Discovery of Action Rules at Lowest Cost in Spark

Angelina A. Tzacheva, Arunkumar Bagavathi, Lavanya Ayila
Department of Computer Science
University of North Carolina at Charlotte, NC
{aatzache, abagavat, layila}@uncc.edu

***Abstract*— Action Rules or Actionable patterns is a type of rule-based approach in data mining that recommends to a user specific actions, in order to achieve a desired result or goal. The amount of data in the world is growing at an exponential rate, doubling almost every two years. Distributed computing platforms like Hadoop and Spark, have eased the computation of this high velocity data. Leveraging these cutting-edge technologies in the field of Data Mining to process huge volumes of data can improve the performance and allow user to gain insights from large datasets with quick turnaround time. In this paper, we present an approach for discovering low cost actionable patterns, and provide actionable recommendations. We adapt this algorithm to distributed environment using Apache Spark framework. We evaluate the performance of the algorithm with two datasets in transportation and medical domain.**

Keywords*— *Data Mining, Action Rules, Apache Spark, Cost of Action Rules

I. Introduction

Data Mining is the process of analyzing large datasets to find interesting behavioral patterns or hidden opportunities from the underlying data in order to transform and improve. The advent of the Internet and digitization of data has led to overwhelming increase of data in many fields including commercial, medical, industrial, and financial. Data mining techniques aid organizations and users to analyze this ever-expanding universe of digital data and get the most out of the data without getting lost in detail. Wide ranges of sophisticated data mining algorithms are available to classify the data, cluster the data, and identify associations or to find sequential patterns within the data. Though these algorithms produce useful information from the data, they do not focus on providing actionable knowledge that help users to precisely identify lucrative actions and make better decisions to maximize opportunities. Action rules addresses this problem by providing actionable solutions that describes possible transition of objects from one state to another, to improve the condition of an object with respect to a distinguished attribute called a decision attribute [1]. The attributes used to describe the transition of class or decision attribute can group into two groups namely – Stable Attributes and Flexible Attributes. Values of stable attributes cannot form actionable patterns and acts more like constants, whereas the values of flexible attributes are open to changes and contribute for actionable solutions to for the transition of decision attribute.

Action Rule patterns are interesting, if the extracted rules are diverse. The patterns are diverse if its elements differ significantly from each other. The more diverse the extracted patterns are, the more interesting they are. Little research has been done to measure interestingness of association rules [17] or action rules. Action Rules of low cost are considered patterns of high interest. Action Rules costs the user or company in some form of money or resources like energy, power and human resources to make the recommended changes to achieve the desired action or goal. However, most of the Action Rules extraction algorithms does not guarantee the cost economic recommendations to the user or company. To address this problem, Tzacheva et.al. [5] provided heuristics for finding cost and feasibility of the discovered Action Rules. They suggest a heuristic strategy for creating new Action Rules, where objects supporting the new Action Rule also support the initial Action Rule but the cost of reclassifying them is lower or even much lower for the new rule. In this way, the rules constructed are of more interest to the users and in the context of a business action plan. However, the data deluge in recent times poses challenges with the current algorithm's implementation to store, process, analyze the huge volume of data, and produce results within acceptable time limits. Hence, there is a need to rely on the capabilities of distributed computing and parallel processing techniques to process such datasets in near real time and monetize the Action Rules generated.

Many open source frameworks like Apache Hadoop, Apache Spark, Hive, Storm etc. are available today to perform distributed processing on such huge volumes of data on clusters of machines in a distributed fashion. Hadoop MapReduce [9] is a framework that is more suitable for batch processing, but carries the overhead of multiple disk read writes to the distributed file system. This makes MapReduce inappropriate for many data mining and machine learning algorithms. Spark [2] on the other hand, is an in-memory distributed data analysis and cluster computing platform designed to process large datasets using a single programming model. It can achieve up to a 100 times faster processing speed than Hadoop map-reduce when running in-memory and up to 10 times faster when running on disk.

The design goal of Apache Spark is to reduce disk access and provide more support to the iterative algorithms. It also supports batch jobs, machine learning jobs, and interactive querying and includes real-time data stream processing. It uses DAG (Directed Acyclic Graph) engine to optimize its workflows and to provide fault tolerance through the concept of lineage. In this paper, we use the power of Apache Spark to extract low cost Action Rules. We also compare the new method with the algorithm working in non-distributed fashion in terms of performance and the quality of low cost Action Rules recommendations given by the algorithms.

II. RELATED WORK

Ras and Wyrzykowska [1] initially introduced the notion of Action Rules and brings in the idea of generating special type of rules, which provides suggestions to re-classify objects with respect to decision attribute, from a database. Further, in the paper by Tzacheva and Ras [5], the notion of a cost and feasibility of an Action Rule was proposed. They suggest a heuristic strategy for creating new Action Rules, where objects supporting the new Action Rule also support the initial Action Rule but the cost of reclassifying them is lower or even much lower for the new rule. In this way, the rules constructed are of more interest to the users.

A number of algorithms are available in the literature for Action Rules extraction. Ras and Tsay [14] proposed a strategy to extract Action Rules with the help of two classification rules. Ras and Wyrzykowska [6] proposed a new simplified strategy for Action Rule extraction. This strategy suggests that, extracting Action Rules no longer requires pairs of classification rules, but rather “grab” the objects. Im, et.al [15] proposed a new approach for extracting simple Action Rules directly from the given data without producing any classification rules. Mining Action Rules from scratch [3], which presents an exhaustive method, would supply us with all important rules. Clearly, the space of such rules is quite huge, so a generalization technique, such as creating summaries, would provide great means for reducing the space and furnish the user with the essence of the actionable knowledge.

Tzacheva [4] introduced a generalization technique, which creates summaries of Action Rules, by utilizing an exhaustive method. The author provided great means for reducing the output space and furnished the user with the essence of the actionable knowledge. The paper introduced the notion of diversity of Action Rule summaries, and suggested removing the high cost rules, when creating summaries, in order to additionally decrease the space of Action Rules.

Action Rules of low cost are interesting patterns, because they are both actionable patterns, and they help the user to achieve their goals at lowest known cost. The cost of applying the suggested actions, can take the form of money or resources such as monetary cost, energy consumption, human resources, or even moral costs, in order to make the recommended changes to achieve the desired goal, or benefit. In this work, we adapt a lowest cost Action Rule mining method by Tzacheva et.al. [5] to a distributed framework for processing. We propose this distributed implementation to address the ever growing size of data, and the advent of BigData, since the currently existing algorithm does not support such large amounts of data.

In this work, we present an approach, to compute Action Rules of lowest cost for large datasets using Spark distributed framework to adapt the algorithm, and provide scalability with very large datasets. Performance and computational efficiency of the algorithm are studied. We perform experiments with the adapted Action Rules at lowest cost algorithm with Car Evaluation Dataset, and Mammographic mass Datasets in a distributed setup using Apache Spark.

III. METHODOLOGY

We implement the proposed algorithm in [11] for distributed discovery of Action Rules at lowest cost using Apache Spark framework [2] and Hadoop Distributed File System (HDFS) [7] thereby, improving the computational performance, fault tolerance and high availability of the system. In the upcoming sections, we give some background knowledge about Action Rules, a detailed overview of distributed computing frameworks, algorithm to extract Action Rules using such distributed computing frameworks and algorithms for recommending lowest cost Action Rules.

A. Action Rules

In this section, we define Action Rules and give an example of extracting them.

An information system S in “Table I” is defined in “Equation (1)”:

$$S = (X, A, V_A) \tag{1}$$

where,

X is a set of objects: $X = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}$
A is a set of attributes: $A = \{a, b, c, d\}$ and
V_A represents a set of values for each attribute in A.

For example, $V_B = \{b_1, b_2\}$

TABLE I. INFORMATION SYSTEM S

X	**a**	**B**	**c**	**D**
x_1	a_1	b_1	c_1	d_1
x_2	a_3	b_1	c_1	d_1
x_3	a_2	b_2	c_1	d_2
x_4	a_2	b_2	c_2	d_2
x_5	a_2	b_1	c_1	d_1
x_6	a_2	b_2	c_1	d_2
x_7	a_2	b_1	c_2	d_2
x_8	a_1	b_2	c_2	d_1

We extract Action Rules from a special variety of an information system called Decision Table. Information system S becomes Decision table S_D, when we divide the attribute space in S into three different types attributes: Stable Attributes (A_S), Flexible Attributes (A_F) and Decision Attribute(s) (A_D). Stable attributes are the one, which cannot change their values and does not form actionable patterns in the Action Rules. However, values in Flexible attributes can change and eventually become actionable patterns. Decision attribute is also a flexible attribute, on which the user wants to change values from d_1 to d_2. Decision attribute in the decision table should satisfy the property of $A_D \notin \{A_S \cup A_F\}$. Thus, attribute space in decision table can take a representation of "Equation (2)":

$$A = \{A_S \cup A_F \cup A_D\} \tag{2}$$

Action Rules from *S* gives a recommendation of changing which attributes value can result in the desired decision action. An example Action Rule for the Information System *S*, considering attribute *c* as a *Stable attribute* and attribute *d* as a *Decision attribute* is given below in "Equation (3)":

$$(a, a_2 \rightarrow a_1) \wedge (b, b_2 \rightarrow b_1) \wedge (c, c_1) \Rightarrow (d, d_1 \rightarrow d_2) \tag{3}$$

The above Action Rule means that if the attribute *a* change its value from a_2 to a_1, the attribute *b* change its value from b_2 to b_1 and the attribute *c* remains at the value c_1, the resulting action ($d_1 \rightarrow d_2$) is possible. Action Rules can find applications in many fields like in the Medical domain, Action Rules can help improving patient's health and in the Business domain, Action Rules can help improving the revenue of a company.

For each extracted Action Rules, we also find their corresponding *Support* and *Confidence*. *Support* specifies how many records in the dataset support the extracted Action Rule. *Confidence* specifies how the Action Rule is confident with the whole dataset. We use "Equation (4)" and "Equation(5)", given in [13], for calculating *Support* and *Confidence* respectively of any Action Rule *R*.

$$Support(R) = card(Y_2 \cap Z_2) \tag{4}$$
$$Confidence(R) = [card(Y_2 \cap Z_2) / card(Y_2)] \tag{5}$$

where,

Y_2 – right side of all atomic action terms in the precedent part of the Action Rule R

Z_2 – right side value of decision action term, which is present in the antecedent part of the Action Rule R

A wide range of distributed frameworks are available that follows the approach of "Moving compute to data" instead of "Moving data to compute", which is highly suitable while dealing with large datasets. Next sections provide a detail overview of the frameworks and the reason why Spark performs the best over other frameworks for current scenario.

B. *Hadoop MapReduce*

J. Dean and S. Ghemawat, from Google, in [9] proposed a distributed computing programming model called MapReduce, which has a potential to perform parallel processing of large datasets on a cluster of nodes or computers in a fault tolerant manner. It aims at providing the right level of abstraction by separating what to do from how to do and lets the user concentrate on how to put their methods or algorithms into MapReduce programming model, while leaving the complex functionalities like data distribution, task parallelization, load balancing and fault-tolerance to be handled by the framework. Hadoop at its core contains two main components: HDFS (Hadoop Distributed File System) and MapReduce. HDFS is a high-bandwidth, highly available, fault-tolerant distributed file system that stores data as chunks on local disks of nodes in the cluster. It preserves data locality by assuring that distance between data node and slave node is minimum.

MapReduce paradigm has two phases: Map and Reduce. In both phases the function takes input and produces output as <*Key*, *Value*> pairs. A Master node is responsible for handling the whole MapReduce execution process. It assigns map and reduce tasks to the worker nodes. Initially, the master node runs its map phase on worker nodes marked as mappers. As the map function runs on a small chunk of a large dataset, the results generated from this phase are usually intermediate results. At the end of map phase, each map function writes its intermediate output of <*Key*, *Value*> pairs back into HDFS. The worker nodes now marked as reducers, collects the intermediate results, sorts them, and performs computations like aggregation on the collection. One or more reduce functions collects all <*Key,Values_list*>. The results from reduce phase are outputted as <Key, Value> pairs and are written on to HDFS. These output results can form as input to any other MapReduce task or other applications to perform other computations. Hadoop achieves fault tolerance and data loss by storing each data chunk across different nodes of a cluster by a replication factor of three. "Fig. 1" provides an overview of MapReduce execution phase is shown.

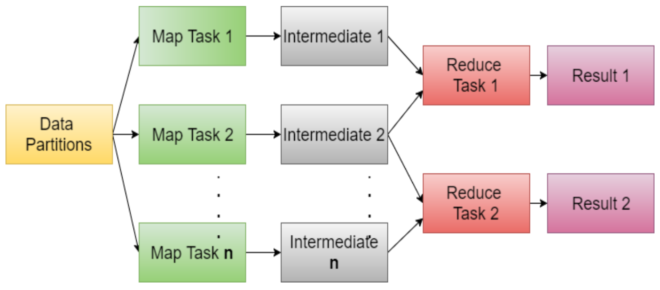


Fig. 1. Overview of Hadoop MapReduce execution.

C. *Spark*

Though MapReduce proves to be a good framework to deal with distributed parallel computing operations, it carries disadvantages such as multiple disk read writes for accessing and storing the data for each MapReduce phase. Due to such overhead, iterative machine-learning algorithms like classification and clustering perform poorly on Hadoop. Spark [2] introduces an in-memory distributed data analysis and cloud computing platform that avoids frequent disk access to the data nodes. It achieves this by using Resilient Distributed Datasets (RDD) [8], an abstraction over a giant set of data on which spark allows performing *Transformations* and *Actions*.

Spark reads the data from the input files, split them into different partitions and store them in node memory as an RDD. Thus, RDDs represent a collection of items distributed across many compute nodes that can be manipulated in parallel. The driver node is responsible for allocating the tasks among multiple worker nodes where the data resides. Tasks allotted to worker nodes can be either *Transformations* or *Actions*.

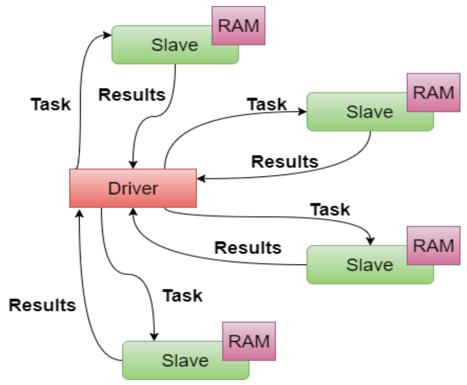


Fig. 2. Overview of Spark execution

Spark is a lazy execution engine. It separates its operation of transformations and actions. The transformations are lazily evaluated or run only when needed. However, action operations execute immediately. Spark transformation operations on an RDD does not show a result until Spark notices an action operation on it. When the Spark execution encounters an action operation, it finds the most efficient way to calculate results thus optimizing the workflow and improving computation efficiency.

When a job is submitted on Spark, it builds an execution plan where it keeps track of all the things that are chained together from different RDDs and how they connect to each other and based on that information it constructs a directed acyclic graph (DAG). In this process, Spark breaks down the job into stages that are created based on chunks of processing that can be done in parallel manner without shuffling things around. Each stage is split into parallelizable tasks which may be distributed across multiple worker nodes. The result of all worker nodes together forms another RDD. Action task collects the resulting RDDs and send the collection to the driver node or save the collection to a storage system like databases. An overview of Spark execution is shown in “Fig. 2”.

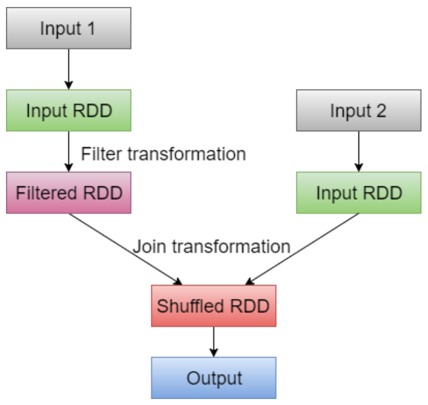


Fig. 3. Sample Lineage Graph in Spark

Spark supports the processing of iterative data science and machine learning algorithms, due to the in-memory computing model, which it employs on RDDs. It handles fault tolerance by maintaining a lineage graph of RDDs. *Lineage graph* is a directed acyclic graph (DAG), which represents the path that the Spark execution (*Transformations* and *Actions*) takes on each node. When a failure occurs at any stage, Spark uses last available working point RDD from the DAG, and restarts all computations from that RDD instead of replicating the data partition across multiple nodes. “Fig. 3” depicts a sample lineage graph of combining RDDs from two inputs. The support for processing iterative algorithms and strategy of fault tolerance, data management makes us to choose Spark for running our algorithm.

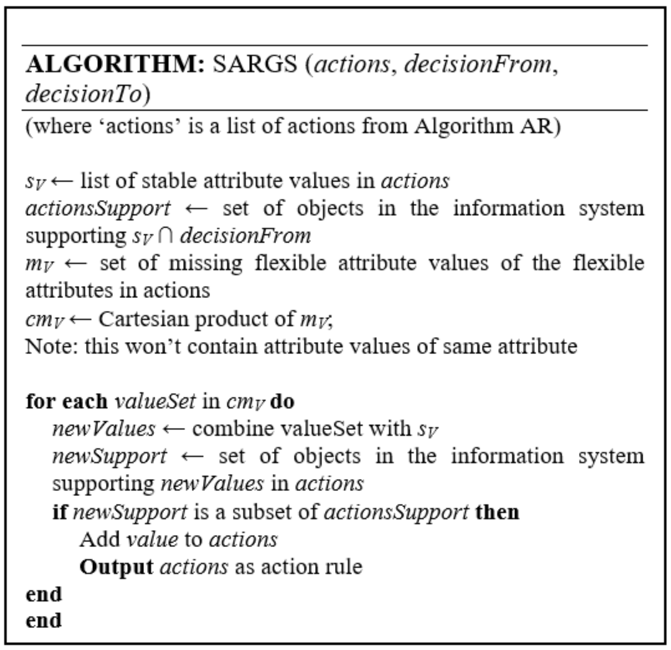
Now we describe the various steps involved in extracting the lowest cost Action Rules, which includes Action Rules extraction from scratch, discuss the existing algorithm to get low cost Action Rules by considering the correlations between individual atomic action sets and presents the proposed algorithm to find low cost Action Rules using the Spark framework.

D. Distributed Actionable Pattern Discovery

In our work, we use the algorithm defined by Bagavathi et. al. [11] to extract Action Rules. The proposed algorithm acts as an alternative to the *ARAS* algorithm implemented in Hadoop MapReduce [12]. Even though the ARAS method extracts Action Rules in reasonable amount of time, it produces some incomplete Action Rules for Information System *S* given in “Table I” like the “Equation (6)” and “Equation(7)”:

$$(a, a_2 \rightarrow a_1) \wedge (b, \rightarrow b_1) \Rightarrow (d, d_2 \rightarrow d_1) \quad (6)$$

$$(a, \rightarrow a_1) \wedge (c,\ c_2) \Rightarrow (d, d_2 \rightarrow d_1) \quad (7)$$



ALGORITHM: SARGS (*actions*, *decisionFrom*, *decisionTo*)

(where ‘actions’ is a list of actions from Algorithm AR)

```
$s_V \leftarrow$ list of stable attribute values in actions
actionsSupport $\leftarrow$ set of objects in the information system supporting $s_V \cap$ decisionFrom
$m_V \leftarrow$ set of missing flexible attribute values of the flexible attributes in actions
$cm_V \leftarrow$ Cartesian product of $m_V$;
Note: this won’t contain attribute values of same attribute

for each valueSet in $cm_V$ do
    newValues $\leftarrow$ combine valueSet with $s_V$
    newSupport $\leftarrow$ set of objects in the information system supporting newValues in actions
    if newSupport is a subset of actionsSupport then
        Add value to actions
        Output actions as action rule
    end
end
```

Fig. 4. Action Rules extraction algorithm in a distributed environment

To overcome the problem of incomplete Action Rules generated, the above algorithm takes all possible combination of missing values in Action Rules and fills out the missing values that has greater Support and Confidence. Since Spark handles all combinations in-memory with RDDs, detecting suitable combinations are much faster than to perform the same in Hadoop MapReduce. "Fig. 4" shows the distributed algorithm in detail.

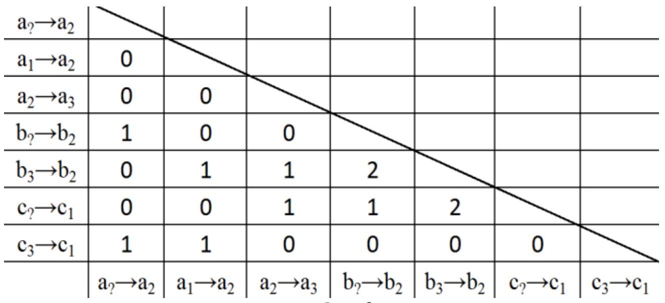
E. Action Set Correlations and Lowest Cost Action Rules

Tzacheva and Ras [16] proposed an approach for discovering Action Rules of lowest cost by taking into account the correlations between individual atomic action terms or sets. An atomic action term is an expression that defines a valid transition of state for a single distinct attribute. For example, $(a, a_1 \rightarrow a_2)$ is an atomic action term, which defines a transition of state for the attribute a from a_1 to a_2, where $a_1, a_2 \in V_a$. Here, the attribute a is a flexible attribute, since it changes its state from a_1 to a_2. An action term 't' consists of all atomic action sets contained in it. Action Rules are composition of *$n -$ pair terms*, means it contains 'n' atomic action terms. For example, a *$2 -$ pair action set* can be represented as "Equation (8)".

$$t = (a, a_1 \rightarrow a_2) \wedge (b, b_1) \qquad (8)$$

It consists of two atomic action terms, namely $(a, a_1 \rightarrow a_2)$ and (b, b_1).

Our method extracts all atomic action sets from the list of Action Rules discovered, and builds a Correlation Matrix as shown in "Fig. 5", which shows the most frequent pairs of atomic action sets within the list of Action Rules, which occurs atleast the user's frequency ϑ times in the entire Action Rules set.



	$a_? \to a_2$	$a_1 \to a_2$	$a_2 \to a_3$	$b_? \to b_2$	$b_3 \to b_2$	$c_? \to c_1$	$c_3 \to c_1$
$a_? \to a_2$							
$a_1 \to a_2$	0						
$a_2 \to a_3$	0	0					
$b_? \to b_2$	1	0	0				
$b_3 \to b_2$	0	1	1	2			
$c_? \to c_1$	0	0	1	1	2		
$c_3 \to c_1$	1	1	0	0	0	0	

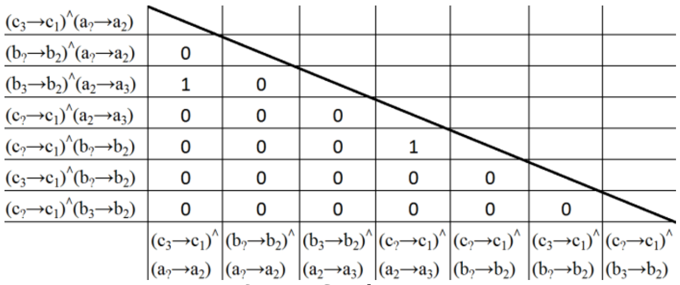
Fig. 5. 1-Pair Correlation Matrix

An atomic action term pair is frequent if it satisfies the *minimum frequency* threshold ϑ of the user. Initially, the algorithm scans the correlation matrix to find all *1-pair action term* combinations that are frequent and marks them. In the next step, the algorithm constructs *2-pair* correlation matrix by combining the marked sets from the *1-pair* correlation matrix. The process continues until no more action set pairs are marked in a correlation matrix.

The approach assumes that if an action set pair is marked frequent, then there exists a correlation between the changes which each individual atomic action set triggers. For example, in an Action Rule r_1 in "Equation (9)".

$$r_1 = [a_2 \wedge c_3] \wedge [(b, b_1 \rightarrow b_3) \wedge (d, d_1 \rightarrow d_2)] \Rightarrow (e, e_1 \rightarrow e_2) \qquad (9)$$

Here, if the 2-pair action set $[(b, b_1 \rightarrow b_3) \wedge (d, d_1 \rightarrow d_2)]$ is frequent, then the changes that occur when $(b, b_1 \rightarrow b_3)$ happen are considered to correlate with the changes that occur when $(d, d_1 \rightarrow d_2)$ happen. "Fig. 6" gives a sample 2-pair correlation matrix.



	$(c_3 \to c_1)^\wedge(a_? \to a_2)$	$(b_? \to b_2)^\wedge(a_? \to a_2)$	$(b_3 \to b_2)^\wedge(a_2 \to a_3)$	$(c_? \to c_1)^\wedge(a_2 \to a_3)$	$(c_? \to c_1)^\wedge(b_? \to b_2)$	$(c_3 \to c_1)^\wedge(b_? \to b_2)$	$(c_? \to c_1)^\wedge(b_3 \to b_2)$
$(c_3 \to c_1)^\wedge(a_? \to a_2)$							
$(b_? \to b_2)^\wedge(a_? \to a_2)$	0						
$(b_3 \to b_2)^\wedge(a_2 \to a_3)$	1	0					
$(c_? \to c_1)^\wedge(a_2 \to a_3)$	0	0	0				
$(c_? \to c_1)^\wedge(b_? \to b_2)$	0	0	0	1			
$(c_3 \to c_1)^\wedge(b_? \to b_2)$	0	0	0	0	0		
$(c_? \to c_1)^\wedge(b_3 \to b_2)$	0	0	0	0	0	0	

Fig. 6. 2-Pair Correlation Matrix

A cost ρ is given to all atomic action terms, which is a number in the range $(0, \omega]$, by the experts working in the problem domain.

$$\rho(b, v_1 \rightarrow v_2) \qquad (10)$$

For example, "Equation (10)" denotes the average cost of changing the attribute b from v_1 to v_2. To lower the cost of an Action Rule, the strategy proposes, if there is a frequent action set pair in the Action Rule, and then the atomic action set of the lowest cost within the pair can only be considered for recommending Action Rule to the user to achieve the desired change. This is because paying the cost to make the changes of one atomic action set to occur, would most probably trigger the changes in the correlated atomic action sets to occur as well.

F. Action Set Correlations in Spark

```
ALGORITHM:
CORRELATION(p,threshold)
Map:
        for each row in the data do
                Find all p- element combinations in row
                for each combination do
                        emit (combination,1)
                end
        end
Reduce:
        Group all combinations by key
        action_frequency = Count of 1's in each key
        if action_frequency < threshold then
                delete (key,action_frequency)
        end

Collect all (key, frequency) pairs
```

Fig. 7. Finding Correlations in a distributed environment using Spark

In this section, we propose a distributed implementation of the Action Set Correlations method by using the Apache Spark framework. The proposed algorithm is shown on "Fig.7".

In this work, we use Apache Spark to find correlation matrices and low-cost Action Rule recommendations for Action Rules extracted from the given datasets using the proposed algorithm from [11]. Since Spark, by its design to work efficiently for iterative algorithms, suits well for finding correlation matrices iteratively from the given set of Action Rules. Fig. 7 shows the algorithm for calculating the frequent action set pairs that meets minimum frequency threshold. The algorithm given in "Fig. 7" starts building *1-item* action terms from the input Action Rules and counts frequency of each single action terms. The algorithm emits the action term only if the frequency of the action term is greater than the user's threshold ϑ. Algorithm runs iteratively, finding *2-items*, *3-items* and so on until the algorithm does not emits atleast one frequent action term pair.

All frequent action set pairs are stored in a separate file. A separate Spark program detect the low cost action term for all frequent action terms from this file and outputs patterns like in "Equation (11)":

$$[t_1 \cup t_2 \cup t_3 \cup \ldots\ldots t_n] \Rightarrow t_m \qquad (11)$$

"Equation (11)" represents that when a pattern of $\boldsymbol{t_1} \cup \boldsymbol{t_2} \cup \boldsymbol{t_3} \cup \ldots\ldots \boldsymbol{t_n}$ occurs in the input Action Rules, we should replace this pattern with $\boldsymbol{t_m}$, which obviously reduces the cost of the Action Rule.

The proposed algorithm also handles a sampling method, similar to stratified sampling, of the data instead of giving control to Spark to do randomly partition the data. The input data file is manually split into '*d*' groups, where *d* is the number of distinct decision attribute values. We then measure how much proportion of data each decision value takes. According to this proportion, we take random samples of data from each group creating '*n*' partitions, where *n* is the number of partitions. In this way, each data partition contains same proportion of decision values, which is equal to the original dataset. Spark reads each partition and create RDDs. The algorithm also executes on each RDDs using Spark's *MapPartition* function. The *MapPartition* function performs computations on each partition of the data, and each partition outputs a set of Action Rules with corresponding support and confidence for each rule. The resulting Action Rules from the *MapPartition* function are sorted by the attribute name and outputted as <*Key, Value*> pairs. Action rules are considered as the *Key* part and support and confidence pair of the Action Rule to be the corresponding *Value* part. The *groupByKey* method is then applied on the RDDs to group all supports and confidences of a unique Action Rule and aggregate them to compute final support 'f_s' and confidence 'f_c' of an Action Rule. The Action Rules which meet minimum threshold levels of support and confidence i.e., $f_s >= minimumSupport$ and $f_c >= minimumConfidence$ are written as output into a text file.

IV. EXPERIMENT AND RESULTS

To evaluate our approach in a distributed environment we used Car Evaluation dataset and Mammographic Mass dataset [10] from the University of California, Irvine's Machine Learning Repository maintained by the Department of Information and Computer Science. We compare the results with the efficiency to generate lowest cost Action Rules using the existing algorithm. As our approach aims in adapting the algorithm that computes the lowest cost of Action Rules to work with bigger datasets, and as the original datasets are relatively small, we replicate the original datasets multiple times to test the performance of our proposed approach in a distributed environment. "Table II" provides all details about both the datasets such as number of instances, replication factor, attributes, and decision attribute values.

The Car Evaluation dataset is about evaluating cars on their acceptability condition based on buying price, maintenance cost, and other characteristics of the cars such as number of doors, number of persons it can carry, luggage boot size and safety measures provided. For the purpose of this study, the attributes {*# Persons*, *Doors*} are considered as stable attributes, and the attributes {*Buying*, *LuggageBoot*, *Safety*, *Maintenance*} are considered as flexible attributes. We choose the attribute *Class* as the decision attribute, which is also a flexible attribute.

TABLE II. PROPERTIES OF CAR EVALUATION AND MAMMOGRAPHIC DATASETS

Property	**Car Evaluation Data**	**Mammographic Mass Data**
# of instances	1728	961
Attributes	7 attributes •Buying •Maintenance •Doors •Persons •Luggage Boot •Safety •Class	6 attributes •BI-RADS •Patient's age •Shape •Margin •Density •Severity
Decision attribute values	Class (unacc, acc, good, vgood)	Severity (0 – benign, 1- malignant)
# of instances / decision value	unacc – 1210 acc – 384 good – 69 vgood - 65	0 – 516 1 – 445
Replication Factor	1024	2048
# of instances after replication	1,769,472	1,968,128
Original data size	52 KB	16 KB
Data size after replication	52 MB	26 MB

The Mammographic mass dataset is intended to predict the severity of breast cancer based on BI-RADS assessment, patient's age, shape, margin and density of the cancer. For the purpose of the evaluation of our algorithm, we consider the attributes {*Shape*, *Age*} as stable attributes and the attributes {*BI-RADS*, *Margin*, *Density*} as flexible attributes. We choose the attribute *Severity* as the decision attribute, which is also a flexible attribute. "Table III" depicts the parameters that we set for the selected datasets such as stable attributes, required decision action and the threshold values of minimum support and confidence.

TABLE III. PARAMETERS USED FOR ACTION RULE DISCOVERY

Parameters	**Car Evaluation Dataset**	**Mammographic Mass Dataset**
Stable attributes	Buying, Persons, Doors	Shape, Age
Required decision action	(Class)unacc → acc	(Severity) 1 → 0
Minimum Support and Confidence	2000, 60%	500, 60%

TABLE IV. SAMPLE OUTPUT RESULTS FROM CAR DATASET

Action Rules
1) AR_1 : (buying, high → med) ^ (lugBoot, big → small) ^ (maint, vhigh → med) ^ (persons = 4) ^ (safety, med → high) => (class, unacc → acc) [Support:5104, Old Confidence: 100%, New Confidence: 100%]
2) AR_2 : (buying, med → low) ^ (lugBoot, big → small) ^ (maint, med → low) ^ (persons = 4) ^ (safety, low → med) => (class, unacc → acc) [Support:5104, Old Confidence: 100%, New Confidence: 100%]
3) AR_3 : (buying, high → med) ^ (lugBoot, small → big) ^ (maint, med → vhigh) ^ (persons = more) ^ (safety, low → high) => (class, unacc → acc) [Support:5104, Old Confidence: 100%, New Confidence: 100%]
4) AR_4 : (buying, low → high) ^ (maint, med → low) ^ (persons = 4) ^ (safety, low → high) => (class, unacc → acc) [Support:15312, Old Confidence: 100%, New Confidence: 100%]
Action Rules Cost
1) ARC_1 : (buying, high → med) ^ (lugBoot, big → small) ^ (maint, vhigh → med) ^ (persons = 4) ^ (safety, med → high) [Cost: 1100]
2) ARC_2 : (buying, med → low) ^ (lugBoot, big → small) ^ (maint, med → low) ^ (persons = 4) ^ (safety, low → med) [Cost: 1500]
3) ARC_3 : (buying, high → med) ^ (lugBoot, small → big) ^ (maint, med → vhigh) ^ (persons = more) ^ (safety, low → high) [Cost: 1400]
4) AR_4 : (buying, low → high) ^ (maint, med → low) ^ (persons = 4) ^ (safety, low → high) [Cost: 1100]
Low Cost Action Rules
1) $LCAR_1$: (lugBoot, big → small) ^ (maint, vhigh → med) ^ (persons = 4) (safety, med → high) [Cost: 800]
2) $LCAR_2$: (lugBoot, big → small) ^ (maint, med → low) ^ (persons = 4) ^ (safety, low → med) [Cost: 1000]
3) $LCAR_{3.1}$: (lugBoot, small → big) ^ (persons = more) ^ (safety, low → high) [Cost: 1000]
4) $LCAR_{3.2}$: (buying, high → med) ^ (maint, med → vhigh) ^ (persons = more) ^ (safety, low → high) [Cost: 800]
5) $LCAR_4$: (persons = 4) ^ (safety, low → high) [Cost: 800]

We use the Research Cluster at the University of North Carolina at Charlotte to test our approach on both the datasets. It is a Hadoop cluster containing 6 nodes. For evaluating the performance of our approach in a distributed data processing environment, we execute our algorithm on the selected datasets to extract low cost Action Rules using Spark framework and we compare the results with the computational time on a single machine. "Table IV" shows a sample of the extracted Action Rules, Action Rules Cost and Lowest Cost Action Rules extracted for Car dataset.

From *Action Rules* section in the "Table IV", it is notable that our distributed Action Rules extraction algorithm produces more specific Action Rules that are capable of delivering complete knowledge to the user. Consider the rules AR_1, ARC_1 and $LCAR_1$ from "Table IV".

AR_1 is the Action Rule that describes when '*SeatingCapacity'* is '4', if '*BuyingCost*' decreases from '*high*' to '*medium*', if '*LuggageBootSize*' decreases from '*big*' to '*small*', if '*MaintenanceCost*' decreases from '*veryhigh*' to '*medium*' and if '*SafetyMeasure*' increases from '*med*' to '*high*', certain cars marked as '*Unacceptable*' can become '*Acceptable*' with support of 5104 and confidence of 100%.

In order to make the suggested changes in AR_1, a Meta-Action [16] has to be defined by an expert in the domain. In this case, the car manufacturer determines the Meta-Action in order to make the specified change occur: '*BuyingCost*' decreases from '*high*' to '*medium*'. For example, what can be done do decrease the overall production cost of this make and model car. Meta-Action is an action about the action. In other words, these are high level actions performed by domain experts, which trigger the changes suggested by the Action Rule [16].

Action Rules in *ARC* section gives Action Rules and their corresponding cost given by the experts. ARC_1 defines the same meaning as AR_1. This Action Rule provides the *Cost* that is 1100 units.

Action Rules in *LCAR* section gives low cost Action Rules recommendation for all Action Rules in *AR* and *ARC* section. For example, the Action Rule $LCAR_1$ means that when '*SeatingCapacity*' is '4', '*LuggageBootSize*' decreases from '*big*' to '*small*', if '*MaintenanceCost*' decreases from '*veryhigh*' to '*medium*' and if '*SafetyMeasure*' increases from '*med*' to '*high*', we can obtain the same results but with reduced cost. These changes would trigger the high cost atomic action term *(buying, high → med)*. The algorithm does not stop with providing single low cost Action Rule recommendation. For example, for Action Rule ARC_3 with cost of 1200 units, there are two low cost Action Rules recommendations $LCAR_{3.1}$ with cost of 1000 units and $LCAR_{3.2}$ with cost of 800 units. From these recommendations, the user or a company can choose most desired Action Rule that fit their needs.

"Table V" shows running time of the algorithm in single machine as well as in distributed data processing environment. It can be inferred from the results that, with the current approach the computational time for both the datasets has drastically improved with Spark environment, where the entire processing has completed within seconds, which otherwise would take several minutes with single machine.

TABLE V. EVALUATION OF LOWEST COST ACTION RULES IN SINGLE MACHINE VS SPARK ENVIRONMENT

Dataset	**# Records**	**# Action Rules**	**Single Machine Algorithm Running time (min)**	**Spark Algorithm Running time (sec)**
Car Dataset	1728	40	1.1	4
	2,204,927	415	>15	6.5
Mammographic Mass Dataset	961	185	0.29 (17 sec)	4.5
	1,968,124	443	1.8	6.25

V. CONCLUSION

Considering the large volumes of patterns discovered by data mining methods, an interestingness measure is essential to filter out the patterns to the most useful ones. Action Rules mining discovers actionable patterns, which are considered interesting. Little research has been done to measure interestingness of association rules or Action Rules. Action Rules of low cost are considered patterns of high interest, and as such are important.

The actions suggested by these special rules can be used for decision making purpose and to achieve the desired goals of the user or an organization. They can be applied in several domains including: medical, financial, industrial, educational, and social networks. However, with the advent of Big Data, the Action Rules algorithm require changes in order to adapt it to distributed environment processing and cloud computing. Current cloud computing frameworks offer a few adaptations for machine learning algorithms, however currently there do not exist any adaptations for Action Rules method or Action Rules lowest cost method.

In this work, we present an adaptation of Action Rules at lowest cost method to distributed processing using Apache Spark. The proposed adaptation improves the method, and acts as a scalable solution for producing low cost of Action Rules for large volumes of data at a reasonable processing time. The proposed approach outperforms the existing method in terms of computational efficiency with the Car Evaluation dataset and Mammographic dataset. In the future, we plan to improve the action set correlations matrix in order to reduce the cost, and perform additional experiments with financial data and social network data.

REFERENCES

[1] Ras, Z. and Wieczorkowska, A. (2000). “Action Rules: how to increase profit of a company, In: Principles of Data Mining and Knowledge Discovery”, (Eds. D.A. Zighed, J. Komorowski, J. Zytkow), Proceedings of PKDD’00, Lyon, France, LNAI, No. 1910, Springer, pp. 587-592.

[2] Matei Zaharia et al. “Spark: Cluster Computing with Working Sets”, HotCloud 2010, pp. 2-5.

[3] He, Z., Xu, X., Deng, S., and Ma, R. (2005). “Mining action rules from scratch”, Expert Systems with Applications, pp. 1-15.

[4] Tzacheva, A. A. (2008). “Diversity of Summaries for Interesting Action Rule Discovery”, In: Proceedings of Intelligent Information Systems (IIS 2008), pp. 7-9.

[5] Tzacheva, A. A. and Ras, Z.W. (2005), “Action rules mining”. International Journal of Intelligent Systems, 20(6):719-736, pp. 1-19.

[6] Ras, Z., Wyrzykowska, E., and Wasyluk, H. (2007). “ARAS: Action Rules discovery based on Agglomerative Strategy”, In: Post-Proceedings of 2007 ECML / PKDD Third International Workshop on Mining Complex Data (MCD 2007), pp. 6-9.

[7] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST ’10). IEEE Computer Society, Washington, DC, USA, pp. 1-7.

[8] Matei Zaharia et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. NSDI 2012, pp. 2-8.

[9] Dean, J. and Ghemawat, S., "MapReduce: Simplified Dataprocessing on large clusters," in Proceedings of the 6th conference on Symposium on Operating Systems Design and Implementation, Berkeley, CA, USA, 2004, pp. 10-10.

[10] Lichman, M., (2013), “UCI Machine Learning Repository” [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

[11] Bagavathi, A., and Tzacheva, A. A., "Rule Based Systems in a Distributed Environment: Survey", in Proceedings of International Conference on Cloud Computing and Applications (CCA17), 3rd World Congress on Electrical Engineering and Computer Systems and Science (EECSS’17), June 4-6, 2017 Rome, Italy, DOI: 10.11159/cca17.107

[12] Tzacheva, A.A, Bagavathi, A, Ganesan, P.D (2016). “MR-Random Forest Algorithm for Distributed Action Rules Discovery”, in International Journal of Data Mining and Knowledge Management Process (IJDKP), Vol. 6, No.5, pp. 15-30

[13] Tzacheva, A.A, Sankar, C.C., Ramachandran, S., Shankar, R.A. (2016), “Support Confidence and Utility of Action Rules Triggered by Meta-Actions”, in proceedings of 2016 IEEE International Conference on Knowledge Engineering and Applications (ICKEA 2016), Singapore.

[14] Tsay, L. and Ras, Z.W., "Discovering E-Action Rules from Incomplete Information Systems," in Proceedings of IEEE International Conference on Granular Computing, Atlanta, Georgia, 2006.

[15] Im, S., Ras, Z.W, Tsay, L.S (2011), “Action Reducts”, in Foundations of Intelligent Systems, Proceedings of ISMIS 2011 Symposium, LNAI, Vol. 6804, Springer, pp. 62-69

[16] Tzacheva, A.A, Ras, Z.W. “Association Action Rules and Action Paths Triggered by Meta-Actions”, in Proceedings of 2010 IEEE International Conference on Granular Computing (GrC 2010), P4161, Silicon Valley, California, USA, 14-16 August 2010, pp. 772-776

[17] Tew, C., C. Giraud-Carrier, K. Tanner, and S. Burton. “Behavior-based clustering and analysis of interestingness measures for association rule mining." Data Mining and Knowledge Discovery 28, no. 4 (2014), pp. 1004-1045