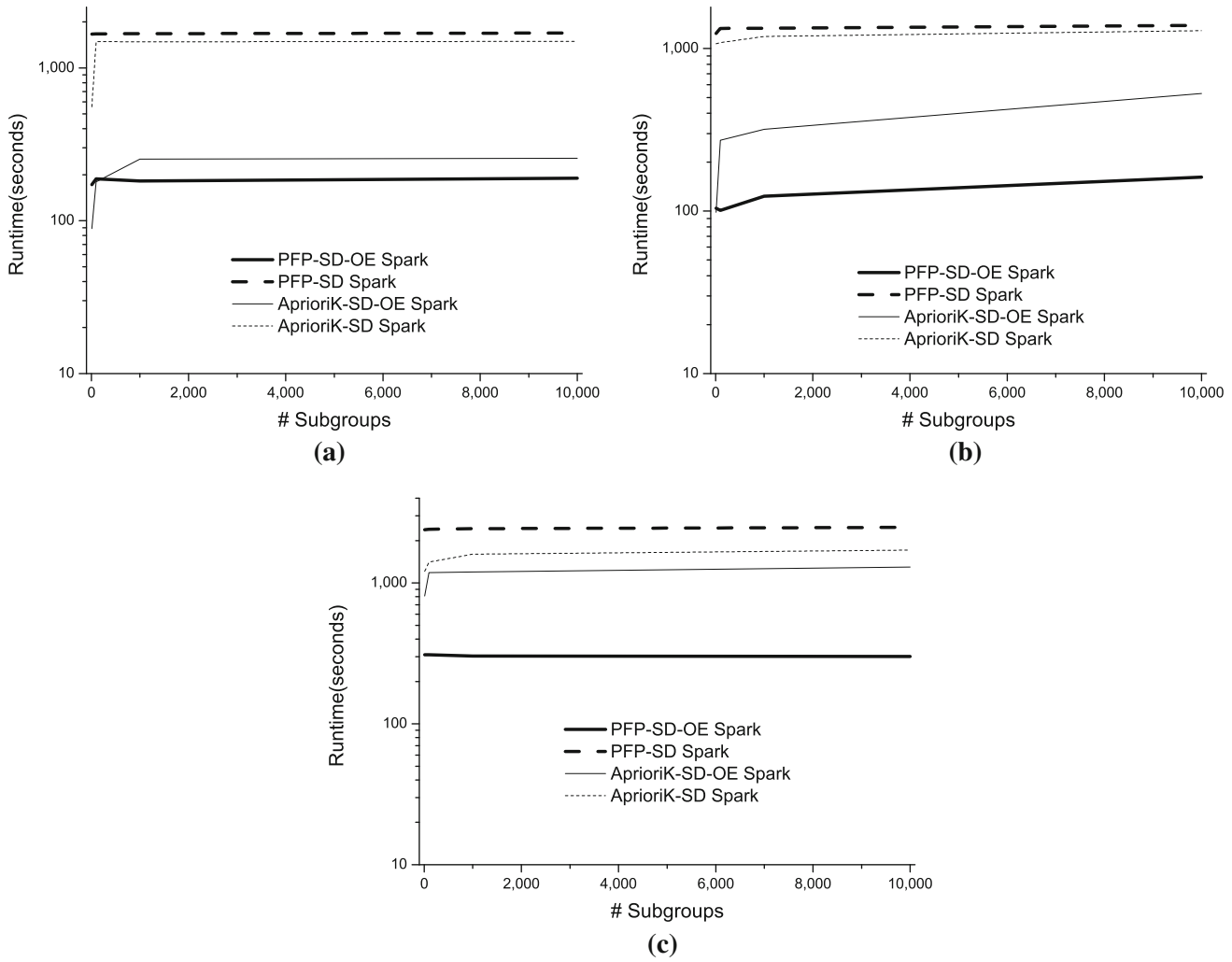

Fig. 4 Runtime considering datasets with 10^6 instances and a search space of $1.38413 \cdot 10^{10}$ subgroups. **a** Binary target with a quality threshold of α_1, **b** Numeric target with a quality threshold of α_2, **c** Nominal target with a quality threshold of α_3

Table 1 Results show the orders of magnitude in the difference of the runtime when each pair of algorithms is compared

Target	Comparisons	Average	Worst	Best
Binary	AprioriK-SD-OE versus AprioriK-SD	1.00	1.00	1.00
	PFP-SD-OE versus PFP-SD	1.00	1.00	1.00
Numeric	AprioriK-SD-OE versus AprioriK-SD	0.75	0.00	1.00
	PFP-SD-OE versus PFP-SD	1.00	1.00	1.00
Nominal	AprioriK-SD-OE versus AprioriK-SD	0.25	0.00	1.00
	PFP-SD-OE versus PFP-SD	1.00	1.00	1.00

The dataset has a search space of $1.38413 \cdot 10^{10}$ subgroups. The average, worst and best case are shown

– *DpSubgroup* [18] based on SD-MAP shares the same approach of using the same data structure, but nominal targets are considered in it.
– *SD-MAP** [16] this algorithm is based on the previous one but considering numeric targets.

These algorithms have been implemented as they are described by their respective authors. The quality and the optimistic estimates functions used in these algorithms are the same as the used by our proposals.

Binary targets In the first place, only binary targets are considered. Figure 5a shows the results when the number of instances changes between 10^5 and $2 \cdot 10^8$. The behavior of AprioriK-SD-OE Spark and SD-MAP is almost the same when the number of instances is up to $1.2 \cdot 10^7$. Although SD-MAP and AprioriK-SD-OE Spark obtain a runtime almost

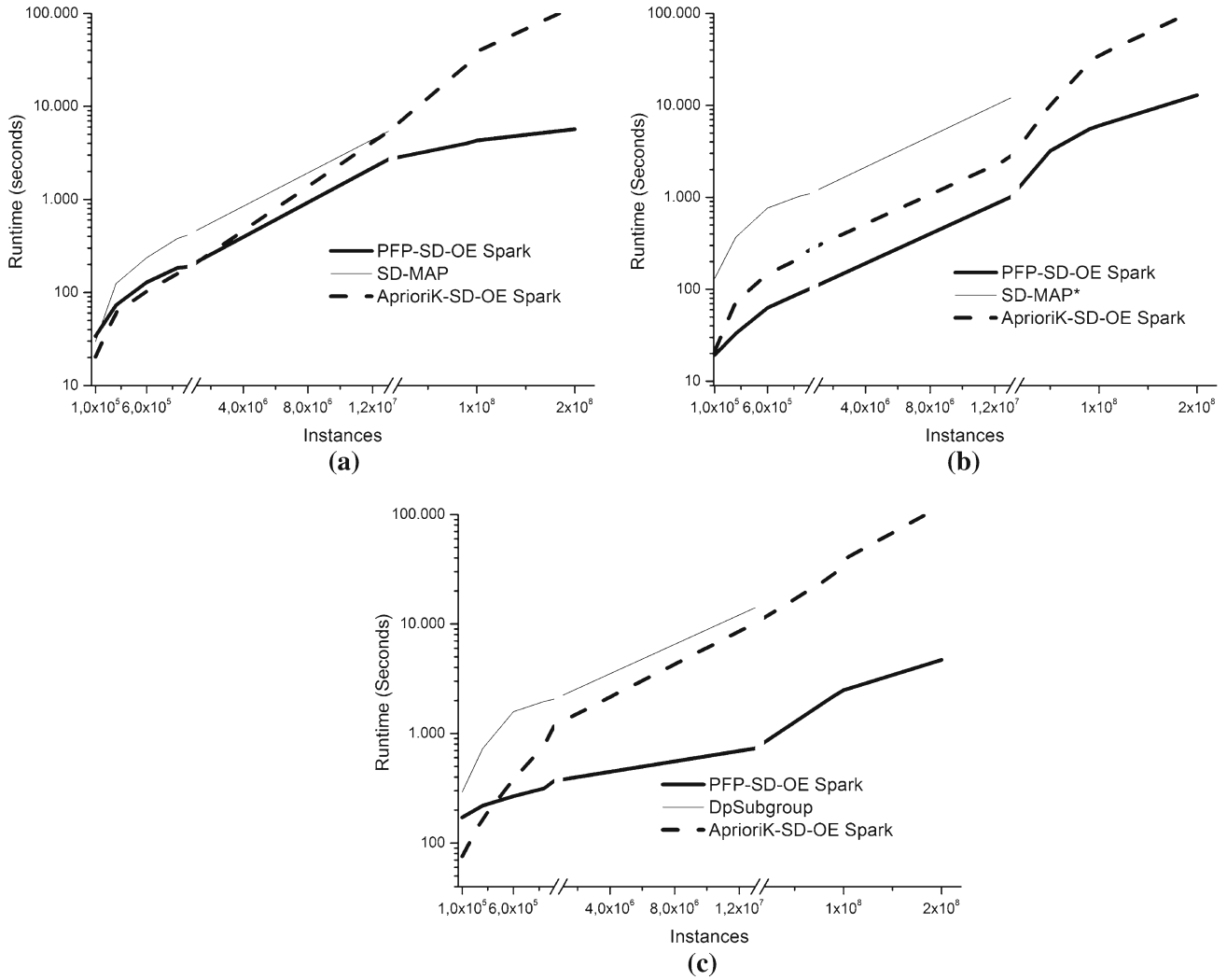

Fig. 5 Runtime on data with a number of instances from 10^5 to $2 \cdot 10^8$ and a search space of $1.38 \cdot 10^{10}$ subgroups. **a** Binary target is considered. **b** Numeric target is considered. **c** Nominal target is considered

equal when a very large number of instances is considered, ApriorikK-SD-OE Spark allows to scale horizontally, whereas that SD-MAP as a sequential approach needs a better hardware with larger quantities of RAM to continue executing. SD-MAP needed more than 24 GBytes of RAM only to run a dataset with 10^6 instances. Finally, it should be noted that PFP-SD-OE Spark obtains the best performance when a large number of instances are considered, when this number is lower ApriorikK-SD-OE Spark obtains better performance than PFP-SD-OE Spark. This last fact is due to that the tackled search space is not bigger enough to justify the time invested in building a FP-Tree, but it could be more efficient to mine directly the subgroups as ApriorikK-SD-OE does. When the number of instances is increased again up to $2 \cdot 10^8$ instances, sequential algorithms are not able to run in a reduced quantum of time, thus only algorithms based on Spark could be run. As it could be seen, the behavior is the same, PFP-SD-OE Spark overcomes ApriorikK-SD-OE Spark with this huge number of instances.

Numeric targets In this case, only numeric targets are considered. Figure 5b shows the results when the number of instances varies from 10^5 to $2 \cdot 10^8$. The behavior of the algorithms is almost the same as the shown with binary targets since the approach is shared. PFP-SD-OE Spark allows to discover subgroups in the most efficient way; it only needs an 8.35 % of time with respect to SD-MAP* to discover the same subgroups on a dataset with 10^6 instances. As it has been noted in previous studies, the tightness of OE_{imp} is lower than OE_{ps} results in a bigger search space. This could be appreciated in two different ways: (1) ApriorikK-SD-OE Spark does not have a similar behavior than SD-MAP*, having SD-MAP* a worse performance since it needs to tackle a bigger search space than SD-MAP, thus a parallel approach

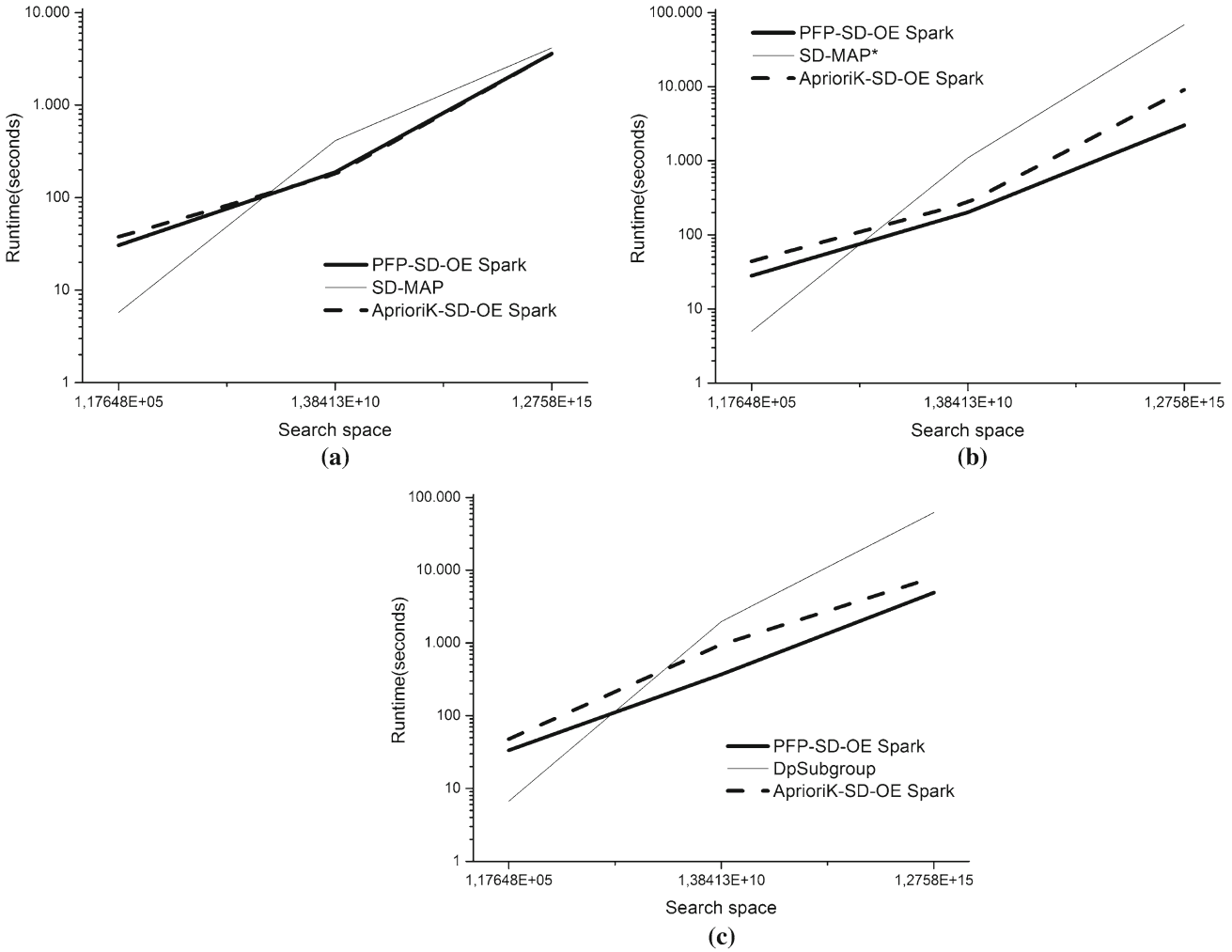

Fig. 6 Runtime on data with 10^6 instances and a search space that varies from $1.17 \cdot 10^5$ to $1.27 \cdot 10^{15}$ subgroups. **a** Binary target is considered. **b** Numeric target is considered. **c** Nominal target is considered

as ApriorIK-SD-OE Spark is preferred; (2) the performance of algorithms based on Spark with a number of instances bigger than $2 \cdot 10^7$ is more similar, since the approach based on FP-Tree is not able to prune as much search space as happening with a tight OE. As the search space is bigger, sequential algorithms are not able to work efficiently, whereas that algorithms based on Spark tackle these bigger search spaces by means of parallelism.

Nominal targets In this case, only nominal targets are considered. The behavior of the algorithms is almost the same as shown previously (see Fig. 5c). Again, it could be appreciated that as OE_{gini} is the least tight, the performance among algorithms is more varied than in the previous cases.

Continuously, this study, the influence of attributes in the performance is also studied in function of the kind of target.

Binary targets Figure 6a shows the results when the number of instances is 10^6 and a varied set of search spaces. As it could be noted, PFP-SD-OE Spark and ApriorIK-SD-OE have a very similar behavior since the OE is tight, thus, both algorithms are able to prune almost the same search space. As happened in previous studies, SD-MAP obtains a very similar behavior as ApriorIK-SD-OE, but the last one enables to scale out, whereas the first one only is able to scale up.

Numeric targets In concrete, Fig. 6b shows the results when the number of instances is 10^6 and a varied set of search spaces. As it could be noted, PFP-SD-OE Spark obtains almost always the best performance, excepting with very small datasets. It is caused because datasets with a small search space produces smaller FP-Tree, thus it could be pointless to try to parallelize the mining process.

Nominal targets As it could be appreciated, the endeavors are the same as the previous studies, in this kind of dataset the OE is the least tight (see Fig. 6c).

4.4 Real-world datasets

The aim of this section is to demonstrate that our proposals are able to mine efficiently subgroups in real-world datasets. This analysis includes a varied set of 30 dataset, considering a number of instances ranging from 900 to more than $4 \cdot 10^7$ instances. As our proposals are exhaustive search algorithms, where all the possible subgroups are examined, any kind of comparison about effectiveness is required since our algorithms guarantee that the best solutions found into the dataset will be returned. In the same regard, as exhaustive search algorithms have been used, the continuous attributes have been discretized using unsupervised techniques [15]: equal-width (EW) and equal-frequency (EF). As no information is known previously, different bins have been used.

Table 2 shows the results obtained, where the labels mean: *Dataset* is the name of the dataset, *instances* is the number of examples, *#Attrs*$(R/I/N)$ is the number of attribute (R represents real attributes, I is used for integer attributes and N for nominal attributes), *Discretization* mean the kind of discretization considered, and the results of runtime for both algorithm in seconds. It also should be noted that EF means that continuous attributes have been discretized using equal-frequency with 4 bins and EW means equal-width using 4 bins. Different bins have been proved, but due to space limitations, only 4 bins have been shown. The rest of results considering different bins is available at http://www.uco.es/grupos/kdis/kdiswiki/SDBigData. In some cases, the process of discretization has not been required since the dataset only contains nominal attributes.

As it could be appreciated, the obtained results are equivalent to the shown previously. Whereas that AprioriK-SD-OE is highly affected by the number of attributes, PFP-SD-OE is not so hampered. This fact could be appreciated in these datasets with a high number of attributes such as Mushroom, Thyroid or Twonorm. It should also be highlighted that a higher number of attributes means a higher search space to tackle, where the search space grows exponentially with the number of attributes. Not only is the number of attributes

Table 2 Runtime in seconds of our proposals when real-world datasets are considered

Dataset	Instances	#Attrs (R/I/N)	Discretization	AprioriK-SD-OE	PFP-SD-OE
Mammographic	961	5 (0/5/0)	EF	6.1	7.1
			EW	5.8	6.5
Tic-Tac-Toe	958	9 (0/0/9)	–	7.2	8.5
Vowel	990	13 (10/3/0)	EF	19.3	12.5
			EW	14.1	7.5
Flare	1066	11 (0/0/11)	–	10.4	9.1
Yeast	1484	8 (8/0/0)	EF	18.3	11.6
			EW	12.2	7.2
Contraceptive	1473	9 (0/9/0)	EF	13.7	8.1
			EW	12.2	7.8
Car	1728	6 (0/0/6)	–	6.4	8.5
Titanic	2201	3 (3/0/0)	EF	8.5	7.2
			EW	11.3	10.1
Abalone	4174	8 (7/0/1)	EF	13.2	9.4
			EW	12.6	7.9
Wine	4898	11 (11/0/0)	EF	14.8	9.4
			EW	13.6	7.6
Banana	5300	2 (2/0/0)	EF	11.3	10.4
			EW	8.0	6.8
Phoneme	5404	5 (5/0/0)	EF	10.1	7.1
			EW	95.5	8.0
Page-blocks	5472	10 (4/6/0)	EF	13.9	9.2
			EW	12.6	7.3
Marketing	6876	13 (0/13/0)	EF	24.7	11.7
			EW	15.9	7.6
Thyroid	7200	21 (6/15/0)	EF	8325.5	1706.0
			EW	715.0	47.45

Table 2 continued

Dataset	Instances	#Attrs (R/I/N)	Discretization	ApriorIK-SD-OE	PFP-SD-OE
Twonorm	7400	20 (20/0/0)	EF	34.0	12.3
			EW	378.9	8.1
Ring	7400	20 (20/0/0)	EF	19.9	11.7
			EW	328.1	8.1
Mushroom	8,124	22 (0/0/22)	–	$17.37 \cdot 10^3$	12.57
Penbased	10,992	16 (0/16/0)	EF	132.1	14.0
			EW	40.4	8.0
Nursery	12,960	8 (0/0/8)	–	12.6	11.9
EEG eye state	14,980	14 (14/0/0)	EF	86.1	19.6
			EW	21.1	8.0
Magic	19,020	10 (10/0/0)	EF	28.8	10.0
			EW	13.34	8.1
Letter	20,000	16 (0/16/0)	EF	524.4	52.1
			EW	63.9	8.2
krvsk	28,056	6 (0/0/6)	–	13.3	11.0
Adult	30,162	14 (6/0/8)	EF	94.9	32.3
			EW	12.8	8.3
Shuttle	43,500	9 (0/9/0)	EF	15.3	9.9
			EW	12.8	8.3
Skin segmentation	245,057	4 (0/4/0)	EF	15.5	8.3
			EW	9.4	8.3
Poker	1,025,010	10 (0/0/10)	–	72.8	46.3
SUSY	5,000,000	18 (18/0/0)	EF	$17.37 \cdot 10^4$	69.1
			EW	29.4	27.7
Activity recognition	43,930,257	10 (6/0/4)	EF	462.7	69.1
			EW	330.5	69.7

A quality threshold of α_3 has been used, and 100 subgroups have been extracted for each dataset. $\#Attrs$ means the number of attributes, where R is used to refer to real attributes, I means integer attributes and N refers to nominal attributes

important, but also the distribution of the data since it will determine the search space. By this very reason, the same dataset with different kinds of discretization could result in a different search space. One example of this fact could be Twonorm, where with equal-frequency the search space is smaller than equal-width. This fact is not dependent to the used discretization, since in Ring dataset the contrary happens.

Likewise, as happening in previous studies PFP-SD-OE is not as much hampered as ApriorIK-SD-OE by the number of instances. If the runtime for the dataset with a highest number of instances is analyzed, it could be noted that ApriorIK-SD-OE required more than 669 % of the runtime needed by PFP-SD-OE (see Table 2, ActivityRecognition dataset). However, if the smallest dataset is analyzed, it could be noticed that ApriorIK-SD-OE requires a lower runtime than PFP-SD-OE (see Table 2, Mammographic dataset).

5 Conclusion

In this work, two new efficient exhaustive search algorithms have been proposed to mine subgroups on Big Data, relying on the MapReduce framework. On one hand, ApriorIK-SD-OE is an iterative algorithm based on Apriori [5]. On the other hand, PFP-SD-OE is an algorithm using a highly efficient data structure based on PFP-Growth [24], SD-MAP [13], SD-MAP* [16] and DpSubgroup [18]. The two proposed algorithms have proved to be highly efficient in mining subgroups on Big Data, and they are able to work with binary, numeric and nominal targets.

Both proposals have been implemented in Apache Spark. This platform has been selected among others since it has proved to obtain excellent results in iterative algorithms [10]. Both are able to prune the search space by means of optimistic estimates, without losing any subgroups. Whereas that ApriorIK-SD-OE is able to work better when a huge or very

low number of subgroups are extracted, PFP-SD-OE is able to obtain subgroups in a more efficient way when the number of subgroups is not so huge or so low.

All the algorithms have been compared to other highly efficient algorithms in the SD field depending of the kind of target, showing that our proposals outperform them when Big Data is taken into account. The experimental stage revealed that the proposed algorithms perform really well when both the search space increases (a maximum search space of $1.276 \cdot 10^{15}$ subgroups have been used) and the number of instances drastically increases, achieving a good performance even for datasets comprising up to $2 \cdot 10^8$ different instances.

Acknowledgements This work was supported by the Spanish Ministry of Economy and Competitiveness and FEDER funds, Project TIN-2014-55252-P.

References

1. Wu, X., Zhu, X., Wu, G.Q., Ding, W.: Data mining with big data. IEEE Trans. Knowl. Data Eng. **26**(1), 97–107 (2014)
2. Han, J., Kamber, M.: Data Mining: Concepts and Techniques. Morgan Kaufmann, San Francisco (2011)
3. Herrera, F., Carmona, C.J., González, P., Jesus, M.J.: An overview on subgroup discovery: foundations and applications. Knowl. Inf. Syst. **29**(3), 495–525 (2010)
4. Ventura, S., Luna, J.M.: Pattern Mining with Evolutionary Algorithms. Springer, Berlin (2016)
5. Agrawal, R., Imieliński, T., Swami, A.: Mining association rules between sets of items in large databases. SIGMOD Rec. **22**(2), 207–216 (1993)
6. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. SIGMOD Rec. **29**(2), 1–12 (2000)
7. Luna, J.M., Romero, J.R., Romero, C., Ventura, S.: On the use of genetic programming for mining comprehensible rules in subgroup discovery. IEEE Trans. Cybernet. **44**(12), 2329–2341 (2014)
8. Scheffer, T., Wrobel, S.: Finding the most interesting patterns in a database quickly by using sequential sampling. J. Mach. Learn. Res. **3**, 833–862 (2003)
9. Grosskreutz, H., Rüping, S., Wrobel, S.: Proceedings, Part I European Conference, ECML PKDD 2008, Antwerp, Belgium, September 15-19, 2008. Tight Optimistic Estimates for Fast Subgroup Discovery (Berlin, Heidelberg, 2008) pp. 440–456 (2008)
10. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, Ser. HotCloud'10, Berkeley (2010)
11. Klösgen, W.: Advances in knowledge discovery and data mining. In: Fayyad, U.M., Piatetsky-Shapiro, G., Smyth, P., Uthurusamy, R. (eds.) Explora: A Multipattern and Multistrategy Discovery Assistant, pp. 249–271. American Association for Artificial Intelligence, Menlo Park (1996)
12. Kavšek, B., Lavrač, N., Jovanoski, V.: 5th International Symposium on Intelligent Data Analysis, IDA: ch, pp. 230–241. APRIORI-SD, Adapting Association Rule Learning to Subgroup Discovery (2003)
13. Atzmueller, M., Puppe, F.: Sd-map-a fast algorithm for exhaustive subgroup discovery. In: 17th European Conference on Machine Learning and 10th European Conference on Principles and Practice of Knowledge Discovery in Databases (ECML/PKDD 2006). Lecture Notes on Computer Science, vol. 4213, pp. 6–17. Springer (2006)
14. Klösgen, W.: Advances in knowledge discovery and data mining. In: Fayyad, U.M., Piatetsky-Shapiro, G., Smyth, P., Uthurusamy, R. (eds.) Explora: A Multipattern and Multistrategy Discovery Assistant. American Association for Artificial Intelligence, Menlo Park (1996)
15. García, S., Luengo, J., Herrera, F.: Data Preprocessing in Data Mining. Springer, Switzerland (2015)
16. Lemmerich, F., Atzmueller, M., Puppe, F.: Fast exhaustive subgroup discovery with numerical target concepts. Data Min. Knowl. Discov. **30**(3), 711–762 (2015)
17. Atzmueller, M., Lemmerich, F.: Fast subgroup discovery for continuous target concepts. In: Foundations of Intelligent Systems, pp. 35–44. Springer, Berlin (2009)
18. Grosskreutz, H., Rüping, S., Wrobel, S.: Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2008, Antwerp, Belgium, September 15-19, 2008, Proceedings, Part I. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, ch. Tight Optimistic Estimates for Fast Subgroup Discovery, pp. 440–456
19. Rajaraman, A., Ullman, J.D.: Mining of Massive Datasets. Cambridge University Press, New York (2011)
20. Padillo, F., Luna, J.M., Cano, A., Ventura, S.: A data structure to speed-up machine learning algorithms on massive datasets. In: Proceedings of the 11th International Conference on Hybrid Artificial Intelligence Systems, ser. HAIS 2016, Seville, Spain, pp. 365–376 (2016)
21. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM - 50th anniversary issue: 1958 - 2008, **51**(1), 107–113 (2008)
22. Lam, C.: Hadoop in Action, 1st edn. Manning Publications Co., Greenwich (2010)
23. Luna, J.M.: Pattern mining: current status and emerging topics. Prog. Artif. Intel. **5**(3), 1–6 (2016)
24. Li, H., Wang, Y., Zhang, D., Zhang, M., Chang, E.Y.: Pfp: Parallel fp-growth for query recommendation. In: Proceedings of the 2008 ACM Conference on Recommender Systems, ser. RecSys '08. New York, NY, USA: ACM, pp. 107–114 (2008)