Rare Pattern Mining

Ian Smith

April 15, 2017

University of Manitoba

# Introduction

Data mining is defined as the non-trivial extraction of implicit, previously unknown, and potentially useful information from data . Rare Pattern Mining is the extraction of interesting patterns that occur very infrequently within a database. Currently, much of the research available on data mining focuses on frequent pattern mining; however, Rare Pattern Mining is a growing field in computer science . My research will demonstrate that a frequent pattern mining algorithm, Frequent Pattern Growth, can be modified to retrieve interesting rare itemsets that can then be studied. I hope to also show that our Rare Pattern Growth algorithm can do this more efficiently than other potential approaches to Rare Pattern Mining.

There are quite a few different applications for Rare Pattern Mining; for one example, Rare Pattern Mining can be employed for fraud detection. In a database of credit card transactions, the rare patterns could turn out to be fraudulent, and the credit card company could use a Rare Pattern Mining algorithm to detect and prevent these transactions from harming their customers. A company that does this might be able to attract more customers by promising them protection.

There are two important ways to determine whether or not a pattern can be considered rare: its support, and its confidence level. Support represents the number of time the items that make up the set occur together in the database . Confidence represents a proportional value that shows how frequently one part of the itemset occurs with the other(s) . Because an itemset with very little support is likely, if not guaranteed to have a low confidence measure, it is only necessary to consider the support value of the itemset to determine whether or not it is rare.

The Frequent Pattern Growth algorithm uses a divide-and-conquer approach toward mining frequent patterns. The frequent single items of each transaction are added one at a time into a tree data structure, and then each frequent itemset is mined from the tree using a recursive function. My Rare Pattern Growth will operate in much the same way. The more frequent rare items will be added into the tree first, and mined from it last.

The organization of this report is as follows: first I will examine existing literature in data mining that focuses on rare itemset mining, as well as Frequent Pattern Growth. Second, I will describe my algorithm to mine rare itemsets using diagrams and pseudo-code. Third, I will analyze the results of my algorithm and compare those results to what was found by other researchers. Finally, I will conclude with an examination of what I learned from running the algorithm, and how it could be applied to the data mining field as a whole. I will also be examining what future iterations could do to improve.

# Literature Review

Aggarwal et al wrote a comprehensive book about the study of frequent pattern mining. This resource has proved invaluable in examining Frequent Pattern Growth (FP-Growth) for this project, and other areas of data mining, as it successfully provides an overview of the different frequent pattern mining methods . In particular, they explained that Apriori [AS94] is a valid but inefficient way of finding interesting itemsets in a database. Apriori’s weakness lies in its need to repeatedly validate the itemsets against the database . FP-Growth, however, only needs to access the database a small number of times, regardless of the size of the database .

Weng proposed an Apriori-based mining approach called Fuzzy Apriori Rare Itemset Mining (FARIM), for mining “specific rare itemsets consisting of quantitative data”.[[1]](#footnote-1) Weng proposed using FARIM for low test or quiz scores in a school setting; if there was a student, or a group of students struggling with class content, then determining exactly what it is they are struggling with would help in finding a solution [5]. Weng believed that his approach would be more successful if it included clustering and classification methods, and if the support parameter was inferred from the data.

Hemalatha et al wrote about finding rare itemsets in data streams, as opposed to static datasets. To that end, they proposed an algorithm for finding Minimal Infrequent Patterns from Data Streams, defined three measures for outlier detection, and created a Minimal Infrequent Pattern based Outlier Detection algorithm. They found, among other things, that their methods were well suited for extracting useful data from sensor data streams and identifying meaningful outliers from those streams.

Wu et al wrote about Attribute-Oriented Induction (AOI), and proposed using AOI to mine negative generalized knowledge from datasets. Their reasoning has to do with medical data; for example, if only a few Taiwanese people were infected with the H1N1 flu virus the number of people that are Taiwanese and have contracted H1N1 will be very small, and not considered a frequent itemset. However, if few Taiwanese contracted H1N1, then that might indicate that the Taiwanese were somehow resistant to the disease.

Agrawal et al presented an overview of how data mining techniques could be used to detect anomalies in datasets. Their Classification approach was to build a model based on the normal behavior of the system, and then feed testing data into that model in order to determine which datasets were anomalous . They tried various different Clustering approaches including k-Means, k-Medoids, and other approaches. They found that hybrid approaches, which combine Clustering and Classification based anomaly detection systems, had the best chance at finding anomalous behaviors in Intrusion Detection Systems.

Lin et al wrote about using the Frequent Pattern Growth algorithm to find frequent itemsets, and in particular to reduce the number of candidate itemsets examined by the algorithm, and reducing the number of times it is necessary to scan the entire database. This is because the Apriori algorithm requires scanning the database repeatedly, and Dynamic Hashing and Pruning algorithm improves the performance of Apriori and lowers the cost of database scanning. But FP-Growth goes even further to improve the performance of Frequent Pattern Mining by reducing the need to access the database even further.

Lin et al then propose using an Improved FP-Growth algorithm to improve the performance of FP-growth. They do this in part by using an address-table structure to lower the complexity of mapping frequent 1-itemsets in an FP-tree, and by using a hybrid FP-tree mining method that reduced the need to rebuild conditional FP-trees. Their simulation shows that their algorithm improved the performance of FP-growth by an order of magnitude in terms of execution time.

Cagliero et al actually use FP-Growth in a similar to fashion to what I am using it for, Rare Pattern Mining; however, they propose a system that weighs the items in the dataset based on their relevance to the mining process. In a traditional approach, the influence of any particular item is ignored beyond its support value in the database. This creates a binary system, either an itemset is interesting or it is not. With a weighted system, itemsets would be of varying importance in the results set .

Bhattacharyys et al wrote about how data mining could be utilized to detect or prevent credit card fraud. According to them $4 billion was lost in 2008. They went on to identify two different approaches for detecting fraud: supervised and unsupervised. Supervised fraud detection methods detect fraudulent transactions by estimating based on samples of both fraudulent and legitimate transactions in order to classify new transactions as one or the other . In unsupervised fraud detection models outliers are identified as potential cases of fraud and sought out.

Bhattacharyya et al then went on to examine two different mining techniques for finding fraudulent transactions: random forests and support vector machines, together with logical regression . They concluded that the random forests demonstrated better performance and greater efficiency compared to support vector machines . They suggested that future study should focus on the different varieties of fraudulent behavior, for example the difference between stolen and counterfeit credit cards.

Yu wrote about finding outliers in very large datasets. This is because, as they put it, “Modern companies are awash in data on customers, clients, suppliers, and industry trends” . Their contribution, called FindOut, was intended to detect outliers in complicated data patterns of various densities. It did so using signal-processing techniques and a novel deviation outlier detection approach. It was also successful in indentifying various percentages of outliers in large datasets .

Adda et al wrote about finding rare patterns. They modeled their approach on the Apriori algorithm, and used it to detect abnormal usage in web applications. They found that their approach was flexible and able to detect suspicious behaviors not seen before . They also provide some insight into the importance of rare patterns in different scientific fields, such as biology, medicine, and in information security. With security, rare patterns might suggest suspicious or perhaps malicious intentions on the part of the user.

# Description

The software prototype, which was written in C++ and compiles using G++, operates in four steps. First, it reads in the transaction database, then it creates an array of Transaction objects, then it removes the non-rare items from those transactions, then it adds those transactions one-at-a-time to a tree structure, then it analyzes that tree structure to find all the rare patterns in the database. All of these steps are completed for each transaction database that the software uses, and all the objects that are created are deleted before it can move on to the next database.

Each database is then used with varying levels of maximum support, to provide a more complete demonstration of the software’s capabilities. I have provided some sample code for select functions and for each step of the software’s operations. This code is a close facsimile for what is present in the source code of the software.

## Transaction Stage

The transaction stage is fairly simple to understand. The software reads in the transaction database. The databases are all stored in the form of a text files with each line indicating a transaction, except the first line indicates the number of transactions contained in the file. Each line after that first one represents a transaction, where the first number is the transaction’s unique ID number. After that, on the same line, is the number of items in the transaction. Lastly are the items themselves, also indicated by natural numbers. The simplest transaction database the software uses, called PreciseDB.txt, is shown below.



Figure : Contents of PreciseDB.txt

This transaction database contains four transactions, which have IDs of one through four. The largest transaction has four items (1, 2, 3, and 5), and the smallest only has two (2 and 5). The transaction stage will return a TransactionList containing four transactions. Some basic pseudo-code for this stage follows.



Figure : get\_transactions function

This function takes the contents of the database, the number of transactions, and the initial transaction list as parameters. Each line is used to create a transaction, and then that transaction is added to the list. The transaction list is updated in place, so there is no need to return it. The transaction itself is comprised of the ID, number of items, and the items themselves. All of these are stored in memory integers.

## Pruning Stage

The Pruning Stage is where the non-rare items are removed from all the transactions in the database. The database itself is not altered, but any number of Transaction objects that the software uses could be. This stage is completed in several steps. First, every item in the database is added to a Set object, along with its support.

get\_itemset()

{

total = 0

for (i = 0; i < present; i++)

{

// total represents the largest possible number of items in the database

total+=list[i]->get\_length()

}

set = new Set(total)

if (set != NULL)

{

// first get the support of all items in the transaction database

for (i = 0; i < present; i++)

{

items = list[i]->get\_items()

length = list[i]->get\_length()

for (j = 0; j < length; j++)

{

curr = items[j]

item = new Item(curr)

// add\_item automatically increments support if the item is already present in the database

set->add\_item(item)

}

}

set->resize(set->get\_present())

}

return set

}

Figure : get\_itemset function

The code above creates a new set to be used for the header table for the tree. Each item in the database is added to that set, which automatically increments the support of an item if it is added more than once, rather than storing that item multiple times. After this the non-rare items are removed from the set and from the list of transactions.

Next, the actual pruning begins .The software removes each item that has too much support in the database from the Itemlist. Once again, this is a multi-step process. First, the non-rare items must be removed from the header table stored as a set, and then those non-rare items have to be removed from the transactions themselves. Any transaction that contains nothing but non-rare items is removed from the list.

Set::remove\_non\_rare\_items(max\_support)

{

non-rares = 0

for (i = 0; i < present; i++)

if (set[i]->get\_support() > max\_support)

non-rares++

int new\_size = size - non-rares

// create replacement array of ListItems

new\_set = new ListItem \*[new\_size]

next = 0

for (i = 0; i < present; i++)

{

if (set[i]->get\_support() <= max\_support)

{

n = new Item(curr->get\_name(), set[i]->get\_support())

new\_set[next] = n

next++

}

}

this->present = next

this->size = new\_size

this->set = new\_set

}

TransactionList::remove\_non\_rare\_items(set)

{

revised = 0

for (i = 0; i < present; i++)

{

temp = list[i]->remove\_non\_rare\_items(set)

if (temp != NULL)

{

revised++

}

list[i] = temp

}

replacement = new Transaction\*[revised]

index = 0

for (i = 0; i < present; i++)

{

if (list[i] != NULL)

{

replacement[index] = new Transaction(list[i])

index++

}

}

list = replacement

present = revised

size = revised

}

Figure : Removing non-rare items

The first function, which is part of the Set class, is responsible for removing non-rare items from the header table. It first creates a new set large enough for all the rare items. Then it creates copies of all the rare items and adds them to the new set. The second function takes this header table and applies it to the list of transactions. It determines the number of transactions that will still have items. If a transaction contains only non-rare items it is not added to the new list.

Both the list of transactions and the header table are sorted using a modified quick sort algorithm in this stage as well. The items with the most support are placed at the front of both. This optimizes the tree, making it more likely to be a balanced, and therefore more efficient, tree structure. The last step of this stage is to print out the revised transaction list and the revised header table to the console.

## Construction Stage

In this stage, the transactions are used to construct a Rare Pattern tree. This tree has an array of roots rather than a single one. For the input file PreciseDB.txt, which was shown above, the final RPTree, along with the header table, looks like this:

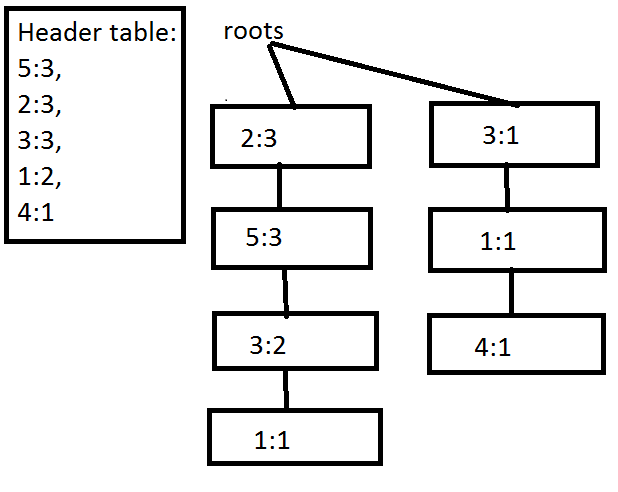


Figure : PreciseDB RPTree with Maximum Support of 3

The reason it looks the way it does has to do with the way the transactions are added to the tree. The items with the most support are the ones that are likely to have the most children. The tree that resulted from the software in this instance was perhaps not the most efficient shape it could have been, since it is two smaller trees where no node has more than one child. However, this database is simple enough to get away with this, and it is likely that the trees constructed for larger databases will be more efficient. If they are not, then that says something about the data which will be addressed later.

RPTree::add\_transaction(transaction)

{

items = transaction->get\_items()

size = transaction->get\_length()

added = false

if (present > 0)

{

for (i = 0; i < present && !added; i++)

{

item = roots[i]->get\_item();

name = item->get\_name();

for (j = 0; j < size && !added; j++)

{

if (name == items[j])

{

temp->increment\_support();

// remove items[j] from items array

int rep[size-1];

int q;

for (q = 1; q < size; q++)

{

rep[q-1] = items[q];

}

size--;

roots[i]->add\_transaction(rep, size);

added = true;

}

}

}

// added is false if the transaction has nothing in common with any of the roots

if (!added)

{

// none of the items in the transaction correspond to a branch, need a new root

q = new Item(items[0])

add = new TreeNode(q)

replacement[size-1]

for (i = 1; i < size; i++)

replacement[i-1] = items[i]

size--

items = replacement

add->add\_transaction(replacement, size)

add\_root(add)

}

}

else

{

// first root case

q = new Item(items[0])

add = new TreeNode(q)

replacement[size-1]

for (i = 1; i < size; i++)

replacement[i-1] = items[i]

size--

add->add\_transaction(replacement, size)

add\_root(add)

}

}

Figure : RPTree::add\_transaction

This function shows how each transaction is added to the RPTree data structure. First, the roots are examined to see if the transaction can be added to any of the sub-trees. If so, then the support value of the root is incremented and the transaction is recursively added to that root’s children. Otherwise it is necessary to create a new root for the tree. The rest of the transaction is then added recursively to that new root. The third case accommodates adding the very first transaction to the tree, which is done in much the same way.

void TreeNode::add\_transaction(array, size)

{

if (size > 0)

{

// first case: this node has no children

if (children\_number == 0)

{

i = new Item(array[0])

child = new TreeNode(i)

add\_child(child)

new\_array = revise\_array(array, size)

child->add\_transaction(new\_array, size-1)

}

else

{

// Second, try to add the transaction to one of the children

curr = NULL;

stop = false;

for (i = 0; i < children\_number && !stop; i++)

{

curr = children[i];

name = q->get\_name();

for (j = 0; j < size && !stop; j++)

{

if (name == array[j])

{

stop = true

curr->increment\_quantity()

swap(0, j, array)

int \*new\_array = revise\_array(array, size)

curr->add\_transaction(new\_array, size-1)

}

}

}

// Third, create a new group of children for this node

if (!stop)

{

i = new Item(array[0])

i->set\_support(1)

child = new TreeNode(i)

add\_child(child)

new\_array = revise\_array(array, size)

child->add\_transaction(new\_array, size-1)

}

}

}

}

Figure : TreeNode::add\_transaction

## Mining Stage

The final stage of the software is to recursively mine the RPTree to get a list of rare patterns that are present in the tree. Since the only items that are present in the tree at all are themselves rare, there’s no need to ensure that the sets returned are rare. However, it is interesting to note the closed itemsets.

The examine method for the RPTree class is quite straight forward. It calls each root’s examine method and then combines the rare patterns found by that method with those of all the other roots. The Node’s examine method is more complex, and it was also the source of a number of memory leaks that needed plugging while writing the software.

TreeNode::examine()

{

set = new Set()

singleton = new Set(this->item)

set->add\_item(singleton)

if (children\_number > 0)

{

for (i = 0; i < children\_number; i++)

{

// examine each child node

Set \*child\_set = children[i]->examine()

// merge only adds the contents of the new set to set right now

set->add\_sets(child\_set)

child\_set->add\_item\_to\_sets(q);

set->add\_sets(child\_set);

}

}

return set;

}

Figure : TreeNode::examine

This function creates and combines several very similar sets. First, it creates a set for the item stored by the current node, and adds it to the set that will be returned. This is possible because Sets contain a number of ListItems, of which Sets are a subclass. Anyway, it calls examine recursively for each child node, and then combines the resulting set with the one it will be returning, and also adds the node’s item to each of the sets returned by the child’s examine method.

Both of the Set methods that are called in TreeNode::examine are very simple. The add\_sets method simply adds the sets returned by the child’s examine method to the set used to call the method. The add\_item\_to\_sets method adds the supplied item to each subset stored within a set. The rare patterns are then printed to the console, and then added to the output file corresponding to the trial number.

# Analysis

I was able to use the software prototype to find and determine the support values for a number of different rare itemsets. In this section I will examine the results of the experiments, and press the significance of being able to find the rare itemsets in a transactional database using a Rare Pattern Growth algorithm over some of the other possibly approaches, namely the Apriori-based approach. Lastly, I will compare the results of the algorithm to the results of other documented experiments.

## Results

I was successful in finding quite a few different rare itemsets in the transaction databases used by the software. For example, when running the software using the simplest transaction database, the previously mentioned PreciseDB.txt with a maximum support value of three, it returns twenty different rare itemsets. These itemsets are shown in the table below, along with their support values.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Itemset | Support | Frequency |
| 1. | {2} | 3 | 0.75 |
| 2. | {3} | 3 | 0.75 |
| 3. | {5} | 3 | 0.75 |
| 4. | {1} | 2 | 0.5 |
| 5. | {1,5} | 1 | 0.25 |
| 6. | {5,3} | 2 | 0.5 |
| 7. | {1,3} | 2 | 0.25 |
| 8. | {1,5,3} | 1 | 0.25 |
| 9. | {3,2} | 2 | 0.5 |
| 10. | {5,2} | 3 | 0.75 |
| 11. | {1,2} | 1 | 0.25 |
| 12. | {1,5,2} | 1 | 0.25 |
| 13. | {5,3,2} | 2 | 0.5 |
| 14. | {1,3,2} | 1 | 0.25 |
| 15. | {1,5,3,2} | 1 | 0.25 |
| 16. | {4} | 1 | 0.25 |
| 17. | {4,3} | 1 | 0.25 |
| 18. | {4,1} | 1 | 0.25 |
| 19. | {4,3,1} | 1 | 0.25 |

Table : PreciseDB with a Maximum Support of 3 Rare Itemsets

Equation : Frequency equation

One of the most important things that is shown in the table is that nearly all of the itemsets are closed. A closed itemset is one that either does not have a superset, or whose superset has less support. Even though items 2, 3, and 5 occur three times in the database, there is no itemset {2, 3, 5} with a support value of three; instead, it has a support value of two. This is because item 3 occurs in a transaction alongside items 1 and 4, while 2 and 5 always seem to coincide.

These itemsets could refer to any number of things in the real world. I prefer to use the grocery store example. For example, items 2 and 5 could be bread and eggs, while item 3 could be milk. If that was the case, then we would be able to tell that every one of these four transactions contain both bread and eggs, and two thirds of the transactions with bread and eggs also include milk. A possible interpretation of such a phenomenon would be that people like to have their eggs with bread (or toast), and most people also like to have milk at the same time.

It is also important to show how my decision to employ a Rare Pattern Growth approach to this problem panned out. This algorithm may not be as efficient as one that uses more primitive data types rather than large number of different classes, but I believe it is considerably more efficient than any approach that could have been built upon the Apriori algorithm. The reason for this is quite simple. The Apriori algorithm requires repeatedly reading the database in order to determine the support value of any itemset. Since I am using databases that contain increasingly large numbers of transactions, looking up the support value of every potentially rare itemset would be quite demanding on a system. Every iteration of the algorithm would require reading the list of transactions, and every single item in those transactions.

## Comparisons to Literature

Rare Pattern Growth approaches appear to the second most common way to mind a database for rare itemsets, after the Apriori algorithm. Considering the solution present in this software, it is easy to understand why it is so common. There are two main reasons: first, it requires far fewer database scans than Apriori, making it noticeably more efficient; and second, it uses recursion to find and build upon the rare singletons found in the header table. Furthermore, it is able to reliably recreate the transactions in the database without re-reading the database.

The benefits of using a growth-based approach to data mining are well documented in the journals. First, it is stated in that a growth approach reduces the number of times it is necessary to scan the entire database compared to the Apriori algorithm, regardless of whether the Dynamic Hashing and Pruning or Partition approaches are employed. These approaches reduce the need to scan the database repeatedly; however they do not eliminate it. In a growth approach it is possible to access the database a consistent number of times regardless of the itemsets; whereas in the Apriori algorithm, it is necessary to scan the database at the end of each stage of the algorithm in order to determine the support of each new candidate itemset .

I did not employ any sort of clustering- or classification-based approach in order to find particularly interesting rare itemsets. There is a good chance that such an approach would be able to eliminate some of the rare itemsets from the ones returned by the software; and, considering just how many itemsets are being returned under some conditions, this would be a worthy endeavor. However, since the datasets being used by this software are static, classification and clustering may not be as useful as it was found to be in . They used a hybrid clustering- and classification-based approach to model the normal behavior of a system to then find outliers in a stream.

I believe that the approach I have employed has been successful in finding the rare itemsets in a transactional database. It is every bit as successful as doing so as an Apriori-based approach would have been, but it is considerably more efficient in doing so. For example, in they use an approach called FARIM, which was able to mine test scores to determine which students were struggling with class material, and which material they were struggling with. In larger datasets, such as the number of students that took a class over a long period of time, it is not necessary to scan the database more than a few times. In a smaller database, the same number of scans will be required, but with a negligible penalty.

These papers all show how useful a growth-based approach can be in Rare Pattern Mining. Furthermore, they show help to show how a growth-based approach would be an improvement over any Apriori-based approach simply because it greatly reduces the need to scan the database. With a large database, such scans become quite costly.

# Conclusion

The Rare Pattern Growth approach to finding rare patterns in a transactional database has proved to be quite successful. It is considerably more efficient than finding rare patterns in a database by using any Apriori-based approach, which would require numerous scans of what could be a very large database, which would likely be exceedingly costly. With a tree-based approach, it is possibly to find the rare patterns while minimizing the number of times it is necessary to scan the database.

This approach operated in a total of four stages: the Transaction stage, which read in the database and created a list of Transaction objects; the Pruning stage, which removed the non-rare items from both the Transaction list and the Header table; the Construction stage, which built the RP-tree; and the Mining stage, which recursively examined the tree to find the rare multi-item-sets in the tree. The rare itemsets returned by the mining stage were then printed to a file along with their support values.

In the future, it would be worthwhile to execute more of the algorithm within the RPTree class itself. In , they describe an approach where the header table, which stores the values of each item in the transactional database that is in the tree, is a part of the tree. As transactions are added to the tree, the header table is updated to reflect the updated values of each item. This would require the transactions to be pruned of non-rare items at the same time. This would likely improve efficiency because the transactions would be able to be sorted one at a time to reflect the items in the tree already, which might lead to a more balanced tree that would be more efficient to mine itemsets from.

In conclusion, I believe that the software I have constructed is successful in mining rare patterns from datasets of various sizes, even ones as large as to contain ten thousand transactions. RP-Growth is far more efficient than any Apriori-cased approach because it reduces the need to repeatedly scan the database, and because it uses recursion to mine the tree to find the rare itemsets.

# Bibliography

|  |  |
| --- | --- |
| [1] | D. Yu, G. Sheikholeslami and A. Zhang, "Finding Outliers in Very Large Datasets," *Knowledge and Information Systems,* vol. 4, pp. 387-412, 2002. |
| [2] | L. Cagliero and P. Garza, "Infrequent Weighted Itemset Mining using Frequent Pattern Growth," *IEEE Transactions on Knowledge and Data Engineering,* vol. 26, no. 4, pp. 903-915, April 2014. |
| [3] | M. Adda, L. Wu, S. White and Y. Feng, "Pattern Detection with Rare Itemset Mining," *International Journal on Soft Computing, Artificial Intelligence and Applications,* vol. 1, no. 1, pp. 1-17, August 2012. |
| [4] | C. C. Aggarwal and J. Han, Eds., Frequent Pattern Mining, Springer Cham Heidelberg, 2014. |
| [5] | C.-H. Weng, "Mining fuzzy specific rare itemsets for education data," *Knowledge-Based Systems,* vol. 24, no. 5, pp. 697-708, July 2011. |
| [6] | C. S. Hemalatha, V. Vaidehi and R. Lakshmi, "Minimal infrequent pattern based approach for mining outliers in data streams," *Expert Systems with Applications,* vol. 42, no. 4, pp. 1998-2012, March 2015. |
| [7] | Y.-Y. Wu, Y.-L. Chem and R.-I. Chang, "Mining negative generalized knowledge from relational databases," *Knowledge-Based Systems,* vol. 24, no. 1, pp. 134-145, February 2011. |
| [8] | S. Agrawal and J. Agrawal, "Survey on Anomaly Detection using Data Mining Techniques," *Procedia Computer Science,* vol. 60, pp. 708-713, 2015. |
| [9] | K.-C. Lin, I.-E. Liao and Z.-S. Chen, "An improved frequent pattern growth method for mininig association rules," *Expert Systems with Applications,* vol. 38, no. 5, pp. 5154-5161, May 2011. |
| [10] | S. Bhattacharyya, S. Jha, K. Tharakunnel and J. C. Westland, "Data mining for credit card fraud: A comparative study," *Elsevier,* vol. 50, no. 3, pp. 602-613, February 2011. |
| [11] | R. S. Debreceny and G. L. Gray, "Data mining journal entries for fraud detection: An exploratory study," *International Journal of Accounting Information Systems,* vol. 11, no. 3, pp. 157-181, September 2010. |

1. The Apriori algorithm is a frequent pattern mining algorithm where frequent single-item-sets are combined to create larger frequent itemsets, and then the database is scanned to determine the support of the new itemsets. This process continues until there are no more itemsets that can be combined [4] [↑](#footnote-ref-1)