Vertical Minimal Rare Pattern Mining in Sparse, Weakly Correlated Data

Elieser Capillar
Faculty of Computer Science
University of Manitoba
Winnipeg, Canada
capillae@myumanitoba.ca

Prabhanshu Shrivastava
Faculty of Computer Science
University of Manitoba
Winnipeg, Canada
shrivasp@myumanitoba.ca

Ngoc Bao Chau Truong
Faculty of Computer Science
University of Manitoba
Winnipeg, Canada
truonnbc@myumanitoba.ca

Chowdhury Abdul Mumin Ishmam
Faculty of Computer Science
University of Manitoba
Winnipeg, Canada
ishmacam@myumanitoba.ca

Abstract— Rare itemsets are essential forms of patterns with several real-world uses, like the interpretation of biological data, mining rare association rules between diseases and their causes, genetics, or anomaly detection. However, extracting rare itemsets from databases presents a number of challenges. In this paper we propose an efficient foundation for mining minimal rare itemsets within sparse, weakly correlated datasets. The proposed approach makes use of an existing vertical mining algorithm for frequent patterns VIPER, and modifies it to find minimal rare patterns in an efficient manner. Experimentation of our algorithm, *MR.VIPER*, was done against horizontal rare itemset algorithm AprioriRare which also discovers minimal rare patterns. We present some empirical data for the performance improvements brought by the optimized strategies.

Keywords—Vertical Mining, Minimal, Rare Pattern, Rare Itemset, VIPER, Sparse Data

1. Introduction & Motivation

Data mining [1, 2] is a set of techniques used for the extraction of implicit, previously unknown, and potentially useful information in the form of patterns within large databases. Since its conception [3, 4], research regarding finding frequent patterns and association rules has been a main focus in the field. Association rule mining aims to find associations between items in databases. The approach is to first determine all frequent patterns within the database and further mine these frequent patterns to find association rules between them. The research for this is still ongoing and so far has provided an incredible foundation for the field of Data Mining. It has played an important role in the mining of other types of patterns (eg., emerging patterns [5], sequential patterns [6], and quantitative mining[7]).

Frequent patterns are normally thought to show recurrence in data, that is, they reveal consistencies and regularities

Sparsity = 30%

TID	Itemsets
T1	ABDE
T2	BCE
T3	ABDE
T4	ABCE
T5	ABCDE
T6	BCD
T7	ABCD
T8	BD
T9	CDE
T10	ADE

Sparsity = 58%

TID	Itemsets
T1	AB
T2	ABE
T3	D
T4	BC
T5	CE
T6	AB
T7	AC
T8	ABC
T9	AC
T10	BC

Figure 1. Two datasets. The left has a density of 70% and the right has a density of 42%

within data. It can be easily inferred that something that is frequent must be important and by extension, something that is not frequent must not be important. In practice however, this is not always the case, and the mining of rare patterns [8] can result in more interesting results than mining for their frequent counterparts.

Rare patterns are worth researching into because they represent infrequent patterns in a database. These infrequent patterns are particularly useful in the fields of biology, medicine, and security.

For example, within the medical field of pharmacovigilance- which is the science of detecting, assessing, and studying adverse drug effects- mining for rare patterns can result in the discovery of associating drugs with adverse effects [9]. Provided with a database of drug effects, if "{Drug} ∪ {Effect A}" is found to be a frequent pattern, and "{Drug} ∪ {Effect B}" is found to be a rare pattern, this information can be used to determine whether or not these are desired and expected effects.

Another example could be within the field of computer network security. Given a network and a database of connections to that network, mining for rare patterns can be done to easily isolate uncommon and unusual connections. From this list of rare connections, it can be further analyzed to determine if any were malicious.

There are already a few algorithms that also mine for minimal rare patterns. These being AprioriRare [9], MRG-Exp [9], and Walky-G [10]. AprioriRare was the foundation that MRG-Exp and Walky-G was built off of. In their respective papers, these optimized algorithms both found that their algorithms performed best when used on dense, highly correlated data. When it came to sparse, weakly correlated data, the runtime and search space covered by these algorithms were similar to that of AprioriRare.

As a remedy for this, we seek to improve the runtime for finding minimal rare patterns in sparse, weakly correlated data. Our approach is to modify an existing vertical mining algorithm to mine for

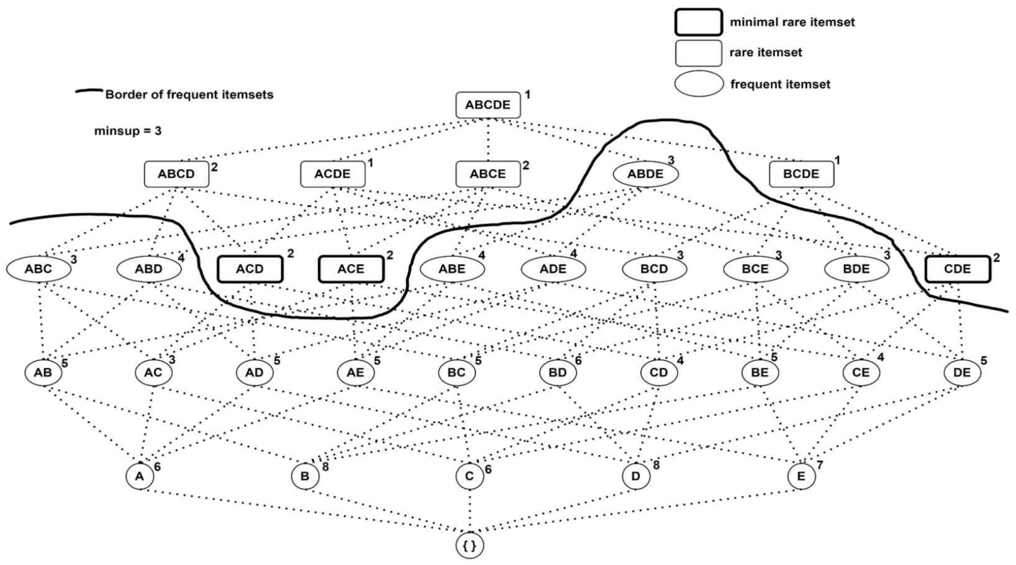

Figure 2. The powerset lattice for the *left* dataset in Figure 1.

minimal rare patterns instead of frequent ones.

In this paper, we propose MR. VIPER, a level-wise, breadth-first vertical algorithm as a foundation that mines for these minimal rare patterns. The first step is to convert a given database to a vertical format if it is not already in a vertical format. Next is to store all singleton *tidsets* in a hash structure. Lastly, perform breadth-first candidate generation and mine for minimal rare patterns.

2. Background & Existing Works

As already mentioned in Section 1, there are already a few algorithms that mine for minimal rare patterns. Again, they are AprioriRare [9], MRG-Exp [9], and Walky-G [10].

First, AprioriRare is a naive horizontal approach that mines for minimal rare patterns. AprioriRare performs its search breadth-first while taking advantage of the downward closure property. It performs its search naively. That is, it performs Apriori as normal, finding all minimal rare patterns as a by-product.

Second, MRG-Exp, is a more efficient algorithm modified from AprioriRare to reduce the search space covered when mining for minimal rare patterns. It achieves this by looking for generators instead of simply itemsets. It is able to find minimal rare itemsets this way because by nature of a generator, if an item is not a frequent generator, it is a minimal rare itemset.

Lastly, Walky-G is a highly optimized vertical algorithm that utilizes a depth-first, right to left approach to massively cut down the computation time required to find minimal rare itemsets. Similar to MRG-Exp, it also looks for generators while using a hash structure for quick access to already found frequent generators. The algorithm abuses the fast lookup times of a hash table when finding all proper subsets of an itemset with the

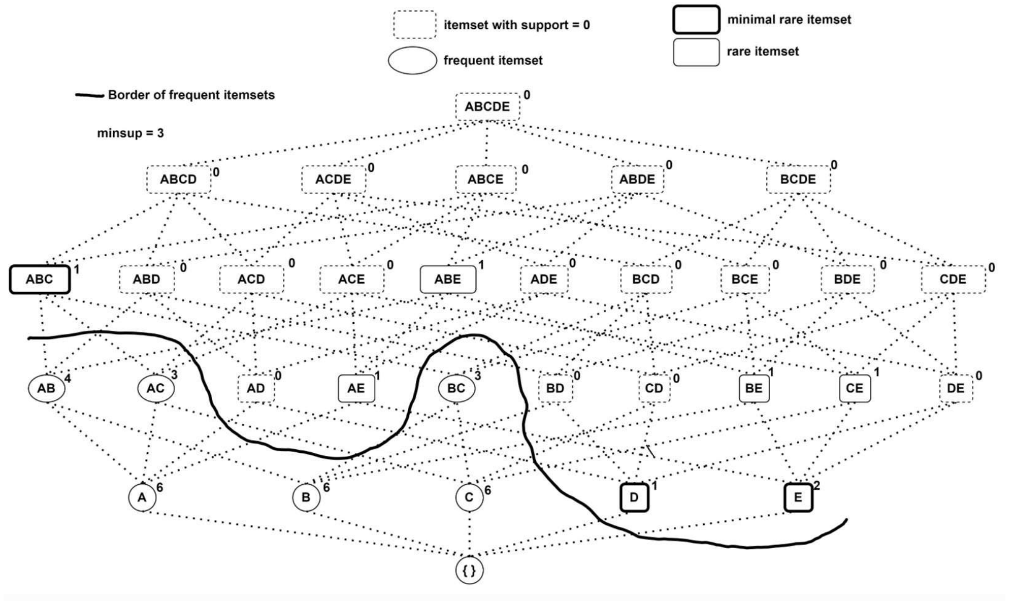

Figure 3. The powerset lattice for the *right* dataset in Figure 1.

same support. This results in fast candidate generation due to the rapid tests for any itemset's generator status.

AprioriRare provided the baseline speed for minimal rare pattern mining. MRG-Exp and Walky-G were able to improve the speed and as mentioned above, they both worked best when used on dense, strongly correlated data. We will look to improve the speed for mining minimal rare patterns on sparse, weakly correlated data.

3. Basic Concepts

Here, we provide a brief background about vertical mining, rare pattern mining, and give a brief explanation to the reasoning behind our approach.

A horizontal database is a set of transactions, where a transaction is a set of items called an *itemset*. Each transaction has a unique identifier (*tid*).

A vertical database is a set of single item equivalence classes, where each equivalence class for an item is the set of transactions (or *tidsets*) that contain that item. There is one equivalence class for each single domain item.

Definition 1. An itemset is a rare itemset *if its support is less than the minimum support threshold.*

Definition 2. An itemset is a minimal rare itemset (mRI) *if it is rare but all of its proper subsets are frequent*.

3.1 Breadth-First Mining for mRIs

Due to the nature of sparse, weakly correlated data, frequent itemsets don't tend to get too large. Take the datasets in Fig 1. The left represents a dataset with a density of 70% and the right represents a dataset with a density of 42%.

From the Hasse Diagrams of these two datasets (Fig 2 and Fig 3 respectively) we can see that minimal rare itemsets lie just

Algorithm 1 ("MR. VIPER" function):

Method: find all **minimal rare itemsets**
Input: **transactional database** (*T*), **maximum transaction length** (*max*), **minimum support** (*minsup*)
Output: list of all **minimal rare itemsets** (*minRare*)

```
 1) level ← 1;
 2) minRare ← {infrequent singletons}; // All minimal rare itemsets
 3) Frequents ← {frequent singletons}; // All Frequent 1-itemsets
 4) Construct vertical tidVectors of each frequent item;
 5) while (|Frequents| ≥ level + 1 && level + 1 ≤ max)
 6) {
 7)         $Frequents_{level+1}$ = generateCandidates(Frequents);
 8)         level ← level + 1;
 9)         sort the $Frequents_{level+1}$
10) }
11) return minRare;
```

past the border of frequent itemsets. Along with our observation about frequent itemsets in sparse datasets, taking a breadth-first approach will minimize the amount of unnecessary candidates generated.

3.2 EClat vs. VIPER

EClat [11] and VIPER [12] are very similar algorithms. The main difference between the two is how their respective *tidsets* are stored. EClat uses equivalence classes while VIPER uses bit vectors. Candidate generation with these two algorithms are virtually the same. By comparing the main limitation of EClat, we can see why storing *tidsets* as bit vectors is an advantageous choice.

When performing the EClat algorithm, the most expensive operation is the intersecting of equivalence classes. Depending on the implementation of these equivalence classes, this can become a major bottleneck for the overall computation time. By storing these equivalence classes as bit vectors, in order to "intersect" two *tidsets*, only a bitwise AND operation is required, which is a very cheap operation. This is why we decided to start our algorithm by modifying VIPER.

4. MR. VIPER

In this section, we present the algorithm *MR. VIPER*, the main focus of this paper. It is a vertical algorithm modified from *VIPER* to mine for mRI's as opposed to frequent itemsets.

4.1 Optimizations

MR.VIPER takes advantage of a hash structure which is a simple dictionary containing key/value pairs. A key is a frequent itemset and the value is its tidset represented as a bit vector. Through the use of this data structure, we can abuse its quick

Algorithm 2 ("generateCandidates" function):

Method: Generate all probable candidates level-wise
Input: list of **frequent-itemsets** of length **level (***Frequents***)**
Output: list of all **frequent** in level + 1 (*Candidates*)

```
1)  result ← ∅   //List of Candidates
2)  Generate Candidate by joining itemset-pairs in Frequents;
3)      if (checkSubset(candidate,Frequents))
4)      {
5)          Remove the candidate;
6)          Calculate the bitset by performing And on itemset-pairs items
7)          Add frequent candidates to result;
8)          Save the infrequent as mRI based on support;
9)      }
10) return result;
```

lookup times for finding proper subsets when generating two-level candidates.

As already mentioned in section 3.2, the intersection of tidsets is the most expensive operation. Along with this fact, candidate generation (which includes the intersection of tidsets) dominates the computation time. By storing tidsets as bit vectors, we can abuse tidsets.

By ordering the frequent singletons found after the first pass of the dataset in ascending order, we are able to cut down on the number of candidates generated.

4.2 Implementation

Mr.Viper creates the hashMap as the first step to store all distinct items in the transactional database as a preprocessing step. Next step is scanning the database line by line or one transaction at a time. During this scan, each distinct item is placed in the hashMap and where the item is the 'Key' and 'Value' is the bit vector representing the set of transactions the item is in. This is the only time we scan the database and so we save time in calculating supports of other itemsets at higher levels.

From the hashMap we calculate each distinct item's support. If an item is frequent it gets added to the list of frequent items. All infrequent items are minimal rare itemsets as all singletons have only one proper subset which is frequent i.e the empty set. Then we sort the frequents first by support and then by lexical order.

The next step involves merging single frequent items to produce the second-level candidates. The method *mergeSingle(list, minsup)* does this while returning the frequent two itemsets by performing a bitwise AND operation on their bitsets, to produce the bitset of the resultant itemset. This step repeats until there are no more frequent itemsets generated or the level of the largest transaction length is reached. After each level of frequent itemsets, they are sorted in ascending order.

Algorithm 3 ("checkSubset" function):

Method: determines whether or not all proper subsets of the provided itemset are frequent
Input: an **itemset** (*itemset*), a list of **all frequent itemsets** (*Frequents*)
Output: a **boolean**

```
1)  if (itemset.length > Frequents.size)
2)  {
3)          return false;
4)  }
5)  subsetFound ← false
6)  for each(fqItemset in Frequents)
7)  {
8)          while (index < itemset.length)
9)          {
10)                 subset ← all the items in itemset except the item at index
11)                 if (subset is in Frequents)
12)                 {
13)                         subsetFound ← true
14)                         return subsetFound
15)                 }
16)                 index++
17)         }
18) }
19) return subsetFound
```

The method *generateCandidate(list)* generates candidates and then calls *checkSubset(itemset, list)* which generates all proper subsets by eliminating one element at a time from the itemset. It then checks its presence in the frequents list and returns whether or not it is found. True returns are added to the *Frequents* list if they meet the minimum support. True returns that do not meet the minimum support are saved as a minimal rare itemset.

5. Experimental Results

In our experiments, we tested MR. VIPER against AprioriRare. Unfortunately, we were unable to find an implementation for MRG-Exp and Walky-G to test our algorithm against. MR. VIPER and AprioriRare were both implemented in Java. The experiments were performed on an Intel Core i5-8265U 1.60GHz CPU on Windows 10 with 8 GB of RAM. The Java function *System.currentTimeMillis* was used between input and output to determine our reported times.

For our experiments, we used the following datasets: T20I6D100K, T25I10D10K, C20D10K, and C73D10K. These datasets were found online on SPMF, an open-source, Java Data Mining library. T20 and C20 were used in the AprioriRare, MRG-Exp and Walky-G papers. We chose

minsup	Execution Time (seconds) AprioriRare	Execution Time (seconds) MR. VIPER	Number of Minimal Rare Itemsets found
T20I6D100K			
10%	0.232	0.301	907
9%	0.331	0.390	958
7%	0.990	0.407	1420
5%	5.615	0.453	5645
T25I10D10K			
40%	0.008	0.050	929
5%	1.464	0.218	10798
4%	2.562	0.240	19649
3%	6.485	0.551	47592
C20D10K			
90%	0.041	0.120	186
60%	0.211	0.210	186
40%	0.550	0.310	190
25%	4.443	2.714	260
C73D10K			
90%	12.406	4.328	1652
89%	21.070	10.964	1701
85%	52.679	14.685	1668

Table 1. Execution times for AprioriRare and Mr. Viper

these datasets to directly compare our results with the results found in their respective papers. The T25 and C73 datasets were additional datasets chosen to compare the effectiveness of *MR.VIPER* on multiple sparse datasets with different correlations.

T20 is a sparse, weakly correlated dataset with a sparsity of 97.77% with 893 distinct items. T25 is a sparse, weakly correlated dataset with a sparsity of 97.33% with 929 distinct items.

C20 is a slightly denser, highly correlated dataset with a sparsity of 89.58% with 192 distinct items. C73 is also a slightly denser, highly correlated dataset with a sparsity of 95.41% with 1592 distinct items.

The execution times of *MR.VIPER* and AprioriRare can be found in Table 1. The table lists the minimum support threshold, the execution time for both algorithms, and the number of minimal rare itemsets found.

For the T20 and C20 datasets, when provided with larger and larger minimum support thresholds, *MR.VIPER* begins to deteriorate in execution time. We believe this is because as the minimum support decreases, there will be more rare items. AprioriRare's linear approach to finding and checking these rare items for minimality is ultimately what results in it being slower than *MR.VIPER*.

For the C73 dataset, an interesting finding is that for all minimum support thresholds provided, *MR.VIPER* performed better than AprioriRare in all cases. We believe this is due to the number of distinct items in C73. Since candidate generation dominates the execution time, we believe that what led to this result was our hash structure implementation. Compared to AprioriRare's linear approach, the fast access times of the hash structure allowed for the time required to generate second-level candidates to be drastically reduced.

6. Conclusion and Future Work

In this paper, we proposed MR. VIPER as a foundation for mining minimal rare patterns in sparse, weakly correlated data. The experimentation results show that when using sparse datasets regardless of correlation strength, *MR.VIPER* will outperform AprioriRare when given low minimum support thresholds. When given larger minimum support thresholds, the search space covered by *MR.VIPER* begins to deteriorate to a similar search space as to AprioriRare, if not more.

An approach for optimization would be finding a way to abuse the fast access times of a Hash Structure for all levels of candidate generation instead of only just for the second level. We believe that being able to abuse a Hash Structure's access speed for all levels of candidate generation will be an important challenge to tackle in future research.

Another approach for optimization could be a different way of representing itemsets that allow for easy comparison of itemset prefixes. Our implementation represents items as an Integer, and performs a linear comparison to determine if two itemsets have the same prefix. A way to achieve this could be a bit vector representation of the itemset, allowing for fast and easy prefix comparison.

References

[1] C. C. Aggarwal. “Data Mining: The Textbook.” Heidelberg, Springer, 2015.

[2] J. Han, et al. “Data Mining: Concepts and Techniques.” Amsterdam, Elsevier, 2011.

[3] R. Agrawal, T. Imielinski, and A. Swami. “Mining association rules between sets of items in large databases.” In Proc. ACM SIGMOD 1993, pp. 207–216.

[4] R. Agrawal, and R. Srikanth. “Fast algorithms for mining association rules.” In Proc. VLDB 1994, pp. 487-499.

[5] G. Dong and J. Bailey. “Contrast Data Mining: Concepts, Algorithms, and Applications.” Chapman & Hall/CRC, 2012.

[6] R. Agrawal and R. Srikant. "Mining sequential patterns," In Proc. ICDE 1995, pp. 3-14.

[7] R. Srikant, & R. Agrawal. “Mining quantitative association rules in large relational tables.” In Proc. ACM SIGMOD 1996, pp. 1-12.

[8] Weiss, G. M. “Mining With Rarity: A Unifying Framework.” SIGKDD Explorations, 2004, Vol. 6, Issue 1, pp. 7 – 19.

[9] L. Szathmary, A. Napoli, P. Valtchev, “Towards Rare Itemset Mining.” In Proc. ICTAI 2007, pp. 305–312.

[10] L. Szathmary, P. Veltchev, A. Napoli, R. Godin. “Efficient Vertical Mining of Minimal Rare Itemsets.” In Proc. CLA 2012, pp. 269–280.

[11] M.J. Zaki et al. “New algorithms for fast discovery of association rules.” In Proc. KDD 1997, pp. 283–286.

[12] P. Shenoy et al. “Turbo-charging vertical mining of large databases.” In Proc. ACM SIGMOD 2000, pp. 22–33.