

## A multithreaded hybrid framework for mining frequent itemsets

Jashma Suresh Ponmudiyan Poovan<sup>1</sup>, Dinesh Acharya Udupi<sup>1</sup>,  
Nandanavana Veerappareddy Subba Reddy<sup>2</sup>

<sup>1</sup>Department of Computer Science and Engineering, Manipal Institute of Technology, Manipal Academy of Higher Education, Manipal, India

<sup>2</sup>Department of Information Technology, Manipal Institute of Technology Bengaluru, Manipal Academy of Higher Education, Bengaluru, India

### Article Info

#### Article history:

Received Apr 11, 2021

Revised Oct 13, 2021

Accepted Nov 15, 2021

#### Keywords:

Data structure  
Hybrid  
Multi-threading  
NegNodesets  
N-list

### ABSTRACT

Mining frequent itemsets is an area of data mining that has beguiled several researchers in recent years. Varied data structures such as Nodesets, DiffNodesets, NegNodesets, N-lists, and Diffsets are among a few that were employed to extract frequent items. However, most of these approaches fell short either in respect of run time or memory. Hybrid frameworks were formulated to repress these issues that encompass the deployment of two or more data structures to facilitate effective mining of frequent itemsets. Such an approach aims to exploit the advantages of either of the data structures while mitigating the problems of relying on either of them alone. However, limited efforts have been made to reinforce the efficiency of such frameworks. To address these issues this paper proposes a novel multithreaded hybrid framework comprising of NegNodesets and N-list structure that uses the multicore feature of today's processors. While NegNodesets offer a concise representation of itemsets, N-lists rely on List intersection thereby speeding up the mining process. To optimize the extraction of frequent items a hash-based algorithm has been designed here to extract the resultant set of frequent items which further enhances the novelty of the framework.

*This is an open access article under the [CC BY-SA](https://creativecommons.org/licenses/by-sa/4.0/) license.*



### Corresponding Author:

Dinesh Acharya Udupi

Department of Computer Science Engineering, Manipal Institute of Technology, Manipal Academy of Higher Education

576104, Manipal, Karnataka, India

Email: dinesh.acharya@manipal.edu

## 1. INTRODUCTION

Mining frequent items deal with extricating itemsets that are commonly occurring in a dataset. Several approaches were designed in previous years to achieve this. A variety of data structures such as Nodesets, DiffNodesets, NegNodesets, N-lists, and Diffsets are among a few that have been employed to mine frequent items. These may broadly be classified into tree-based and list-based data structures. While tree traversal approaches have the obvious drawback of consuming more time for construction and traversal. List based algorithms on the other hand consume more memory because of their structure. Despite these conspicuous drawbacks, these approaches have also been found to have some perks. Tree-based algorithms have accelerated performance in generating candidate item sets, while list-based algorithms that rely on list intersections were found to be more effective in extricating the resultant set of frequent items. This meant that there was a dire need to devise a mechanism whereby the advantages of these data structures could be

exploited to the maximum while keeping the effects of their drawbacks minimal. This led to the advent of hybrid frameworks. These employ a combination of two or more data- structures to extract frequent items and were found to be far more efficient than employing a single data structure alone. However limited efforts have been made to enhance the performance of such frameworks.

The main contributions of the paper and its novelty can be explained as follows. This paper has designed a novel hybrid framework comprising of the tree-based structure NegNodesets and the list-based structure N-lists to mine frequent itemsets. As per our investigation, such a framework has so far not been devised for extracting frequent items. To magnify the performance of the extraction process, a novel framework based on multithreading that exploits the multi-core feature of today's processors has been presented here. Existing approaches were found to have problems with load balancing, synchronization, and communication. Since the proposed work partitions the data across several cores there is an effective balance in the load across the processor. Also, the multithreading feature of the proposed work takes care of synchronization and communication between different threads thus avoiding the need to rely on an added mobile agent to assimilate the results from different cores.

The framework begins its execution by partitioning the dataset into as many partitions that are large enough to fit the memory. Each of these partitions is assigned to different cores and all of these cores are run in parallel. While one core may be using NegNodesets for extracting frequent itemsets the other may be employing N-lists for mining the itemsets. This was achieved through multi-threading. Multi-threading utilizes the multi-core feature of current processors by running on more than one core at the same time. In general, it was observed that the computational efficiency increased as the number of cores employed for processing increased. This design was found to balance the load effectively on different cores of the processor thereby enhancing the performance in terms of run time and memory. The output of each of these partitions was consolidated but this collection was found to contain both frequent as well as infrequent itemsets. This predicament has been addressed by finding the summation of the support of itemsets mined from different partitions. This value is then compared against the global support count and only those itemsets whose support surpasses the global support value are added to the resultant collection of "frequent itemsets".

To optimize the mining of frequent items this work has designed a novel hash-based approach for pruning out infrequent itemsets. Only those itemsets that satisfy the minimum support threshold are added to the hash table and the rest are obliterated. In case of collision, the next free slot is found through open addressing. Traditional approaches to pruning typically involve the erection and spanning of a set enumeration tree. This was found to be taxing in terms of run time and memory. The technique proposed in this paper on the other hand relies on employing a hash table to expedite the resultant set of frequent items thereby bringing down the runtime of the overall execution process.

To analyze the performance of the proposed approach, it has been compared with the recent existing approaches such as Negfin, PrePost+, and multicore tree data structure (TDS) Apriori algorithms. Each of these algorithms has been tested on five datasets downloaded from the UCI repository. With the help of detailed experiments, we have shown that the proposed work outperforms existing approaches in terms of run time and memory while extracting the required collection of frequent itemsets.

The background that led to this research is explained as follows. Frequent itemset mining aims to find the set of items that occur frequently in a dataset. A tree data structure improved single scan pattern tree (ISSP-tree) that employed splitting, shuffling, and merging operations was presented by Ahmed and Nath [1] to extract frequent itemsets. A similar tree-based approach that utilized the hierarchy concept of multi-scale theory was put forward by Xun *et al.* [2] for incremental mining of frequent itemsets. This method had the advantage of avoiding repeated scanning of the database. However, no efforts were made to parallelize the algorithm. The tree-based structure has also been used for extracting data from large uncertain databases. Sha and Halim [3] employed a connected tree technique in combination with 3D linked arrays and a tree-based average probability set up for mining frequent items. Experiments indicated a significant improvement in runtime but memory utilization was higher on a few datasets. An approach to mine Top-Rank-K frequent items was put forth by Abdelaal *et al.* [4]. This paper uses a combination of Nodesets, N-list, and pre-order and post-order code tree (PPC-tree) to extract the resultant set of itemsets. To enhance the efficiency of the proposed approach a dynamic minimum support strategy has been deployed. A similar data structure NegNodesets that uses bitmap representation of itemsets was presented by Nader *et al.* [5]. This involved construction and traversal of a bitmap code (BMC) tree which resulted in the generation of a concise set of frequent itemsets. A similar tree-based methodology was presented by Meng and Sha [6] for evaluating the factors influencing the life contentment and solitude of athletes that had retired. A prefix tree was constructed and traversed to generate the frequent items.

Although the efficacy of tree-based approaches has improved in recent years as stated in the above techniques they continued to incur more time for extricating the final set of frequent itemsets. This is where

list based algorithms have an upper hand. These algorithms have the advantage of using list intersection for mining frequent itemsets which reduce the run time of the mining operation. A List based algorithm PrePost+ was presented by Deng and Lv [7] to mine frequent itemsets. This process involved the construction and traversal of an N-list structure. The use of list intersection operation resulted in an efficient generation of frequent itemsets. This structure has also been used by Nguyen *et al.* [8] in combination with pruning algorithms to mine frequent inter transaction patterns. Bui *et al.* [9] proposed an extension of this structure called weighted N-list structure (WN-list) to mine frequent weighted itemsets. The state-of-the-art algorithm (NFWI) was presented for this purpose. This structure was also employed by Vo *et al.* [10] along with tidset and diffsets to mine top rank-k frequent weighted itemsets. This paper also uses threshold raising and early pruning strategies to amplify the efficacy of extracting top rank-k frequent weighted items.

From the literature, it is clear that N-lists effectively reduce the time required for extracting the itemsets, and consequently the same has been employed in this paper. But despite this advantage, they have the drawback of consuming more memory in most cases. Tree-based approaches on the other hand may be more productive for generating candidates but fall short concerning runtime while extracting the final set of frequent itemsets. This necessitates the development of an approach that harnesses the benefits of both these algorithms while mitigating their obvious drawbacks. This is the crux of hybrid approaches.

Zhang *et al.* [11] put forth a hybrid approach that is an amalgamation of “Apriori” and “graph computing” to mine frequent items. By doing so the benefits of both the algorithms are harnessed while keeping their drawbacks to the minimum. Another hybrid framework that uses a combination of “tree-based” and “inverted list” algorithms was presented by Dawar *et al.* [12] to mine high utility itemsets. This framework however has not done enough investigation on exploring the criteria for switching between both the algorithms. A hybrid framework PARASOL comprising of the combination of resource-constrained mining and parameter constrained mining was presented by Yamamoto *et al.* [13]. The former sets a limit to memory consumption while the latter controls the error rate. This proved beneficial in generating a concise collection of closed itemsets. Ali *et al.* [14] proposed an approach that hybridizes the genetic algorithm along with the local search algorithm. The genetic algorithm produces the best individual from the population while the local search algorithm generates the best local solution by enhancing all neighbor solutions. Although hybrid approaches are generally faster limited efforts have been put forth to test for their scalability. Besides minimal initiatives were taken to boost the efficacy of the pruning techniques employed here.

Trimming the search space can also significantly enhance the productiveness of mining frequent itemsets. Several pruning methods were developed to achieve this. A novel pruning algorithm LengthSort was conferred by Lessanibahri *et al.* [15] to extract frequent Items that satisfy a given length threshold. An alternative approach that comprises removing irrelevant features and rows from datasets of higher dimensions was put forward by Vanahalli and Patil [16]. This method switches between the enumeration of rows and features based on the characteristics of the data during the mining process.

Presently, there has been a surge in the lookout for designing techniques for optimizing the algorithms developed for mining frequent items. Several hash-based algorithms were put forth to optimize their extraction process. Bustio *et al.* [17] put forth a method for mining frequent itemsets from data streams by using a hash-based approach coupled with the lexicographic ordering of itemsets. Besides a partition-based approach as put forward by Srinivasan and Reddy [18]. This divides the dataset into equal partitions and hashes the candidates into a hash table. The primary advantage of the hash-based approach is that it required only linear time even in the worst case. A similar approach was also employed for mining closed itemsets using a hash tree [19]. This process uses a combination of hashing, candidate generation, superset checking, and subset checking.

Alternatively parallel and concurrent approaches were found to amplify the efficacy of mining itemsets. However, while adopting such strategies, deciding on an effective partition-based approach plays a pivotal role. Xun *et al.* [20] proposed a technique that partitions the database based on a similarity metric and locality sensitive hashing technique. The frequent itemsets are then mined from each of the partitions concurrently. Kadappa and Nagesh [21] proposed an approach that involves dividing the dataset into as many partitions that are large enough to fit into the memory. For each partition, all those itemsets that satisfy the local support count are extracted. A second pass is made and compared with the global support count to generate the resultant set of frequent items. A selective partitioning strategy was postulated by Bai *et al.* [22] to extract maximal frequent-itemsets. This approach makes use of a compact data structure called itemset tree and does selective partitioning of the database which reduces the number of scans of the database. A similar approach was designed by Niu *et al.* [23] that constructed an array prefix tree for the parallel extraction of frequent itemsets. To facilitate parallelization, a Spark workflow model was designed.

While designing parallel approaches, recent years have witnessed the deployment of distributed systems for extracting frequent itemsets. A spark-based algorithm was implemented by Rathee and Kashyap [24] for mining frequent patterns from large datasets. This algorithm designs the plan of execution at every iteration and computes its cost. The goal is to select the plan with the minimum cost and this magnifies the

efficiency of this algorithm. A Hadoop-based framework was put forth by Wang and Chang [25] to mine frequent patterns with multiple items supports. Apart from including a support counting phase and a mining phase this algorithm also incorporates the concept of classification of items that categorizes items that have higher homogeneity into the same class. A similar concept was put forward by Hanirex and Kumaravel [26] for mining association rules from a cloud environment. These rules are generated based on the frequent itemsets that are mined from the datasets. To optimize the performance of large-scale mining of frequent itemsets Prabhu *et al.* [27] proposed an approach that uses a Hadoop map reduce framework. This was found to reduce the runtime and disk utilization space thereby enhancing performance. Yimin *et al.* [28] proposed a parallel algorithm based on map-reduce to solve the issues related to time and space complexity during the processing and computing of itemsets. This involved using the DiffNodeset data structure in collaboration with the Hadoop framework to extract the itemsets. A similar approach was put forward by Sornalakshmi *et al.* [29] by a parallel and distributed map-reduce-based Apriori algorithm. The data is divided into clusters and extraction of data is carried out in a distributed manner.

However, despite these advantages, it is observed that distributed systems are usually difficult to deploy, debug and maintain. They are also complex to design in terms of the hardware and software required for communication and security. Moreover, it also suffers from the drawback of increased processing overhead due to the additional computation and exchange of information between the different computers in the system. These drawbacks can be addressed by harnessing the multicore feature of today's processors.

Khawaja *et al.* [30] put forth a TDS for extracting frequent itemsets using the multicore feature of contemporary processors. A divide and conquer approach were adopted here in combination with the Apriori algorithm for extracting the itemsets. A parallel procedure using Dynamic Bit vectors was presented by Huynh *et al.* [31] to extract Frequent Itemsets. In this approach, the itemsets are sorted in ascending order of support count and divided into partitions. Each of these partitions is assigned as a separate task to a different processor. This ensures there is a proper balance of workload between different processors. A related approach using multicores was put forth by AbdulRazaq *et al.* [32] for designing an efficient string matching algorithm. A maximum shift algorithm has been designed for this purpose. Experiments proved that the proposed approach improved the speedup and efficiency of the algorithm. Huynh and Vo [33] employed a similar technique to mine erasable itemsets. A search tree is constructed and each bough is considered to be a separate task. Each of these tasks is run in parallel on a separate core to enable parallel extraction of frequent items. Some of the findings of the literature are summarized in Table 1.

The main emphasis of the proposed approach is to design a framework that would address the drawbacks shown in Table 1. This forms the problem statement for this research. From the literature, we can conclude that although diverse techniques were incurred in the past to mine frequent items they can broadly be divided into tree-based and list-based approaches. These rely on the erection and spanning of a set enumeration tree to prune out infrequent items. This was found to be costly due to the time and space involved in its construction and traversal. This necessitates the development of a mechanism that mitigates these drawbacks. The dependency on a single data structure to mine frequent items usually results in a loss of performance as the mining proceeds. It has been observed that most tree-based structures usually fall short in terms of run time while list-based approaches fall short in terms of memory. This entails the development of a hybrid framework that encompasses two or more data structures. However, limited initiatives were put forth to magnify their performance. Although most contemporary processors are multi-core minimal efforts were made to exploit this feature to its full potential to implement parallelization. While extracting itemsets in parallel deciding on an effective partitioning mechanism is of paramount importance. Most traditional approaches rely on an external mobile agent to assimilate the results of each partition. This can be costly and appropriate measures need to be taken to devise an alternate strategy for consolidation of the results while limiting the overall costs of the proposed approach.

The work proposed in this paper aims to design a framework that would handle the above issues. A novel hybrid framework comprising of the tree-based data structure NegNodesets and the list-based data structure N-lists has been designed here. While NegNodesets offer a concise representation of itemsets, N-lists were found to be faster since it relies on list intersection operation to extract the frequent items. A novel pruning mechanism based on hashing has been put forth here. This curbs the disadvantages of the regular approach that relies on the erection and spanning of a set enumeration tree. To enhance the efficacy of mining, the framework has been designed to exploit the multicore feature of today's processors which also adds to the novelty of the design. Effective load balancing has been facilitated by dividing the dataset into partitions and assigning each partition to a separate core. The itemsets are then mined in parallel from each core. The results of each partition are consolidated using the concepts of multithreading, thus mitigating the need for employing an external mobile agent thereby improving the overall cost of the proposed framework.

**2. PROOF OF CORRECTNESS OF THE PROPOSED APPROACH**

- Let the given dataset be D,
- Divide D into partitions  $D_1, D_2 \dots D_N$  where N is the no of cores in processors
- On each  $D_i$  assign a thread to extract frequent items  $F_i$  such that:
  - a.  $F_1=NNH (D_1, \delta)$ , ----- (1)//Invoke NNH algorithm.
  - b.  $F_2=NLH (D_2, \delta)$ , ----- (2)//Invoke NLH algorithm.
  - c.  $F_3=NNH (D_3, \delta) \dots F_N=NLH (D_N, \delta)$ .

**Table 1. Comparison of different frequent itemset mining algorithms**

Algorithm	Methodology	Drawbacks
ISSP-tree [1]	<ul style="list-style-type: none"> <li>- Employs a tree data structure</li> <li>- Performs splitting, shuffling, and merging operations</li> <li>- Requires single scan, improved runtime</li> </ul>	Heavily dependent on main memory
PFI [3]	Employed a connected tree technique in combination with 3D linked arrays and a tree-based average probability set up for mining frequent items	Memory utilization was higher on a few datasets
Negfin [5]	<ul style="list-style-type: none"> <li>- Uses NegNodeset that relies on a bitmap representation of itemsets</li> <li>- Constructs and traverses a BMC-tree</li> </ul>	Uses a “set enumeration tree” for generating the frequent itemsets which add to the execution time and memory used
PrePost+ [7]	<ul style="list-style-type: none"> <li>- Employs N-list data structure to mine frequent itemsets</li> <li>- Utilizes Children–Parent Equivalence pruning strategy to reduce the search space</li> </ul>	The proposed pruning approach relies on the construction and traversal of a set enumeration tree that adds to the memory consumption
NL-ITP [8]	<ul style="list-style-type: none"> <li>- Uses N-list data structure to extract itemsets</li> <li>- Reduces the search space significantly to generate FITPs</li> </ul>	Shows limited improvement in runtime on sparse datasets
TFWIN+ [10]	<ul style="list-style-type: none"> <li>- Combining mining and ranking phases into one</li> <li>- Uses Tidset, Diffset, and WN-list structures to extract the required itemsets.</li> <li>- Proposes threshold raising strategy and early pruning to effectively extract top rank-k-Frequent Weighted items</li> </ul>	No initiatives were taken to parallelize the approach
ANG [11]	A hybrid method that uses a combination of apriori and graph computing for mining frequent itemsets has been designed	Has not been scaled up to exploit the multi-core feature of current processors.
PARASOL [13]	<ul style="list-style-type: none"> <li>- Uses a combination of parameter constrained and resource-constrained mining techniques</li> <li>- Designed to mine frequent items from a streaming environment</li> </ul>	Scalability has been tested only in terms of transaction length and not in terms of parallelization.
PSS-FIM [18]	Divides the dataset into equal partitions and hashes the candidates into a hash table	Limited efforts made to utilize the multicore feature of current processors
LSPA [21]	Proposed a partition-based approach that compares the local support against the global support to generate the resultant set of frequent items	May classify some frequent items as infrequent and vice-versa
PEMA [22]	<ul style="list-style-type: none"> <li>- Uses a combination of horizontal and vertical partitioning techniques</li> <li>- Mobile ARM agents incrementally integrate the locally mined frequent itemsets to produce the global set of frequent items</li> </ul>	<ul style="list-style-type: none"> <li>- Relies on mobile agents for assimilating the results adding to the overall cost of the algorithm</li> <li>- Novel approaches such as multi-threading have not been employed here</li> </ul>
Adaptive-Miner [24]	Makes execution plans before every iteration and selects the plan that minimizes time and space complexity	Complex to design in terms of hardware and software required for communication
PFIMD [28]	<ul style="list-style-type: none"> <li>- Uses DiffNodeset to increase the cardinality of N-list.</li> <li>- A 2-way comparison strategy is designed to reduce the time complexity of the algorithm</li> </ul>	Limited improvement in memory since DiffNodesets is not a very compact data structure.
MA-TDS [30]	<ul style="list-style-type: none"> <li>- Employs a tree data structure for mining frequent itemsets</li> <li>- Uses the multi-core feature of current processors</li> </ul>	Relies on the apriori technique for pruning which is costly in terms of time
pDBV-SPM [31]	<ul style="list-style-type: none"> <li>- Uses dynamic bit vector for finding support of itemsets</li> <li>- Uses the multi-core architecture of today’s processors for parallel mining of sequential itemsets</li> </ul>	Limited efforts made to increase the efficiency of the pruning of infrequent items

(1)➔Scan the partition  $D_1$  and perform the following steps to extract frequent itemsets from  $D_1$ .

- Build the BMC\_Tree B. An itemset in this tree may be defined as:  
 B (itemset  $I_h$ ): each itemset  $I_h$  is designated using a bitmap code  $B(I_h) = a_{n-1} \dots a_1 a_0$  of size n in the following way: the  $p^{th}$  item in  $I_1$  is  $a_p$ . If an item  $i$  ( $i \in I_1$ ) constitutes  $I_h$ , then the bit representing it is set to 1; otherwise, it is 0. [5].
- Traverse B to generate candidates of frequent-1 itemsets  $C_1$  which may be defined as:  
 $C_1 = \{ \text{Nodeset}(I_h), \text{Ndata of } X \mid X \text{ contains } i_1, \text{ and } \forall i_m, 1 \leq m \leq h, \text{ the bit set to } i_h \text{ in } N. \text{ bitmap\_Code is } 1 \}$  [5].

- FIT= $\emptyset$ ; Htable= $\emptyset$ ; // The resultant collection of Frequent Itemsets FIT and the Hash table Htable are initialized to  $\emptyset$ .
- Find FIT=FIT  $\cup$  C<sub>1</sub>// Add C<sub>1</sub> to FIT.
- Find higher order candidates C<sub>2</sub> = {NegNodeset (I<sub>h</sub> = i<sub>h</sub>i<sub>h-1</sub> ... i<sub>2</sub>i<sub>1</sub>)=Nodeset(I'<sub>h</sub> =  $\neg$  i<sub>h</sub>i<sub>h-1</sub> ... i<sub>2</sub>i<sub>1</sub>)} [5].
- Update FIT=FIT  $\cup$  C<sub>2</sub>// Add C<sub>2</sub> to FIT.
- $\forall x \in$  FIT, if support(x)  $\geq \delta$ , Htable [x]=x mod t, where Htable [x] denotes the next free slot in the hash table and t denotes the total number of itemsets.
- F<sub>1</sub> <- Htable =====> (3) // F<sub>1</sub> contains the set of frequent itemsets of Partition 1.
- (2)>Scan the partition D<sub>2</sub> and perform the following steps to extract frequent itemsets from D<sub>2</sub>.
- Build the PPC\_Tree P. Each itemset in P may be defined as:  
P (itemset J<sub>k</sub>): each node N in PPC\_Tree is represented as ((N\_precode: N\_postcode): count). [7].
- FIT= $\emptyset$ ; Htable= $\emptyset$ ; // The resultant collection of Frequent Itemsets FIT and the Hash table Htable are initialized to  $\emptyset$ .
- Find collection of Frequent-1 itemsets FIT<sub>1</sub>={FI | FI is a series of “PP-codes” of nodes corresponding to each item in the “PPC\_Tree”} [7].
- Find FIT= FIT  $\cup$  FIT<sub>1</sub>// Add FIT<sub>1</sub> to FIT.
- Find higher-order candidates FIT<sub>2</sub>={a<sub>1</sub>a<sub>2</sub> | a<sub>1</sub>a<sub>2</sub> is a series of “PP-codes” in increasing order and produced by the intersection of the N-lists of a<sub>1</sub> and a<sub>2</sub>} [7] i.e.,  $\forall i_a i_b (\in$  FIT<sub>2</sub>), N\_list=i<sub>a</sub>.N\_list  $\cap$  i<sub>b</sub>.N\_list.
- $\forall (x) \in$  N\_list if support(x)  $\geq \delta$ , Htable [x] =x mod t, where Htable [x] denotes the next free slot in the hash table and t denotes the total number of itemsets.
- F<sub>2</sub> <- Htable =====> (4) // F<sub>2</sub> contains the set of frequent itemsets of Partition 2
- From (3) and (4)> the collection of frequent itemsets F<sub>1</sub> and F<sub>2</sub> are mined from partitions D<sub>1</sub> and D<sub>2</sub> using NegNodeHash (NNH) and N-list hash (NLH) algorithms respectively. A similar approach is followed on other partitions as well.
- Assimilating the results of each partition we get F={F | F<sub>1</sub>  $\cup$  F<sub>2</sub>  $\cup$  F<sub>3</sub>...  $\cup$  F<sub>N</sub>}
- $\forall f_i \in$  F, find Sum(f) |Sum(f) =  $\sum_{i=1}^N (S_i(f))$  // Finds the summation of the supports of each itemset in each partition
- If Sum(f)  $\geq \theta$  then G={G  $\cup$  F, where G is a Global frequent Itemset and  $\theta$  is the Global support count}

### 3. RESEARCH METHOD

In this section, a detailed outline of the proposed methodology is explained. A hybrid framework consisting of NegNodesets and N-lists was designed for mining frequent itemsets. The basic procedure is outlined in multithreaded hybrid framework (MHF) algorithm as shown in Algorithm 1. This in turn invokes NNH and NLH algorithms.

The algorithm MHF begins its execution by taking the given dataset as input and preprocessing the dataset. This involves sorting the itemsets in descending order of their support to eliminate infrequent items. The quicksort algorithm is used for this purpose. All transactions that have the same set of items are merged [Lines 1 and 2 of Algorithm 1]. This will help in reducing the size of the dataset. A divide and conquer strategy is employed and the dataset is then divided into as many partitions that are large enough to fit into the memory. Both NNH and NLH algorithms are run in parallel on the partitions [Lines 3 to 5 of Algorithm 1]. A multicore-based framework that employs multi-threading has been designed for this process. Since a multi-core architecture has two or more cores it allows several tasks to be run in parallel enhancing the performance of the mining activity.

Each partition is run on a separate core to ensure the parallel extraction of frequent itemsets. Only those itemsets whose support is greater than the local support is added to the final collection of frequent itemsets. The output of each partition is combined into a resultant file. The final result at this stage will contain both frequent as well as infrequent itemsets. To address this issue, the support of the itemsets produced from each of the partitions is summed up and compared against the global support count. If the total support surpasses the global support, then the item is considered to be frequent and added to the final result [Lines 6 to 11 of Algorithm 1]. This way the final set will consist of only frequent itemsets. The general methodology describing the above process is illustrated in Figure 1.

The NNH as shown in Algorithm 2 involves the building and traversal of a BMC\_Tree data structure that has its basis in bitmap indexing. The root node of this data structure is set to a null value. Every other node possesses an item set. Each node consists of four fields- item name, support count, binary

representation, and child list. The first step in the construction of this tree involves the identification of frequent 1 itemsets and their Nodesets. This was then used to generate higher-order itemsets called NegNodesets. This process continues till all the frequent itemsets have been obtained [Lines 3 to 8 of Algorithm 2]. Once the candidates are obtained only those items that satisfy the minimum support count are mapped to the hash table. The rest are pruned for further consideration. In case of collision, it is handled through open addressing [Lines 9 to 22 of Algorithm 2].

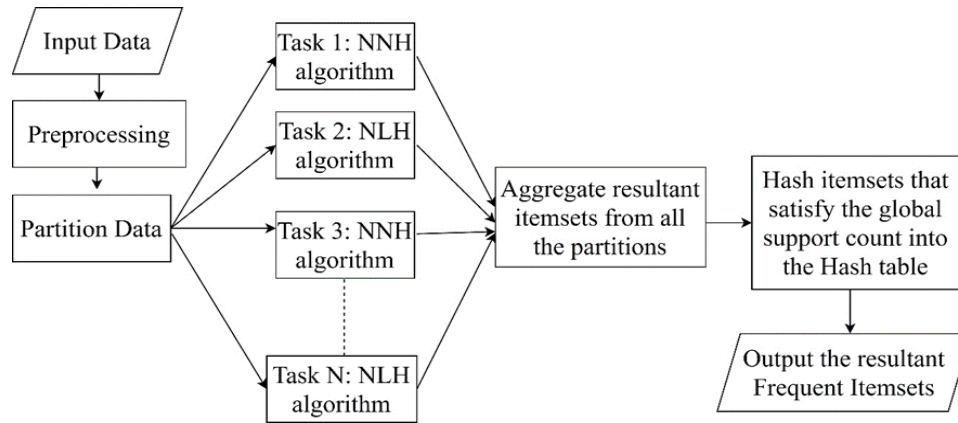


Figure 1. General methodology

NLH algorithm as shown in Algorithm 3 involves the construction and traversal of a “PPC tree” to produce an “N-list” of the itemsets. N-list intersection operation is carried out to generate higher-order item sets [Lines 3 and 4 of Algorithm 3]. The root node is set to null. Each node contains—item name for indicating the item, count- for representing the number of times an item is present, child list—that will contain all the offspring of that particular node, “preorder” and “post-order” for indicating the pre-order and post-order values respectively.

After the construction of the tree, the set of “frequent 1-itemsets” are characterized and these represent the N-lists of frequent 1 itemsets. The same process is carried out for finding frequent 2 itemsets also. This process then repeats itself till all the itemsets are found [Lines 5 to 9 of Algorithm 3]. Once the candidates are obtained only those items that satisfy the minimum support count are mapped to the hash table. The rest are pruned for further consideration. In case of collision, it is handled through open addressing [Lines 10 to 23 of Algorithm 3]. Most conventional algorithms construct and traverse a set enumeration tree to produce the final set of frequent itemsets. This adds to the run time and memory requirements. Also, a novel pruning approach is needed in most cases to avoid unnecessary and redundant visiting of nodes. All these drawbacks have been overcome in this work by adopting a Hash-based strategy to generate the final collection of frequent itemsets.

#### Algorithm 1. MHF ( $D, \delta, \Theta$ )

Input: Transaction database  $D$ , local support  $\delta$ , global support  $\Theta$

Output: Set of frequent itemsets  $F$

1. Sort the itemsets in descending order of support count and prune out infrequent itemsets
2. Perform transaction merging wherever possible
3. Divide the database into partitions  $D \leftarrow D_1, D_2, \dots, D_k$  with  $k \geq 2$  such that each  $D_i$  fits into memory
4. for each  $D_i$  do
5. Allocate each partition to a separate thread such that  $F_1 \leftarrow \text{NNH}(D_1, \delta), F_2 \leftarrow \text{NLH}(D_2, \delta), F_3 \leftarrow \text{NNH}(D_3, \delta), \dots, F_k \leftarrow \text{NNH}(D_k, \delta)$
6. Merge the results into  $F$  such that  $F \leftarrow F_1 \cup F_2 \cup \dots \cup F_k$
7. For each local frequent itemset  $f \in F$  do
8.  $\text{Sum}(f) = \sum_{i=1}^k (S_i(f))$ ;
9. if  $\text{Sum}(f) \geq \Theta$  then  $G = G \cup f$  where  $G$  is a global frequent itemset
10. else
11. Discard  $f$ ;
12. end if
13. end for

**Algorithm 2. NNH ( $D_i, \delta$ )**Input: Transaction database  $D_i$ , minimum support threshold  $\delta$ .

Output: The set of all frequent itemsets, FIT.

```

1. FIT= $\emptyset$ ;
2. Htable =  $\emptyset$ ;
3. Traverse database D and build the BMC tree to find  $C_1$ 
4. FIT = FIT  $\cup$   $C_1$ ;
5. for each node X in the BMC_Tree do
6. Add N data of X into the Nodeset of item X.Iname to generate  $C_2$ 
7. FIT = FIT  $\cup$   $C_2$ ;
8. end for
9. for each item  $x \in$  FIT do
10. if support( $x$ )  $\geq$   $\delta$  then
11. find=0;
12. do
13. hash=hash( $x$ , find);
14. if (Htable[hash] == null) then
15. Htable[hash] =  $x$ ;
16. return;
17. end if
18. find++
19. while (find < SIZE);
20. end if
21. end for
22. FIT  $\leftarrow$  Htable;
23. return FIT;

```

**Algorithm 3. NLH ( $D_i, \delta$ )**Input: Transaction database  $D_i$ , Minimum Support threshold  $\delta$ .

Output: The set of all frequent itemsets, FIT.

```

1. FIT= $\emptyset$ ;
2. Htable =  $\emptyset$ ;
3. Traverse database D and construct the PPC_Tree;
4. Traverse the PPC_Tree to obtain the N_lists of frequent 1-itemsets FIT $_1$ ;
5. Generate the set of frequent 2- itemsets FIT $_2$ ;
6. for each  $i_a i_b \in$  FIT $_2$  do
7. Create N_list by performing NL_intersection ( $i_a$ .N_list,  $i_b$ .N_list);
8. end for
9. FIT  $\leftarrow$  FIT $_1$ ;
10. for each itemset  $x \in$  N_list do
11. if support( $x$ )  $\geq$   $\delta$  then
12. find=0;
13. do
14. hash=hash( $x$ , find);
15. if (Htable[hash] == null) then
16. Htable[hash] =  $x$ ;
17. return;
18. end if
19. find++
20. while (find < SIZE);
21. end if
22. end for
23. FIT  $\leftarrow$  Htable;
24. return FIT

```

**3.1. Illustration with example**

The proposed methodology can be explained with the help of an example. For the dataset shown in Table 2, both NNH and NLH algorithms are run in parallel on each partition. Assume the dataset is divided into  $N$  partitions and assigned to separate threads. Each thread runs on a separate core. Partition 1 will then be running the NNH algorithm while the second partition will be running the NLH algorithm, partition 3 will be running the NNH algorithm, and so forth. The NNH algorithm begins by scanning the dataset and constructing the BMC\_Tree as shown in Figure 2(a). These are then used to find Nodesets as shown in Figure 2(b). These in turn serve as a blueprint for generating higher-order itemsets called NegNodesets. This is illustrated in Figure 2(c). The NegNodesets whose support is greater than the local support are inserted into the hash table and the rest are pruned from consideration.

Partition 2 on the other hand will be running the NLH algorithm. The dataset is scanned and a PPC tree is constructed as shown in Figure 3. This tree is traversed to generate the N-list of Frequent-1 itemsets. An intersection operation is performed on the N-list of Frequent-1 itemsets to generate higher-order itemsets.



The generation of frequent itemsets using N-lists is shown in Figure 4. All those itemsets produced here that satisfy the local support are hashed into the hash table. Finally, the results of each partition are consolidated and compared against the global support count to generate the final collection of frequent itemsets.

Table 2. Dataset description

TID	Items	Ordered FI
1.	5, 2, 7, 4	2, 4, 5
2.	3, 5, 2, 1	1, 2, 3, 5
3.	3, 2, 1, 9	1, 2, 3
4.	1, 4, 8	1, 4
5.	1, 4, 3, 2, 6	1, 2, 3, 4

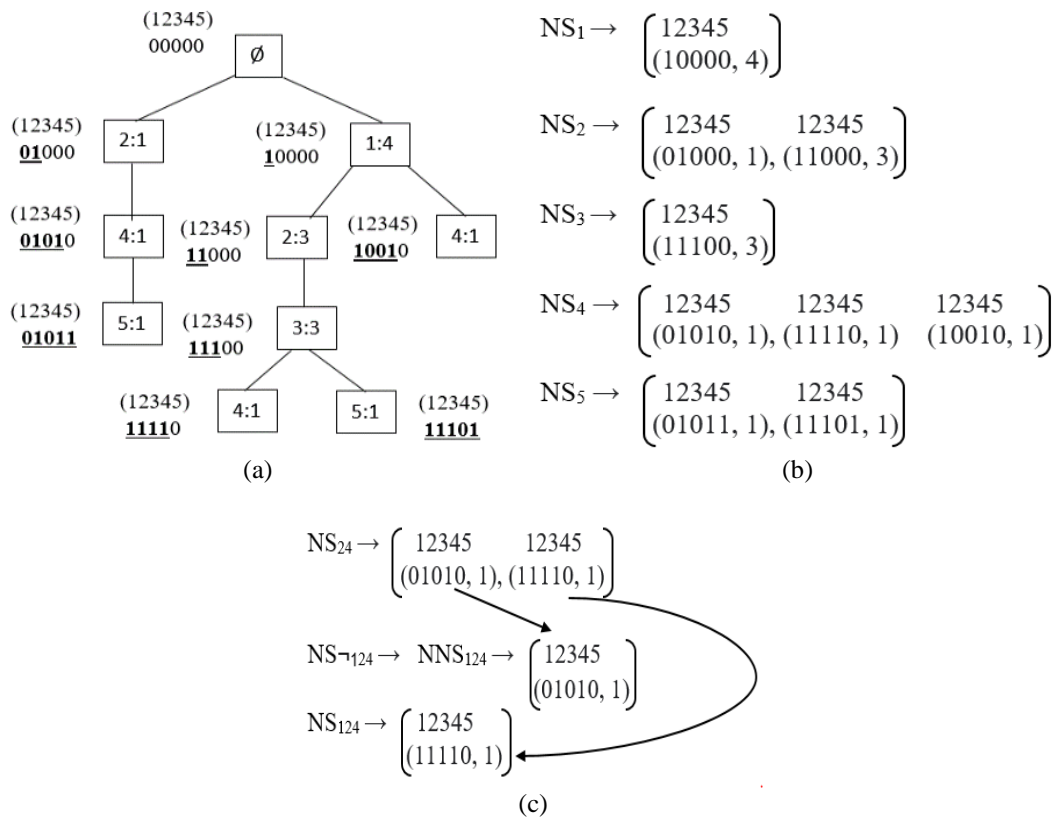


Figure 2. NLH algorithm: (a) BMC tree, (b) nodesets, and (c) Nodeset and NegNodesets of itemset 124

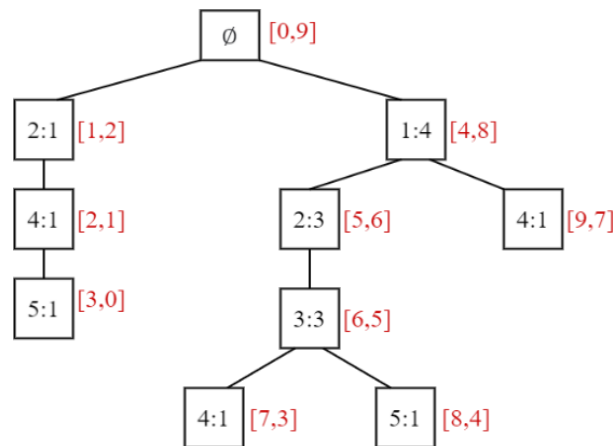


Figure 3. PPC tree

N-list for each items are  
 NL<sub>1</sub>-> <(4, 8):4>  
 NL<sub>2</sub>-> <(1, 2):1, (5, 6):3>  
 NL<sub>3</sub>-> <(6, 5):3>  
 NL<sub>4</sub>-> <(2, 1):1, (7, 3):1, (9, 7):1>  
 NL<sub>5</sub>-> <(3, 0):1, (8, 4):1>

N list for 2 itemsets are as follows:  
 NL<sub>14</sub>-> <(4, 8):1, (4, 8):1>  
 After merge we get <(4, 8):2>  
 NL<sub>23</sub>-> <(5, 6):3>  
 :  
 :  
 Continuing this way we find other itemsets as well

Figure 4. N-Lists generated by NLH algorithm

### 3.2. Advantages of the proposed approach

The proposed hybrid framework takes advantage of the perks of both NegNodesets as well as N-lists. While NegNodesets ensure a concise representation of itemsets, N-lists were found to be faster to extract the resultant set of frequent items. The algorithms proposed in this paper rely on a novel Hash-based pruning to prune out infrequent itemsets. Only those itemsets that satisfy the minimum support threshold are added to the hash table and the rest are pruned from consideration. In case of collision, it was handled through open addressing. This reduced the computational complexity to  $O(\log g)$  thereby enhancing the efficiency of the mining process.

## 4. RESULTS AND DISCUSSION

All experiments were performed on an Intel Pentium i5 core, having a 2.5 GHz processor with 4.0 GB RAM and 64-bit Windows OS. The algorithms presented in this paper were coded in Java and compiled using Eclipse IDE. The proposed approach has been tested on five sets of datasets namely, Skin [34], PowerC [35], PAMAP [36], kddcup99 [37], and OnlineRetail [38] datasets, the details of which are shown in Table 3. All of these datasets have been downloaded from the UCI repository.

The proposed approach has been compared with the recent existing tree-based algorithm Negfin [5] and the list-based algorithm PrePost+ [7]. Besides, it has also been compared with the existing multicore TDS Apriori algorithm [30] which is also a recent algorithm that uses a tree data structure for mining frequent items on a multi-core processor. Each of these algorithms was tested on five datasets that were downloaded from the UCI repository.

Table 3. Dataset description

Dataset Name	Transaction Count	Item Count	Average Item Count Per Transaction
OnlineRetail	541,909	2,603	4.37
Kddcup99	1,000,000	135	16
PAMAP	1,000,000	141	23.93
Skin	245,057	11	4.0
PowerC	1,040,000	140	7

Experiments conducted indicate that the proposed approach outperforms each of these existing algorithms in terms of runtime and memory. The improvement in runtime on Skin, PowerC, OnlineRetail, PAMAP and kddcup99 datasets are illustrated in Figures 5(a) to 5(e) respectively. Likewise the improvement in memory on Skin, PowerC, OnlineRetail, kddcup99 and PAMAP datasets are illustrated in Figures 6(a) to 6(e) respectively. The percentage improvement of the proposed MHF algorithm with regards to the existing algorithms is shown in Tables 4 and 5 respectively. It can be concluded that the proposed MHF algorithm has an aggregate improvement in run time by 48.26%, 39.48%, and 21.57% in comparison to Negfin, PrePost+, and multicore TDS Apriori (MA-TDS) algorithms respectively. With regards to memory, it has been observed that the proposed MHF algorithm has an aggregate improvement of 49.7%, 36.25%, and 20.7% in comparison to Negfin, PrePost+, and MA-TDS algorithms respectively.

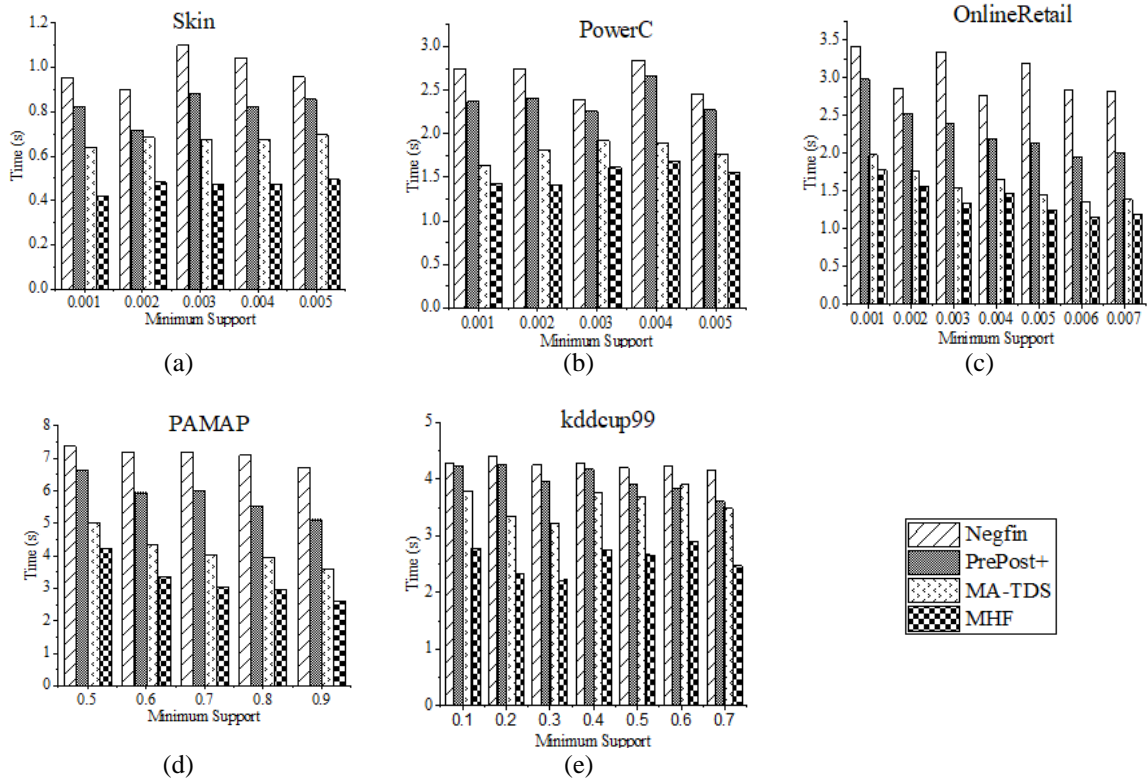


Figure 5. Comparison in the execution time of MHF with Negfin, PrePost+ and MA-TDS algorithm on (a) skin, (b) PowerC, (c) OnlineRetail, (d) PAMAP, and (e) kddcup99 for generating frequent itemsets

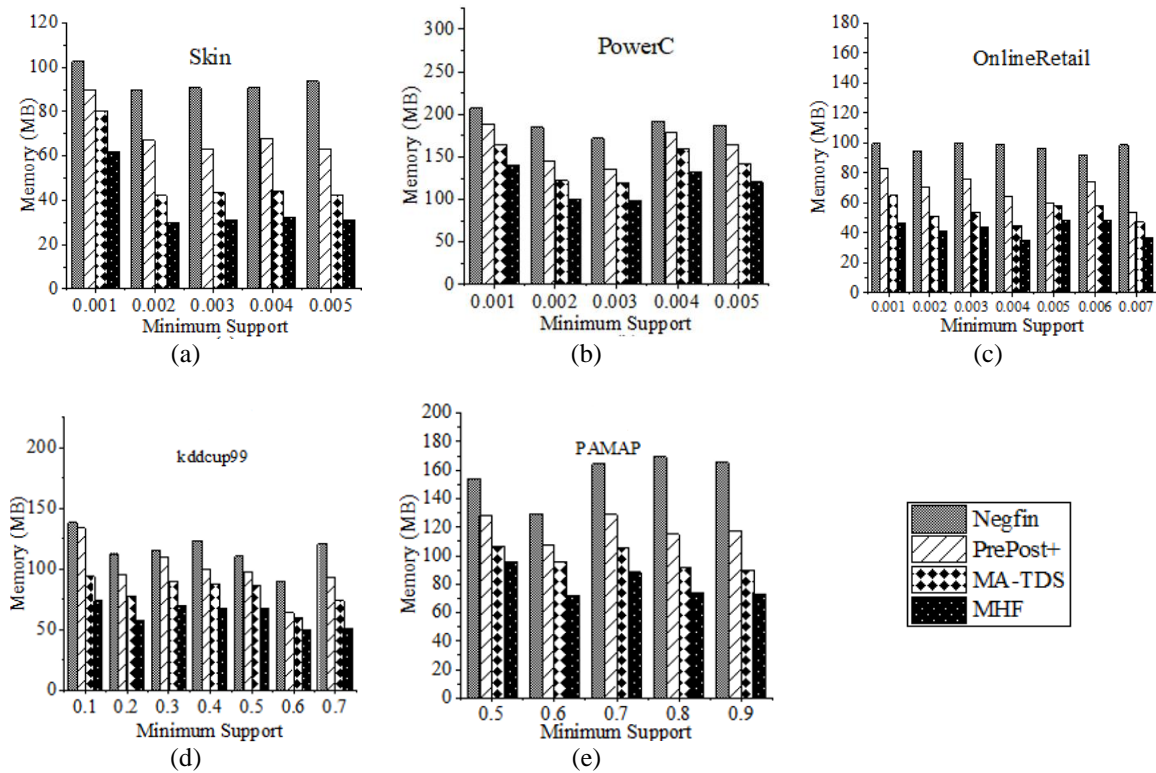


Figure 6. Comparison in the memory consumption of MHF with Negfin, PrePost+ and MA-TDS algorithm on (a) skin, (b) PowerC, (c) OnlineRetail, (d) kddcup99, and (e) PAMAP datasets for generating frequent itemsets

Table 4. Comparison of runtime on 5 datasets

Dataset	Percentage Improvement in Runtime		
	MHF vs. Negfin	MHF vs. PrePost+	MHF vs. MA-TDS
Kddcup99	38.97280967	34.7826087	27.79984114
Online Retail	54.02667169	39.73685836	12.50224215
PAMAP	54.41375042	44.68150474	22.89208222
PowerC	41.40074282	35.44050104	14.39486214
Skin	52.50707643	42.80496713	30.2760463

Table 5. Comparison of memory usage on 5 datasets

Dataset	Percentage Improvement in Memory Consumption		
	MHF vs. Negfin	MHF vs. PrePost+	MHF vs. MA-TDS
Kddcup99	46.06879607	36.74351585	22.98245614
Online Retail	56.15835777	37.96680498	20.8994709
PAMAP	48.52752881	32.55033557	17.62295082
PowerC	37.42071882	27.00369914	16.14730878
Skin	60.34115139	47.00854701	25.89641434

#### 4.1. Performance on multicore systems

To analyze the efficacy of multicore systems the proposed approach has been executed with the various number of cores on PAMAP, PowerC, Skin, Kddcup99, and OnlineRetail datasets as shown in Figures 7(a) to 7(e) respectively. It was observed that the speedup increased by about 2 times with 2 cores and nearly 4 times with 4 cores. The average speedup rate is 1.89, 1.98, 1.91, 1.93, and 2.02 on 2 cores and 3.83, 3.97, 3.92, 4.02, and 3.98 on 4 cores with Kddcup99, OnlineRetail, PAMAP, PowerC, and Skin datasets respectively

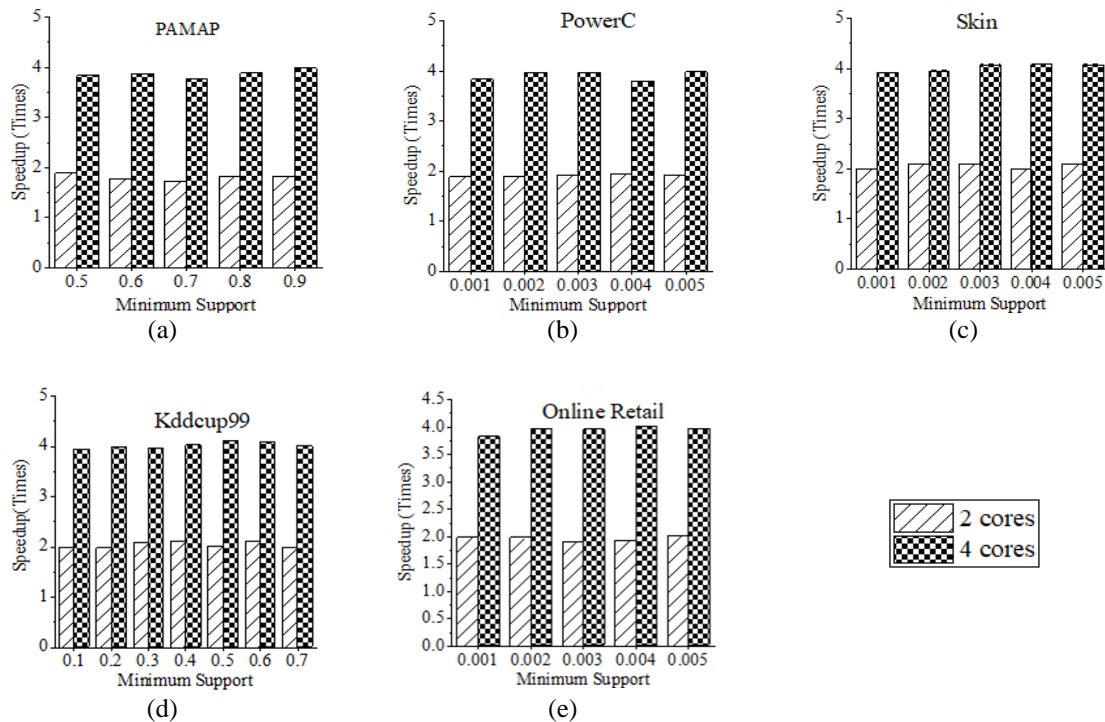


Figure 7. Comparison in the speedup of MHF with different numbers of cores on (a) PAMAP, (b) PowerC, (c) skin, (d) Kddcup99, and (e) OnlineRetail datasets

#### 4.2. Time complexity

##### 4.2.1. Negfin algorithm

The Negfin algorithm comprises of construction of the BMC\_Tree, generation of Nodesets, and construction of frequent itemsets. The construction of BMC\_Tree has a time complexity of  $O(gt \times git \times$

log git) in the worst-case. The next segment involves the generation of the Nodesets consisting of frequent 1-itemsets and this has a computational complexity of  $O(2^{\text{git}})$ . The last segment involves the building of the frequent itemset tree and has a time complexity of  $O(2^{\text{git}} g)$  where  $2^{\text{git}}$  the maximum number of nodes in the tree and  $g$  is the cardinality of NegNodesets. Therefore, the time complexity of the Negfin algorithm is max ( $O(\text{gt} \times \text{git} \times \log \text{git})$ ,  $O(2^{\text{git}})$ ,  $O(2^{\text{git}} g)$ ), which is  $O(2^{\text{git}} g)$ . Here  $\text{gt} = |\text{DB}|$  and  $\text{git} = |I|$ .

#### 4.2.2. PrePost+ algorithm

The PrePost+ algorithm consists of intersection ( ) and Building\_Pattern\_Tree ( ). The aggregate computational complexity of Intersection ( ) is  $O(h + g)$  where  $h$  is the cardinality of the first list and  $g$  is the cardinality of the second list. The second part is Building\_Pattern\_Tree ( ). This has a time complexity of  $O(2^{\text{git}})$ . Therefore, the cumulative time complexity of the PrePost+ algorithm is max ( $O(h+g)$ ,  $O(2^{\text{git}})$ ) which is  $O(2^{\text{git}})$ . Here  $\text{git} = |I|$ .

#### 4.2.3. MA-TDS algorithm

The main components in the MA-TDS algorithm are the construction of the Tree Data Structure, pruning by Apriori technique, and parallelizing by multicore. The construction of the tree has a worst-case time complexity of  $O(2^{\text{git}})$ . Here  $\text{git} = |I|$ . Applying Apriori property requires comparing  $(k-1)$  subsets with  $(k-1)$  itemsets and hence requires  $(k-1) \times (k-1)$  which is  $O(k^2)$  where  $k$  is the length of the itemset. Since this comparison has to cover the entire tree we get  $O(2^{\text{git}} k^2)$ . Since the process is carried out in parallel across  $N$  cores we have  $O(2^{\text{git}} k^2)/N$ . If  $2^k = 1$  we get  $O(1 \times k^2)/N$ .

#### 4.2.4. MHF algorithm

The MHF algorithm invokes NNH and NLH algorithms. The primal component of the NNH algorithm involves constructing the BMC\_Tree. The time complexity of this part in the worst case is  $O(\text{gt} \times \text{git} \times \log \text{git})$ . The second part involves generating Nodesets of all “frequent 1-itemsets”. The time complexity for this component in the worst case is  $O(2^{\text{git}})$ . The proposed NNH algorithm incorporates a hash-based pruning mechanism. And this step has a worst-case time complexity of  $O(\log g)$  where  $g$  is the cardinality of the itemset. Therefore, the time complexity of the NNH algorithm should be max ( $O(\text{gt} \times \text{git} \times \log \text{git})$ ,  $O(2^{\text{git}})$ ,  $O(\log g)$ ) which is  $O(2^{\text{git}})$ . Here  $\text{gt} = |\text{DB}|$  and  $\text{git} = |I|$ .

The two main operations in NLH are NL\_intersection ( ) and Hashing. The resultant time complexity of NL\_intersection ( ) is  $O(h + g)$ . In the NLH algorithm, the pruning is carried out by using hashing which has a worst-case time complexity of  $O(\log g)$ . Therefore, the time complexity of the NLH algorithm should be max ( $O(h + g)$ ,  $\log g$ ) which is  $O(h+g)$  where  $h$  and  $g$  are the cardinalities of  $NL_1$  and  $NL_2$  respectively. Since both the algorithms are running simultaneously the total time complexity is  $(O(2^{\text{git}}) \times O(h+g))$ . Suppose  $l = 2^{\text{git}}$ . Therefore, the total time complexity of the MHF algorithm is  $O(l \times (h + g))$ . If  $N$  cores are used then the total time complexity is  $O(l \times (h + g))/N$ .

### 4.3. Space complexity

#### 4.3.1. Negfin algorithm

The first part of the Negfin algorithm is constructing and traversing the BMC\_Tree, and the space complexity for this component is  $O(g^2)$ . The second part that generates Nodesets has a space complexity of  $O(g)$ . The third part that involves the construction and traversal of the “set enumeration tree” is  $O(2^{\text{git}})$ . Therefore, the space complexity of the Negfin algorithm is max ( $O(g^2)$ ,  $O(g)$ ,  $O(2^{\text{git}})$ ) which is  $O(2^{\text{git}})$ . Here  $\text{git} = |I|$ .

#### 4.3.2. PrePost+ algorithm

The primal component of the PrePost+ algorithm is NL\_intersection ( ) and this part has a space complexity of  $O(h + g)$  where  $h$  and  $g$  are the cardinalities of  $NL_1$  and  $NL_2$  respectively. The PPC tree is not considered for space complexity calculation since it is deleted once the N-lists are obtained. The next step involves the construction of the frequent itemset tree and has a space complexity of  $O(2^{\text{git}})$  where  $\text{git} = |I|$ . Therefore, the space complexity of the PrePost+ algorithm should be max ( $O(h + g)$ ,  $2^{\text{git}}$ ) which is  $O(2^{\text{git}})$ .

#### 4.3.3. MA-TDS algorithm

The main component of the MA-TDS algorithm is constructing the tree data structure and this part has a space complexity of  $O(2^{\text{git}})$ . The pruning does not take any additional space and so is not considered. Hence the total space complexity is  $O(2^{\text{git}})$ . If  $N$  cores are used then the total space complexity is  $O(2^{\text{git}})/N$ .

#### 4.3.4. MHF algorithm

The MHF algorithm consists of NNH and NLH algorithms. Considering the NNH algorithm, the space complexity for constructing and traversal of the tree for mining frequent itemsets is (Number of nodes in the tree  $\times$  Maximum number of items in each node) =  $O(g \times g) = O(g^2)$ . The second part is the generation of Nodesets. The space complexity here is  $O(g)$ . The space complexity of hashing is  $O(g)$  where  $g$  is the space occupied by the  $g$  itemsets. Therefore, the space complexity of the NNH algorithm is  $\max(O(g^2), O(g), O(g))$  which is  $O(g^2)$ . In the NLH algorithm, the space complexity of NL\_intersection is  $O(h + g)$  where  $h$  and  $g$  are the cardinalities of  $NL_1$  and  $NL_2$  respectively. Here pruning is carried out by using hashing which has a worst-case space complexity of  $O(g)$ . Therefore, the space complexity of the NLH algorithm should be  $\max(O(h + g), O(g))$  which is  $O(h + g)$ . Since both the algorithms are running simultaneously, therefore, the total space complexity of the MHF algorithm is  $(O(g^2) \times O(h + g)) = O(g^2(h + g))$ . If  $N$  cores are used then the total space complexity is  $O(g^2(h + g))/N$ .

#### 4.4. Difference between the proposed approach and the competing methods

From the analysis conducted we can conclude that the proposed MHF algorithm outperforms the existing Negfin, PrePost+, and MA-TDS algorithms in terms of runtime, memory, speedup, time, and space complexity. This enhancement in efficacy can be explained as follows. The work presented in this paper is a hybrid approach that takes advantage of the perks of both NegNodesets as well as N-lists. While NegNodesets ensure a concise representation of itemsets, N-lists were found to be faster to extract the resultant set of frequent items. The existing Negfin and PrePost+ algorithms were dependent on the erection and spanning of a set enumeration tree for extracting the resultant set of frequent items which was found to be costly due to the time required for its construction and traversal. The existing MA-TDS algorithm although recent still relied on the apriori principle for pruning out infrequent itemsets which were found to be computationally taxing. The algorithms proposed in this paper on the other hand rely on a novel Hash-based pruning to prune out infrequent itemsets. Only those itemsets that satisfy the minimum support threshold are added to the hash table and the rest are pruned from consideration. In case of collision, it was handled through open addressing. Since the implementation is carried out in Java 8, if the number of items in the hash table grows beyond a threshold Java's HashMap will switch to a balanced tree thereby reducing the computational complexity to  $O(\log g)$ . This was found to be a significant improvement in comparison to the traditional set enumeration tree which had a computational complexity of  $O(2^{g^{it}})$ . This further intensified the efficiency of the mining process.

The deployment of a multi-core-based partition approach whereby the dataset is divided into partitions and distributed between different cores of the processor was found to effectively facilitate load balancing thereby boosting the efficiency of the parallel mining process. We have run the proposed algorithms on two cores as well as four cores. It was observed that the performance nearly doubled as the number of cores increased. Since the implementation is fundamentally based on multi-threading the dependency on employing a Mobile Agent to assimilate the results across different partitions is avoided which further amplifies the performance of the overall approach.

## 5. CONCLUSION





A novel multithreaded hybrid framework MHF has been designed in this paper for the parallel extraction of frequent items using the multicore feature of today's processors. The proposed framework uses a combination of NegNodesets and N-list data structures for mining frequent itemsets. By doing so, the advantages of both have been harnessed here while keeping their drawbacks to a minimum. Existing algorithms that use these structures usually rely on a set enumeration tree to extract the resultant set of frequent items which can be taxing in terms of run time as well as memory. To overcome this drawback a Hash-based pruning approach has been proposed in this paper. As per our discernment, such a framework has not been designed so far. The proposed algorithm has been tested on five datasets downloaded from the UCI repository. Results show that the proposed approach outperforms existing algorithms in terms of run-time memory, speedup, and time and space complexity. As a part of future work, we plan on exploring the efficacy of the proposed framework on streaming data and big datasets.

## REFERENCES





- [1] S. A. Ahmed and B. Nath, "ISSP-tree: an improved fast algorithm for constructing a complete prefix tree using single database scan," *Expert Systems with Applications*, vol. 185, Dec. 2021, doi: 10.1016/j.eswa.2021.115603.
- [2] Y. Xun, X. Cui, J. Zhang, and Q. Yin, "Incremental frequent itemsets mining based on frequent pattern tree and multi-scale," *Expert Systems with Applications*, vol. 163, Jan. 2021, doi: 10.1016/j.eswa.2020.113805.
- [3] A. Shah and Z. Halim, "On efficient mining of frequent itemsets from big uncertain databases," *Journal of Grid Computing*,

- vol. 17, no. 4, pp. 831–850, Dec. 2019, doi: 10.1007/s10723-018-9456-0.
- [4] A. A. Abdelal, S. Abed, M. Al-Shayegi, and M. Allaho, “Customized frequent patterns mining algorithms for enhanced top-rank-k frequent pattern mining,” *Expert Systems with Applications*, vol. 169, May 2021, doi: 10.1016/j.eswa.2020.114530.
- [5] N. Aryabarzan, B. Minaei-Bidgoli, and M. Teshnehlab, “negFIN: an efficient algorithm for fast mining frequent itemsets,” *Expert Systems with Applications*, vol. 105, pp. 129–143, Sep. 2018, doi: 10.1016/j.eswa.2018.03.041.
- [6] Q. Meng and J. Sha, “Tree-based frequent itemsets mining for analysis of life-satisfaction and loneliness of retired athletes,” *Cluster Computing*, vol. 20, no. 4, pp. 3327–3335, Dec. 2017, doi: 10.1007/s10586-017-1080-4.
- [7] Z.-H. Deng and S.-L. Lv, “PrePost+: an efficient N-lists-based algorithm for mining frequent itemsets via children-parent equivalence pruning,” *Expert Systems with Applications*, vol. 42, no. 13, pp. 5424–5432, Aug. 2015, doi: 10.1016/j.eswa.2015.03.004.
- [8] T.-N. Nguyen, L. T. T. Nguyen, B. Vo, N.-T. Nguyen, and T. D. D. Nguyen, “An N-List-based approach for mining frequent inter-transaction patterns,” *IEEE Access*, vol. 8, pp. 116840–116855, 2020, doi: 10.1109/ACCESS.2020.3004530.
- [9] H. Bui, B. Vo, H. Nguyen, T.-A. Nguyen-Hoang, and T.-P. Hong, “A weighted N-list-based method for mining frequent weighted itemsets,” *Expert Systems with Applications*, vol. 96, pp. 388–405, Apr. 2018, doi: 10.1016/j.eswa.2017.10.039.
- [10] B. Vo, H. Bui, T. Vo, and T. Le, “Mining top-rank-k frequent weighted itemsets using WN-list structures and an early pruning strategy,” *Knowledge-Based Systems*, vol. 201–202, Aug. 2020, doi: 10.1016/j.knsys.2020.106064.
- [11] R. Zhang, W. Chen, T.-C. Hsu, H. Yang, and Y.-C. Chung, “ANG: a combination of apriori and graph computing techniques for frequent itemsets mining,” *The Journal of Supercomputing*, vol. 75, no. 2, pp. 646–661, 2019, doi: 10.1007/s11227-017-2049-z.
- [12] S. Dawar, V. Goyal, and D. Bera, “A hybrid framework for mining high-utility itemsets in a sparse transaction database,” *Applied Intelligence*, vol. 47, no. 3, pp. 809–827, Apr. 2017, doi: 10.1007/s10489-017-0932-1.
- [13] Y. Yamamoto, Y. Tabei, and K. Iwanuma, “PARASOL: a hybrid approximation approach for scalable frequent itemset mining in streaming data,” *Journal of Intelligent Information Systems*, vol. 55, no. 1, pp. 119–147, Aug. 2020, doi: 10.1007/s10844-019-00590-9.
- [14] Z. A. Ali, S. A. Rasheed, and N. N. Ali, “An enhanced hybrid genetic algorithm for solving traveling salesman problem,” *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 18, no. 2, pp. 1035–1039, May 2020, doi: 10.11591/ijeecs.v18.i2.pp1035-1039.
- [15] S. Lessanibahri, L. Gastaldi, and C. González Fernández, “A novel pruning algorithm for mining long and maximum length frequent itemsets,” *Expert Systems with Applications*, vol. 142, Mar. 2020, doi: 10.1016/j.eswa.2019.113004.
- [16] M. K. Vanahalli and N. Patil, “An efficient dynamic switching algorithm for mining colossal closed itemsets from high dimensional datasets,” *Data & Knowledge Engineering*, vol. 123, Sep. 2019, doi: 10.1016/j.datak.2019.101721.
- [17] L. Bustio-Martínez, M. Letras-Luna, R. Cumplido, R. Hernández-León, C. Feregrino-Urbe, and J. M. Bande-Serrano, “Using hashing and lexicographic order for frequent itemsets mining on data streams,” *Journal of Parallel and Distributed Computing*, vol. 125, pp. 58–71, Mar. 2019, doi: 10.1016/j.jpdc.2018.11.002.
- [18] U. M. Srinivas and E. S. Reddy, “Partition based single scan method for mining frequent item sets,” *International Journal of Engineering and Advanced Technology*, vol. 8, no. 6, pp. 4917–4922, Aug. 2019, doi: 10.35940/ijeat.F9237.088619.
- [19] U. Mohan Srinivas and E. Srinivasa Reddy, “Mining closed item sets using partition based single scan algorithm,” *International Journal of Recent Technology and Engineering*, vol. 8, no. 2, pp. 3885–3889, Jul. 2019, doi: 10.35940/ijrte.A1920.078219.
- [20] Y. Xun, J. Zhang, X. Qin, and X. Zhao, “FiDooP-DP: data partitioning in frequent itemset mining on Hadoop clusters,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 101–114, Jan. 2017, doi: 10.1109/TPDS.2016.2560176.
- [21] V. Kadappa and S. Nagesh, “Local support-based partition algorithm for frequent pattern mining,” *Pattern Analysis and Applications*, vol. 22, no. 3, pp. 1137–1147, Oct. 2019, doi: 10.1007/s10044-018-0752-x.
- [22] A. Bai, M. Dhabu, V. Jagtap, and P. S. Deshpande, “An efficient approach based on selective partitioning for maximal frequent itemsets mining,” *Sādhanā*, vol. 44, no. 8, Aug. 2019, doi: 10.1007/s12046-019-1158-1.
- [23] X. Niu, M. Qian, C. Wu, and A. Hou, “On a parallel spark workflow for frequent itemset mining based on array prefix-tree,” in *2019 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, Nov. 2019, pp. 50–59, doi: 10.1109/WORKS49585.2019.00011.
- [24] S. Rathee and A. Kashyap, “Adaptive-miner: an efficient distributed association rule mining algorithm on spark,” *Journal of Big Data*, vol. 5, no. 1, Feb. 2018, doi: 10.1186/s40537-018-0112-0.
- [25] C.-S. Wang and J.-Y. Chang, “MISFP-growth: Hadoop-based frequent pattern mining with multiple item support,” *Applied Sciences*, vol. 9, no. 10, May 2019, doi: 10.3390/app9102075.
- [26] D. K. Hanirex and A. M. Kumaravel, “An adaptive approach for association rule mining in cloud computing using hadoop technology,” *Journal of Advanced Research in Dynamical and Control Systems*, vol. 11, no. 7 Special Issue, pp. 1745–1748, 2019.
- [27] M. S. G. Prasad, H. R. Nagesh, and S. Prabhu, “High performance computation of big data: performance optimization approach towards a parallel frequent item set mining algorithm for transaction data based on hadoop MapReduce framework,” *International Journal of Intelligent Systems and Applications*, vol. 9, no. 1, pp. 75–84, Jan. 2017, doi: 10.5815/ijisa.2017.01.08.
- [28] M. Yimin *et al.*, “PFIMD: a parallel MapReduce-based algorithm for frequent itemset mining,” *Multimedia Systems*, vol. 27, no. 4, pp. 709–722, Aug. 2021, doi: 10.1007/s00530-020-00725-x.
- [29] M. Sornalakshmi *et al.*, “An efficient apriori algorithm for frequent pattern mining using mapreduce in healthcare data,” *Bulletin of Electrical Engineering and Informatics*, vol. 10, no. 1, pp. 390–403, Feb. 2021, doi: 10.11591/eei.v10i1.2096.
- [30] S. G. Khawaja, A. Tehreem, M. U. Akram, and S. A. Khan, “Multicore framework for finding frequent item-sets using TDS,” in *Advances in Intelligent Systems and Computing*, vol. 552, Springer International Publishing, 2017, pp. 340–349.
- [31] B. Huynh, B. Vo, and V. Snasel, “An efficient method for mining frequent sequential patterns using multi-Core processors,” *Applied Intelligence*, vol. 46, no. 3, pp. 703–716, Apr. 2017, doi: 10.1007/s10489-016-0859-y.
- [32] A. A. AbdulRazzaq, Q. S. Hamad, and A. M. Taha, “Parallel implementation of maximum-shift algorithm using OpenMp,” *Indonesian Journal of Electrical Engineering and Computer Science (IJECS)*, vol. 22, no. 3, pp. 1529–1539, Jun. 2021, doi: 10.11591/ijeecs.v22.i3.pp1529-1539.
- [33] B. Huynh and B. Vo, “An efficient method for mining erasable itemsets using multicore processor platform,” *Complexity*, vol. 2018, pp. 1–9, Oct. 2018, doi: 10.1155/2018/8487641.
- [34] R. Bhatt and A. Dhall, “Skin segmentation dataset,” *UCI Machine Learning Repository*. UCI Machine Learning Repository, 2012.
- [35] G. Hebrail and A. Barard, “Individual household electric power consumption data set,” *UCI Machine Learning Repository*. 2012.
- [36] A. Reiss, “PAMAP2 Physical Activity Monitoring Data Set,” *UCI Machine Learning Repository*. 2012.
- [37] S. J. Stolfo, W. Fan, W. Lee, A. Prodromidis, and P. K. Chan, “KDD cup 1999 dataset,” *UCI Machine Learning Repository*. 1999.
- [38] D. Chen, “Online retail data set,” *UCI Machine Learning Repository*, 2015.





**BIOGRAPHIES OF AUTHORS**

**Jashma Suresh Ponmudiyan Poovan**     is currently working as an Assistant Professor in the Department of Computer Science and Engineering at Manipal Institute of Technology, Manipal, India, and has 8 years of experience to her credit. She is pursuing a Ph.D. from Manipal Institute of Technology, Manipal Academy of Higher Education in the area of Data Mining. She obtained her master's degree in Computer Science and Engineering from Anna University (2013) and her Bachelor's Degree from Kannur University (2010). Her research interests include Data Mining, Data Analytics, and Machine Learning. She can be contacted at email: jashma.suresh@manipal.edu.



**Dinesh Acharya Udupi**     is currently working as a Professor in the Department of Computer Science and Engineering, Manipal Institute of Technology, Manipal, India, and has over 30 years of experience to his credit. He received his Ph.D. degree from Manipal Academy of Higher Education, Manipal, in 2008. His work has been published in several international journals and has presented articles in various national and international conferences throughout his career. His research interests include data analytics in the healthcare, agriculture, and financial sectors. He can be contacted at email: dinesh.acharya@manipal.edu.



**Nandanavana Veerappareddy Subba Reddy**     is currently working as a Professor in the Department of Information Technology, Manipal Institute of Technology Bengaluru, Manipal Academy of Higher Education, Bengaluru, India and has over 30 years of experience to his credit. He was the former Vice-Chancellor of Mody University, Rajasthan, and also served as the Director of Sikkim Manipal Institute of Technology. Also, he held the office of the Associate Director R&D at MIT, Manipal, and also served as the Department Head of the same institute from September 2001 to September 2007. His work has been published in several international journals and has presented articles in various national and international conferences throughout his career. His research interests include data mining, machine learning, pattern recognition, and image processing. He can be contacted at email: nvs.reddy@manipal.edu.