

Generic Itemset Mining Based on Reinforcement Learning

Kazuma Fujioka
kazuma.fujioka@kindai.ac.jp
Kindai University
Osaka, JAPAN

Kimiaki Shirahama
shirahama@info.kindai.ac.jp
Kindai University
Osaka, JAPAN

ABSTRACT

One of the biggest problems in itemset mining is the requirement of developing a data structure or algorithm, every time a user wants to extract a different type of itemsets. To overcome this, we propose a method, called *Generic Itemset Mining based on Reinforcement Learning* (GIM-RL), that offers a unified framework to train an agent for extracting any type of itemsets. In GIM-RL, the environment formulates iterative steps of extracting a target type of itemsets from a dataset. At each step, an agent performs an action to add or remove an item to or from the current itemset, and then obtains from the environment a reward that represents how relevant the itemset resulting from the action is to the target type. Through numerous trial-and-error steps where various rewards are obtained by diverse actions, the agent is trained to maximise cumulative rewards so that it acquires the optimal action policy for forming as many itemsets of the target type as possible. In this framework, an agent for extracting any type of itemsets can be trained as long as a reward suitable for the type can be defined. The extensive experiments on mining high utility itemsets, frequent itemsets and association rules show the general effectiveness and one remarkable potential (agent transfer) of GIM-RL. We hope that GIM-RL opens a new research direction towards learning-based itemset mining.

CCS CONCEPTS

• Information systems → Data mining: • Computing methodologies → Reinforcement learning.

KEYWORDS

itemset mining, reinforcement learning, high utility itemset, frequent itemset, association rule

1 INTRODUCTION

Much research effort has been made on itemset mining that aims to discover interesting relations among items in a large-scale dataset [8]. The dataset consists of a large number of transactions each of which contains different items. From such a dataset, the goal of itemset mining is to extract interesting sets of items (i.e., itemsets) in terms of a user-specified interestingness measure [12]. Depending on the user's needs, different interesting measures can be chosen to extract, for instance, *Frequent Itemsets* (FIs) consisting of frequently co-occurring items, *Association Rules* (ARs) comprised of correlated items, *High Utility Itemsets* (HUIs) formed by highly profitable items, and so on. From a more general perspective, items in a transaction can be viewed as attributes in one data instance like image, sensor recording, amino acid set. Thus, starting from

customer transaction analysis [1], itemset mining is utilised in various applications such as image classification, healthcare, bioinformatics and so on [8].

Itemset mining is difficult because of the huge search space. Assuming that there are m distinct items in a dataset, the search space is defined as the set of all the possible $2^m - 1$ itemsets. The naive approach to examine each of these itemsets is clearly infeasible. Hence, researchers have developed various data structures and algorithms to effectively prune the search space by considering a property specific to a target itemset type. The most popular one is the “downward closure property” for FIs, meaning that any subset of an FI must also be frequent [1, 8]. Apriori algorithm utilises this property to dramatically prune the search space by ignoring itemsets that contain one or more infrequent subsets of items [1]. In addition, the downward closure property is used to construct FP-tree (Frequent Pattern tree) that offers efficient, hierarchical organisation of items related only to FIs [14]. Also, considering the lack of the downward closure property for HUIs, upper bound utilities are defined so that the property is maintained among “potential” itemsets that have possibilities to be HUIs [2, 30]. These upper bound utilities are exploited to construct a tree structure that hierarchically maintains items related only to potential itemsets.

However, the above kind of itemset mining methods are inflexible because a specific data structure or algorithm is needed to extract a different type of itemsets. Especially, one itemset type is defined by an interestingness measure and there exist many such measures as listed in [12]. It is clearly impractical to develop a data structure or algorithm for each of these itemset types. In other words, diverse types of itemsets remain undiscovered because neither data structure nor algorithm is available. Therefore, one crucial issue in itemset mining is the development of a unified framework to extract different types of itemsets.

For this issue, we propose *Generic Itemset Mining based on Reinforcement Learning* (GIM-RL). The main idea comes from how human searches for a target type of itemsets in a dataset. Most probably, human begins with briefly going through the dataset to get rough knowledge about which items seem to be important (or unimportant) for the target type, which items are related to each other and so on. Then, he/she makes a “plausible itemset” consisting of some important items, and examines whether it matches the target type or not. Afterwards, based on knowledge obtained in the past search experiences, human edits the plausible itemset by adding or removing items to create a new plausible itemset. One important point is that human adaptively changes an approach to make plausible itemsets depending on a target type as well as past experiences. Hence, human's itemset search described above is considered a key for devising a unified itemset mining framework.

We focus on *Reinforcement Learning* (RL) that is rooted in behavioural psychology and provides a framework where an artificial agent interacts with an uncertain environment to adaptively acquire the optimal policy of sequential decision-making [11, 22, 23]. The agent takes an action at each time step and the environment produces a reward as the response to the action. This trial-and-error step is repeated numerous times to accumulate rewards obtained by various actions at diverse states of the environment. Thereby, RL attempts to find the optimal policy that enables the agent to decide a sequence of actions which maximise cumulative rewards.

RL is utilised in GIM-RL to train an agent for extracting a target type of itemsets from a dataset in the following way: First, the agent performs an action to update an itemset by adding or removing an item. Then, the environment characterised by the dataset generates a reward that expresses how relevant the itemset updated by the agent is to the target type. GIM-RL collects a large number of trial-and-error steps in which the agent sometimes succeeded or sometimes failed in forming itemsets of the target type. By analysing these trial-and-error steps, the agent is trained to have the optimal policy for adding or removing items to form itemsets of the target type. Noted that GIM-RL outlined above can extract any type of itemsets as long as one can define a reward that appropriately represents the relevance of an itemset to the type. We demonstrate that GIM-RL can be generally applied to mining of HUIs [2, 19, 26, 30], FIs [1, 8, 14] and ARs [1, 8, 9].

One big by-product of GIM-RL is that it outputs not only extracted itemsets but also a trained agent. On the other hand, most of existing methods leave nothing behind except extracted itemsets. In other words, GIM-RL retains knowledge obtained in the mining process as the trained agent, while it is quite wasteful that most of existing methods throw such knowledge out. With respect to this, there exist many cases where the same mining process is performed on multiple related datasets. For example, one may want to inspect trend changes by extracting HUIs from two datasets collected in different terms. It is reasonable to use knowledge obtained from one of these datasets to speed up the mining process on the other dataset. Moreover, if the latter dataset is too huge to perform itemset mining from scratch, knowledge from the former dataset may be useful for accomplishing adequate mining on the latter one.

Based on the above consideration, we investigate GIM-RL’s novel potential called *agent transfer*. This means that an agent is firstly trained on a source dataset, and then it is transferred to another compatible agent for a target dataset. If the source and target datasets are related to each other, they are thought to be characterised by similar relations among items. Thus, the agent for the source dataset is expected to be useful also for the target dataset. That is, the transferred agent only needs fine-tuning and offers more efficient itemset mining on the target dataset, compared to an agent trained from scratch. Unlike itemset mining, this kind of “transfer learning” is popular in different application domains where a model pre-trained on a source dataset is transferred to another model designed for a target dataset [18, 24]. We bring transfer learning into itemset mining as agent transfer, and show its possibility to significantly accelerate itemset mining.

2 RELATED WORK

Existing itemset mining methods are roughly divided into two categories, *exhaustive* and *non-exhaustive*. The former includes methods that enumerate all the itemsets matching a target type. However, the runtime of exhaustive methods significantly degrades as a dataset enlarges, especially, the increase in the number of distinct items causes the exponential expansion of the search space. To overcome this, non-exhaustive methods generate an approximate set of itemsets matching the target type. Below we first review several existing methods in the exhaustive and non-exhaustive categories, and then clarify the advantages of GIM-RL compared to those methods.

In general, an exhaustive method uses a data structure or algorithm specialised to a target type. For example, considering the downward closure property for FIs, Apriori algorithm [1] and FP-growth algorithm based on FP-tree [14] are developed for efficient enumeration of FIs. Apart from FIs, a divide-and-conquer approach based on a bitwise vertical representation of a dataset is developed to extract closed FIs, each of which has no superset supported by the same set of transactions [20]. The method in [31] enumerates maximal FIs that are included in no other FI, by devising a depth-first itemset expansion and dataset reductions. For efficient extraction of weighted FIs consisting of items associated with high weights, FP-growth is extended by crafting upper bound weights, three pruning techniques and a parallel mining algorithm [15]. Infrequent weighted itemsets consisting of rare and lowly weighted items are extracted by revising FP-growth with a specialised interestingness measure and a technique of early discarding unpromising items [5] (please see [16] for existing infrequent itemset mining methods). An HUI is an extension of a weighted FI in the sense that its utility is computed by considering both the weight and quantity of each item in a transaction. To efficiently extract all the HUIs, researchers have developed a list that facilitates the expansion of an itemset and the corresponding utility calculation [19], and a tree structure based on the downward closure property of upper bound utilities [2, 30].

Exhaustive methods reviewed above are inflexible because a different data structure or algorithm is needed to extract a different type of itemsets. In contrast, GIM-RL can extract diverse types of itemsets only by defining a reward suitable for each type. By referring to the reward definitions for HUIs and FIs in Sections 3.2.1 and 3.2.2, one can easily design rewards for weighted FIs and infrequent weighted itemsets. In addition, it is expected that most of maximal FIs can be extracted using a reward, which is computed by checking the frequency of an itemset and the existence of its supersets in the set of already explored itemsets. This reward leads an agent to form itemsets that not only are frequent but also include more items. Closed FIs are likely to be extracted using a similar reward. Furthermore, many types of itemsets like ARs are divided into the antecedent and consequent, and let us assume the extraction of itemsets characterised by one item in the consequent. Under this setting, by defining a reward similar to the one in Section 3.2.3, GIM-RL is expected to train an agent that can extract itemsets matching each of 38 interestingness measures listed in [12].

One of the most popular non-exhaustive itemset mining approaches is *Evolutionary Computation* (EC) that offers a metaheuristic where

nature-inspired operators are iteratively used to update itemsets into better ones in terms of a fitness function [29]. One feature of EC-based methods is their predictable runtimes because itemset extraction is terminated by a specified number of iterations and no complicated process is needed for updating itemsets. One main class of EC-based methods is characterised by *Genetic Algorithm* (GA) that iteratively selects promising itemsets using a fitness function, and exploits them to create new itemsets based on crossover and mutation operators [21, 27]. Another main class is based on *swarm intelligence-based algorithms* that iteratively update itemsets based on operations inspired by the collective behaviours of swarms like ants, bees and bats [26, 29].

Since there is no guarantee that GIM-RL can extract all the itemsets of a target type, it is classified as a non-exhaustive method and has a high similarity to EC-based methods. This is because a reward in GIM-RL corresponds to a fitness function in EC-based methods, and any type of itemsets can be extracted by defining a suitable fitness function for the type. However, the biggest difference is that each EC-based method relies on a heuristically pre-defined strategy (i.e., metaheuristic) to extract itemsets, while GIM-RL analyses a dataset and learns such a strategy as a trained agent. In other words, the former only outputs extracted itemsets, whereas GIM-RL produces those itemsets as well as the trained agent that captures generalised characteristics of items in the dataset and can be transferred for another similar dataset.

Another popular approach to non-exhaustive itemset mining is *pattern sampling* that approximates a probability distribution over the search space by associating each itemset with a probability, which is proportional to its relevance to a target type in terms of an interestingness measure [4, 7]. Thus, itemsets sampled according to this probability distribution are a representative subset of itemsets matching the target type. However, only a limited number of itemset types can be treated by pattern sampling because of their compatibilities with sampling algorithms, for instance, an itemset type needs to be represented by a specified weight function form [4] or by a combination of XOR constraints [7]. GIM-RL can extract a much more variety of itemsets. In addition, no consideration is given on whether a probability distribution approximated for a source dataset can be transferred to the one for a target dataset. For this, GIM-RL offers a very flexible agent transfer in which an agent can be transferred between the source and target datasets even if they are characterised by different sets of distinct items.

Agent transfer is related to incremental itemset mining where a dataset is updated by adding, deleting and modifying transactions [2, 8, 17], and itemset mining in a stream where transactions arrive in rapid succession [6, 8]. While these two tasks treat datasets that change over time, source and target datasets for agent transfer are fixed before the itemset extraction. As another substantial difference, the above two tasks focus on data structures to efficiently manage or summarise information necessary for extracting itemsets [2, 6, 17]. In agent transfer, such information is held by an agent (neural network) that captures generalised relations among items in the source or target dataset.

To our best knowledge, GIM-RL is the first method that adopts RL to train a machine learning model (i.e., agent defined by a neural network) for itemset mining. Correspondingly, agent transfer is

not explored in any existing work. As discussed in Section 4.3, one problem of GIM-RL is its slow runtime because it needs multiple scans over a dataset to compute rewards and states of an environment. Of course, fast reward/state computation is one important future work. But, we believe that the efficiency of GIM-RL cannot be measured only by its runtime. The reason is that, according to a user-defined reward for a target type, GIM-RL adaptively trains an agent that can extract itemsets of this type, although one or more days may be needed. This seems much more efficient compared to a case where someone spends one or more months to implement a specialised data structure or algorithm for the target type.

3 GIM-RL

In this section, we first describe GIM-RL’s general framework that can be commonly used to extract various types of itemsets. Then, we provide specific reward definitions to extract HUIs, FIs and ARs.

3.1 General Framework

Let $\mathcal{D} = \{T_1, \dots, T_N\}$ be a dataset containing N transactions, and $\mathcal{I} = \{i_1, \dots, i_M\}$ be a set of M distinct items each of which is included in at least one transaction in \mathcal{D} . Each transaction T_n ($1 \leq n \leq N$) in \mathcal{D} is a subset of \mathcal{I} (i.e., $T_n \subseteq \mathcal{I}$). Letting $|T_n|$ denote the number of items in T_n , we describe $T_n = \{i_{n,1}, \dots, i_{n,|T_n|}\}$ where $i_{n,l}$ ($1 \leq l \leq |T_n|$) is the l th item in T_n (i.e., $i_{n,l} \in \mathcal{I}$). We denote by X an itemset consisting of items in \mathcal{I} (i.e., $X \subseteq \mathcal{I}$). Let us consider an interestingness measure $\varphi(X)$ that takes X as input, analyses transactions in \mathcal{D} and outputs a value expressing the relevance of X to a target type. For example, $\varphi(X)$ for FIs returns the support (frequency) of X in \mathcal{D} and $\varphi(X)$ for HUIs outputs X ’s utility. In addition, $\varphi(X)$ can be flexibly used for an itemset type requiring multiple conditions. For instance, to extract ARs by considering the support and confidence of X , one can design $\varphi(X)$ that outputs a value depending on whether X meets either or both of support and confidence thresholds. Under the setting described above, the goal of itemset mining is to extract from \mathcal{D} every itemset X for which $\varphi(X)$ is larger than a pre-defined threshold ξ (i.e., $\varphi(X) \geq \xi$).

We formulate itemset mining as an RL problem shown in Fig. 1. It is assumed that the environment signifies one step of the mining process. Specifically, the k th step is performed to modify the itemset defined by a bit-vector b_k into a new one that is likely to match a target type. Formally, $b_k = (b_{k,1}, \dots, b_{k,M})^T$ is an M -dimensional binary vector where $b_{k,m} \in \{0, 1\}$ ($1 \leq m \leq M$) represents the inclusion of the m th item i_m in the itemset, namely, $b_{k,m} = 1$ indicates i_m is included, otherwise not-included. We express the itemset defined by b_k as $X(b_k)$. The environment produces a state $s_k = (s_{k,1}, \dots, s_{k,M})^T$ having the same dimensionality to b_k . Here, $s_{k,m}$ ($1 \leq m \leq M$) exhibits how useful it is to change $b_{k,m}$ for forming an itemset of the target type. In our implementation, $s_{k,m}$ is computed based on the simulation of the one-step-ahead future, in which $b_{k,m}$ is virtually changed to create the bit-vector b'_k and the corresponding itemset $X(b'_k)$. That is, $X(b_k)$ and $X(b'_k)$ differ only in the inclusion of i_m . Then, $s_{k,m}$ is calculated as $\varphi(X(b'_k))$ by applying an interesting measure $\varphi(X)$ to $X(b'_k)$. Since the computation of s_k needs to check transactions in \mathcal{D} , s_k in Fig. 1 is connected with the non-filled arrow from \mathcal{D} .

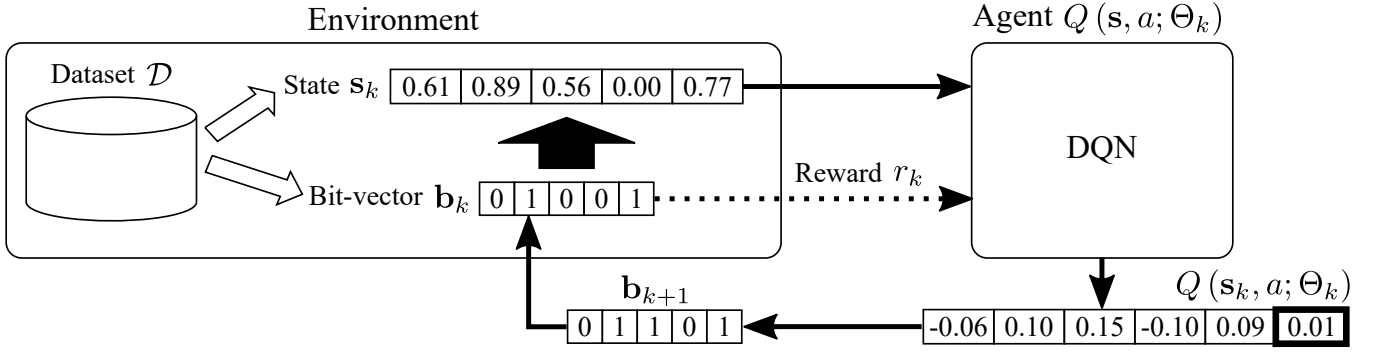


Figure 1: An overview of GIM-RL.

s_k shows hints about which items are likely to be included or excluded to form an itemset of the target type. Thus, as illustrated by the solid arrow from s_k to the agent in Fig. 1, s_k is fed into the agent to decide an action of changing one value in b_k . As a result, b_k is updated into b_{k+1} , meaning that the new itemset $X(b_{k+1})$ is created by adding or removing one item to or from $X(b_k)$. The agent’s action is designed based on human’s itemset search where he/she modifies an itemset by thinking about which item’s inclusion or exclusion is crucial for the target type. Then, the environment generates a reward r_k that has a close relation to $\varphi(X(b_{k+1}))$. Like s_k , the computation of r_k needs to check occurrences of $X(b_{k+1})$ in \mathcal{D} , so a non-filled arrow is placed between the bit-vector (that is now b_{k+1}) and \mathcal{D} in Fig. 1. As depicted by the dashed arrow in Fig. 1, the agent receives r_k as an evaluation score for the action taken at the k th step. If $X(b_{k+1})$ matches the target type, the agent obtains high r_k . Afterwards, the environment creates a new state s_{k+1} based on which the agent updates b_{k+1} into b_{k+2} and receives r_{k+1} . This way, the agent iteratively updates the bit-vector to form various itemsets that possibly match the target type. In this framework, our goal is to train the agent that maximises cumulative rewards over steps. This means that the agent can extract as many itemsets matching the target type as possible.

It should be noted that s_k only provides the information for selecting one action, and does not include information for determining a sequence of multiple actions. In other words, the target type of itemsets are not necessarily extracted by greedily changing b_k ’s value that corresponds to s_k ’s highest value. Instead, the agent needs to have an intelligent policy to select an action at the current step by considering what kind of itemsets will be obtained at the future steps. Intuitively, we aim to train an agent that takes actions causing itemsets of non-target types at some consecutive steps, in order to extract many itemsets of the target type after those steps.

To this end, we employ *Q-learning* that trains an agent characterised by a *Q function* which takes as input an action a and a state s of the environment, and outputs a ’s quality at s [11, 22, 23]. Ideally, the optimal *Q function* $Q^*(s, a)$ quantifies a ’s quality as follows:

$$Q^*(s, a) = \max_{\pi} \mathbb{E} \left[\sum_{k'=k}^K \gamma^{k'-k} r_{k'} \mid \tilde{s}_k = s, \tilde{a}_k = a, \pi \right], \quad (1)$$

where K is the final step of the mining process, γ is the discount factor by which a reward at a further future step is discounted more strongly, and π represents a policy for action selection. Eq. 1 means that a ’s quality at s is computed as the maximum expected cumulative reward, which is achievable after seeing s at the k th step and taking a . Here, s and a are assigned to the variables \tilde{s}_k and \tilde{a}_k for representing a state and an action at the k th step, respectively. Simply speaking, in itemset mining, $Q^*(s, a)$ indicates the maximum number of itemsets of the target type after taking a . However, it is impractical to directly build $Q^*(s, a)$ because of the huge number of state-action combinations. Thus, considering the recent success of deep Q-learning [11, 22, 23], we approximate $Q^*(s, a)$ by a neural network, called *Deep Q-Netwrok* (DQN), defined by a set of parameters Θ . Therefore, $Q^*(s, a)$ is parametrised as $Q(s, a; \Theta)$, and our goal is to optimise Θ of the DQN so that actions selected based on $Q(s, a; \Theta)$ form many itemsets of the target type. Note that we interchangeably use the terms “agent” and “DQN” and the notation “ $Q(s, a; \Theta)$ ” in the following discussions.

Let \mathcal{A} be the set of possible actions. In Fig. 1, $Q(s, a; \Theta)$ is used to compute $Q(s_k, a; \Theta)$ that approximates the quality of each action $a \in \mathcal{A}$ at the specific state s_k of the k th step. Then, the action a_k is chosen as the one corresponding to the highest value of $Q(s_k, a; \Theta)$. With no deep elaboration, one may define \mathcal{A} to include M actions each of which changes $b_{k,m}$ in b_k (i.e., $|\mathcal{A}| = M$). But, as depicted by the bold-lined rectangle at the bottom-right of Fig. 1, in order to help the agent explore diverse itemsets, we add to \mathcal{A} one more action that randomly initialises the bit-vector. This means that the agent is given an opportunity to stop updating b_k and re-start itemset mining from the initialised bit-vector. As an implementation detail, the random bit-vector initialisation is performed based on the probability distribution where the probability of $b_{k,m} = 1$ is proportional to the frequency of the m th item i_m in \mathcal{D} , and is repeated until the bit-vector defines an itemset existing in \mathcal{D} . Conversely, it is meaningless to examine itemsets not-existing in \mathcal{D} .

According to Bellman equation, the optimisation of Θ is done by making $Q(s_k, a_k; \Theta)$ and $r_k + \gamma \max_{a'} Q(s_{k+1}, a'; \Theta)$ as close as possible [11, 22, 23]. An intuitive interpretation is that if $Q(s, a; \Theta)$ is a good approximation of $Q^*(s, a)$, the maximum expected cumulative reward estimated for taking a_k at s_k should be equal or very similar to the sum of r_k obtained by taking a_k at s_k with the

maximum expected cumulative reward estimated for the action at the next state s_{k+1} . However, the following two issues make it unstable to optimise Θ by directly using $Q(s_k, a_k; \Theta)$ and $r_k + \gamma \max_{a'} Q(s_{k+1}, a'; \Theta)$. First, the agent’s experiences (s_k, a_k, r_k, s_{k+1}) s obtained at consecutive steps are very similar because only one value in b_k is changed at each step except its random initialisation. As a result, $Q(s_k, a_k; \Theta)$ trained on those experiences is very biased. To overcome this, we use a *replay memory* \mathcal{P} that is a queue to store the recent $|\mathcal{P}|$ experiences, and optimise Θ on experiences randomly sampled from \mathcal{P} [22, 23]. Second, Θ is included in the “target value $r_k + \gamma \max_{a'} Q(s_{k+1}, a'; \Theta)$ ” that $Q(s_k, a_k; \Theta)$ targets to approximate. That is, the target value changes every time Θ is updated, which makes the optimisation of Θ unstable. To address this, separate DQNs are used for $Q(s_k, a_k; \Theta)$ and the target value [22, 23]. The parameters Θ of the DQN $Q(s_k, a_k; \Theta)$ are updated every step. On the other hand, the DQN $Q(s_k, a_k; \Theta^-)$ used for the target value is called *target network* and characterised by the parameters Θ^- that are periodically updated every a certain number of steps. To sum up, as shown in the following equation, $Q(s, a; \Theta)$ is trained by minimising the squared difference between $Q(s_k, a_k; \Theta)$ and the target value $r_k + \gamma \max_{a'} Q(s_{k+1}, a'; \Theta^-)$ on experiences randomly sampled from \mathcal{P} :

$$\mathbb{E}_{(s_k, a_k, r_k, s_{k+1}) \sim \mathcal{P}} \left[\left(r_k + \gamma \max_{a'} Q(s_{k+1}, a'; \Theta^-) - Q(s_k, a_k; \Theta) \right)^2 \right] \quad (2)$$

Finally, GIM-RL to train the agent $Q(s, a; \Theta)$ is summarised in Algorithm 1 of Section A. Overall, GIM-RL executes E episodes each of which consists of K steps to form different itemsets through updating the bit-vector.

3.2 Reward Definition

We explain our definitions of rewards used to extract HUIs, FIs and ARs. Of course, it is possible to define better rewards than the ones below. All the rewards used in this paper are defined based on the common scheme that deals with the following four cases:

Case 1: The itemset $X(b_{k+1})$ resulting from the action a_k at the state s_k does not exist in \mathcal{D} . The agent receives a reward of -1 , so that it is guided to not perform a_k at s_k as well as at a similar state.

Case 2: The interestingness measure value $\varphi(X(b_{k+1}))$ is less than the quarter of a pre-specified threshold ξ (i.e., $\varphi(X(b_{k+1})) < \xi/4$) and the agent receives a reward of 0. This means that taking a_k at s_k or a similar state has no important impact on forming the target type of itemsets.

Case 3: $\varphi(X(b_{k+1})) \geq \xi/4$ is the condition of this case, which is further divided into the four sub-cases, $\xi/4 \leq \varphi(X(b_{k+1})) < \xi/2$, $\xi/2 \leq \varphi(X(b_{k+1})) < 3\xi/4$, $3\xi/4 \leq \varphi(X(b_{k+1})) < \xi$ and $\xi \leq \varphi(X(b_{k+1}))$, in which the agent receives rewards of 1, 2, 3 and 4, respectively. These rewards are defined according to how close $X(b_{k+1})$ is to the target type. Thereby, the agent is informed of how important it is to take a_k at s_k or a similar state for forming itemsets of the target type. The reason why the last sub-case $\xi \leq \varphi(X(b_{k+1}))$ is included is explained below.

Case 4: This case is defined by two conditions. The first is that $X(b_{k+1})$ matches the target type (i.e., $\xi \leq \varphi(X(b_{k+1}))$), and the second is that $X(b_{k+1})$ has not yet been extracted in the current episode. Thus, $X(b_{k+1})$ is a newly extracted itemset of the target

type, and we aim to extract such itemsets. Thus, by providing a very high reward of 100, the agent is led to extract many unique itemsets of the target type. Also, if $X(b_{k+1})$ is an already extracted itemset, it seems unreasonable to regard a_k as meaningless because $X(b_{k+1})$ anyway matches the target type. This situation is captured by the last sub-case in Case 3, and the agent gets a reward (4) that is much smaller than the one (100) in this case.

To complete the above-mentioned scheme, we describe the computation of $\varphi(X(b_{k+1}))$ for each of HUI, FI and AR.

3.2.1 Reward for HUI Extraction. For the n th transaction $T_n = \{i_{n,1}, \dots, i_{n,|T|}\}$ in \mathcal{D} , the l th item $i_{n,l}$ ($1 \leq l \leq |T_n|$) is associated with the item-specific utility $p(i_{n,l})$ and the quantity $q(i_{n,l})$ in T_n . Under this setting, $\varphi(X(b_{k+1}))$ representing the utility of $X(b_{k+1})$ is computed as follows [2, 19, 26, 30]:

$$\varphi(X(b_{k+1})) = \sum_{X(b_{k+1}) \subseteq T_n \wedge T_n \in \mathcal{D}} \sum_{i_{n,l} \in X(b_{k+1}) \wedge i_{n,l} \in T_n} p(i_{n,l}) q(i_{n,l}) \quad (3)$$

The inner summation refers to each transaction containing $X(b_{k+1})$ as T_n , and computes the sum of item-specific utilities weighted by their corresponding quantities for all items in $X(b_{k+1})$. As signified by the outer summation, $\varphi(X(b_{k+1}))$ is calculated as the total of such weighted sums for all transactions containing $X(b_{k+1})$.

3.2.2 Reward for FI Extraction. $\varphi(X(b_{k+1}))$ is defined as $\text{sup}(X(b_{k+1}))$ representing the support (frequency) of $X(b_{k+1})$ in \mathcal{D} , that is, the number of transactions containing $X(b_{k+1})$.

3.2.3 Reward for AR Extraction. To define an AR, an itemset X is firstly divided into two subsets X_a and X_c (i.e., $X = X_a \cup X_c$), and a rule $X_a \rightarrow X_c$ is created by viewing X_a and X_c as the antecedent and consequent, respectively. $X_a \rightarrow X_c$ is regarded as an AR if $\text{sup}(X)$ is larger than the minimum support threshold min_sup and the confidence, which is the conditional probability of X_c given X_a (i.e., $\text{sup}(X)/\text{sup}(X_a)$), is larger than the minimum confidence threshold min_conf . GIM-RL extracts ARs using a reward that jointly considers the support and confidence conditions.

As one remark, the current GIM-RL is limited to extracting ARs each of which is characterised by X_c with one item. But, focusing only on such ARs is considered reasonable because allowing X_c to include multiple items often produces an extremely large number of ARs. In addition, several existing methods address the extraction of ARs with X_c s characterised by single items [28, 32]. Moreover, assuming that human is paying attention to one itemset, he/she is likely to think whether a rule is created by regarding the itemset as X_a and a new item as X_c , or by seeing an item in the itemset as X_c and the other items as X_a . GIM-RL implements this human’s search based on the difference between $X(b_k)$ and $X(b_{k+1})$. If $X(b_{k+1})$ is the itemset created by adding one item to $X(b_k)$, we consider that the agent takes the action to use $X(b_k)$ and the added item as X_a and X_c , respectively. For an inverse case where $X(b_{k+1})$ is the itemset created by removing one item from $X(b_k)$, the agent’s action is to use $X(b_{k+1})$ and the removed item as X_a and X_c , respectively.

Such an agent’s action to decide X_a and X_c that may constitute an AR, is evaluated by a reward based on the following two $\varphi(X(b_{k+1}))$ s. The first one $\varphi_1(X(b_{k+1}))$ is used in Cases 2 and 3 to check the support of $X_a \cup X_c$ which is the itemset including more

items between $X(b_k)$ and $X(b_{k+1})$. If $\varphi_1(X(b_{k+1})) = \sup(X_a \cup X_c)$ is equal to or larger than $\xi_1/4 = \min_sup/4$, the agent gets a reward of 1, 2, 3 or 4. This is because these X_a and X_c are close to constituting an AR in terms of the support condition. The second $\varphi(X(b_{k+1}))$ denoted by $\varphi_2(X(b_{k+1}))$ is used in Case 4, and checks the confidence of $X_a \rightarrow X_c$, that is, $\varphi_2(X(b_{k+1})) = \sup(X_a \cup X_c)/\sup(X_a)$. Note that Case 4 is triggered only when Case 3 is passed by $\min_sup \leq \varphi_1(X(b_{k+1}))$. Thus, if $\xi_2 = \min_conf \leq \varphi_2(X(b_{k+1}))$, X_a and X_c obtained by the agent action are verified as the AR $X_a \rightarrow X_c$ and a reward of 100 is produced. Otherwise, the agent gets a reward of 4 resulting from Case 3.

Finally, it is possible to extract ARs that are individually characterised by X_c with multiple items, by allowing an agent to take actions for changing multiple values in the bit-vector. However, this causes an exponential increase of the agent’s action space, so a smart approach seems needed. This issue is left as our future work.

4 EXPERIMENTAL RESULTS

We test GIM-RL using the five datasets shown in Table 1. These datasets are chosen because they have different characteristics in terms of numbers of transactions, numbers of distinct items and average numbers of items in one transaction, as respectively exhibited in rows N , M and “Avg. $|T_n|$ ” in Table 1. In addition, the datasets in Table 1 are popularly used in many existing works [4, 7, 9, 26, 27]. Through the extractions of HUIs, FIs and ARs from these datasets, we aim to demonstrate the generality of GIM-RL.

Table 1: Statistics about the experimental datasets.

	Chess	Mushroom	Accidents_10%	Connect	Pumsb
N	3196	8416	34018	67557	49046
M	75	119	468	129	2113
Avg. $ T_n $	37	23	34	43	74

4.1 Results for Itemset Mining

For each of the HUI, FI and AR extractions, the following two evaluations are performed: First, we use a baseline method that is developed to exhaustively enumerate all the itemsets of a target type. The number of itemsets extracted by the baseline method is maximum, so we examine how close the number of itemsets extracted by GIM-RL is to this maximum. Second, the effectiveness of agents trained by GIM-RL is examined by comparing the five agents below. Note that these agents commonly take s_k as input and output a_k to change the value of one dimension in b_k in Fig. 1, but they are different in how to decide a_k based on s_k .

Random: This agent just changes the value of a randomly selected dimension in b_k . With *Random*, we aim to show the difficulty of each itemset extraction task where randomly changing values in b_k yields few itemsets of a target type.

State- ϵ : As described in Section 3.1, each dimension’s value in s_k represents the usefulness of changing the value of the corresponding dimension in b_k based on the simulation of the one-step-ahead future. One may think that s_k already contains enough information about which value in b_k should be changed. To answer this, we test

State- ϵ that changes the value of b_k ’s dimension corresponding to the highest value in s_k with the probability $1 - \epsilon$, while changing the value of a randomly selected dimension with the probability ϵ to preserve the variety of itemsets to be explored.

State-prob: The motivation for testing this agent is the same to the one of *State- ϵ* . But, different from *State- ϵ* , *State-prob* determines a_k by random sampling according to the probability distribution, where the probability of changing the value of one dimension in b_k is proportional to the value of the corresponding dimension in s_k .

GIM-RL-State: This is the basic version of GIM-RL. A DQN that accepts s_k and outputs $Q(s_k, a; \Theta)$ is trained in the framework of Fig. 1. DQNs with the same architecture are trained for extracting HUIs and ARs, while simpler DQNs are used for FIs. Please see Section B for more details of these DQNs.

GIM-RL-Fusion: This is an extended version of *GIM-RL-State* by defining the output as the sum of $Q(s_k, a; \Theta)$ and s_k . To be precise, s_k misses the dimension for the random bit-vector initialisation described in Section 3.1, so s'_k is created by appending to s_k one dimension with the very small value “ $0.02 \times (\text{average of } s_k)$ ” for the initialisation. Then, the output of *GIM-RL-Fusion* is computed as $Q'(s_k, a; \Theta) = \lambda s'_k + (1 - \lambda)Q(s_k, a; \Theta)$. Of course, the additional use of s'_k is expected to make $Q'(s_k, a; \Theta)$ attain better action selection than $Q(s_k, a; \Theta)$. But, $Q'(s_k, a; \Theta)$ plays a more important role to advance training of an agent. In principle, the agent can be trained when positive rewards are obtained. In other words, training of the agent does not proceed as long as rewards are zero or negative. Regarding this, at the beginning of training, it is statistically difficult to find “positive itemsets” that offer positive rewards only by the imperfect agent $Q(s_k, a; \Theta)$. For this, s'_k significantly increases the probability of finding positive itemsets because it represents rough estimation of which items are likely to constitute positive itemsets. After finding some positive itemsets, the agent can be trained to some extent. This boosts the probability that the agent can find positive itemsets by its own exploitation based on the trained $Q(s_k, a; \Theta)$, that is, the agent can do further training by itself. Considering this, λ in $Q'(s_k, a; \Theta)$ is gradually reduced to weaken the effect of s'_k as training proceeds. Please see Section C for the specific setting of λ .

Results for HUI Mining: Table 2 summarises the results for HUI mining. First, HUI-Miner [19] implemented in SPMF library [10] is used as a baseline to extract all the HUIs from each dataset. The thresholds in the second row of Table 2 are the same to the ones used in [26, 27]¹. As shown in Table 2, 100% or nearly 100% of HUIs are extracted from each of the four datasets using *GIM-RL-Fusion*. This verifies the effectiveness of GIM-RL to train agents for extracting HUIs. Also, the fact that no HUI is extracted by *Random* implies the difficulty of HUI extraction. In addition, the poor performances of *State- ϵ* and *State-prob* indicate the ineffectiveness of directly using s_k . In what follows, *Random*, *State- ϵ* and *State-prob* are sometimes called “non-training agents” because their action selection approaches are fixed in advance and are not optimised. Finally, the main reason why no HUI is extracted by *GIM-RL-State* for Accidents_10% is that it fails to find first some positive itemsets,

¹Also for the other threshold settings used in [26, 27], we have obtained results to validate the effectiveness of GIM-RL.

so its training does not proceed. In contrast, *GIM-RL-Fusion* aided by s'_k stably produces very good performances on all the datasets.

Table 2: An overview of the HUI mining results.

	Chess	Mushroom	Accidents_10%	Connect
ξ (%)	29.0%	14.5%	13.0%	32.0%
Baseline	176	199	127	171
Random	0	0	0	0
State- ϵ	56	82	51	67
State- $prob$	0	0	0	0
<i>GIM-RL-State</i>	176	199	0	169
<i>GIM-RL-Fusion</i>	176 (100%)	197 (98.9%)	127 (100%)	169 (98.8%)

Results for FI Mining: Table 3 shows the results for FI mining. FP-growth [14] implemented in SPMF library [10] is selected as a baseline to extract all the FIs. By referring to [9], the thresholds in the second row are determined so that moderate numbers of FIs (1000-10000 FIs) are extracted. Note that FIs are extracted by the non-training agents, although numbers of extracted FIs significantly vary depending on datasets. For this, we point out the following two reasons attributed to the random bit-vector initialisation. First, a bit-vector is initialised according to the probability distribution based on the frequency of each item, so the itemset defined by this bit-vector has a relatively high probability to be an FI. In addition, updating this bit-vector leads to find other FIs with not-low probabilities. Second, the initialisation is repeated until the bit-vector defines an itemset that exists in \mathcal{D} . Focusing only on such itemsets dramatically reduces the search space, especially for small datasets like Chess. Because of the above two reasons, even *Random* can extract FIs. But, despite the fact that the non-training agents occasionally extract many FIs, *GIM-RL-Fusion* or *GIM-RL-State* extracts the highest numbers of FIs for all the datasets. This validates the effectiveness of GIM-RL also for extracting FIs².

Table 3: An overview of the FI mining results.

	Chess	Mushroom	Pumsb	Connect
ξ (%)	80%	35%	90%	95%
Baseline	8227	1121	2607	2201
Random	2258	63	304	1541
State- ϵ	587	440	427	394
State- $prob$	4926	502	955	2144
<i>GIM-RL-State</i>	6154	1101	2544	2072
<i>GIM-RL-Fusion</i>	6843 (83.2%)	969 (86.4%)	2578 (98.8%)	2199 (99.9%)

Results for AR Mining: In Table 4, the results for AR mining are presented. Our baseline method first uses FPGrowth_association_rules implemented in SPMF library [10] to extract all the ARs, and then retains ARs each of which is characterised by a consequence with one item. The thresholds $\xi_1 = min_sup$ and $\xi_2 = min_conf$

²For Chess, 7901 FIs (96.0% of FIs) are extracted by *GIM-RL-Fusion* using a DQN with the architecture used for the HUI and AR extractions.

are set based on [9], so that a reasonable number of ARs are extracted from each dataset. The trend of the results in Table 4 is similar to the one in Table 3. Because of the random bit-vector initialisation, the non-training agents can extract ARs from each dataset, but their performances significantly degrade on large datasets like Pumsb and Connect. In contrast, *GIM-RL-Fusion* can stably extract almost all ARs from each of the datasets.

Table 4: An overview of the AR mining results.

	Chess	Mushroom	Pumsb	Connect
ξ_1 (%)	90%	50%	90%	95%
ξ_2 (%)	80%	80%	80%	80%
Baseline	2351	331	11366	10106
Random	2214	330	120	1287
State- ϵ	639	115	640	615
State- $prob$	2350	331	481	3545
<i>GIM-RL-State</i>	2202	331	4832	4026
<i>GIM-RL-Fusion</i>	2340 (99.5%)	330 (99.6%)	10111 (88.9%)	9225 (91.2%)

We describe a deeper insight in the AR extraction on Connect. Fig. 2 shows how many ARs are extracted in each episode by *Random*, *State- $prob$* , *GIM-RL-State* and *GIM-RL-Fusion*. *State- ϵ* is omitted because of the very low number of extracted ARs. In Fig. 2, for each agent, the number of extracted ARs in one episode is counted without considering whether each AR is already extracted or not. Thus, the sum of numbers of ARs extracted by the agent over all the episodes is the total cumulative number of ARs. As shown in Fig. 2, the numbers of ARs extracted by the non-training agents are constantly low over episodes, and they are nearly invisible. On the other hand, *GIM-RL-State* is trained to extract ARs during about the first 50 episodes, and afterwards keeps extracting ARs although the extraction is unstable as illustrated by the significantly varied numbers of ARs over episodes. Compared to this, *GIM-RL-Fusion* aided by s'_k is trained to extract ARs in the first 200 episodes, and continues to stably extract the large numbers of ARs.

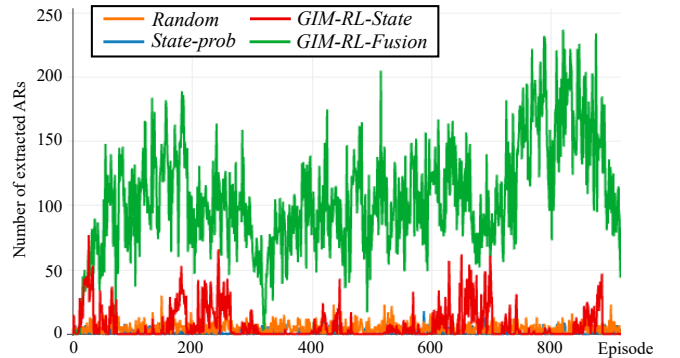


Figure 2: The transition of numbers of ARs extracted by each of *Random*, *State- $prob$* , *GIM-RL-State* and *GIM-RL-Fusion* over the passage of episodes (the dataset is Connect).

4.2 Results for Agent Transfer

We investigate agent transfer where an agent trained on a source dataset is transferred into another agent for a target dataset which is related to the source one. For this purpose, each of the datasets in Table 1 is split into two parts, especially, the first 60% of transactions and the remaining 40% constitute the source and target partitions, respectively. As shown in Table 5, one characteristic is that the number of distinct items in the source partition (M_{src}) is different from the one in the target partition (M_{tgt}). That is, some items are included only in the source or target partition. We perform and test agent transfer between these source and target partitions.

Table 5: The difference between the number of distinct items in the source partition (M_{src}) and the one in the target partition (M_{tgt}) for each of the experimental datasets.

	Chess	Mushroom	Accidents_10	Connect	Pumsb
M_{src}	72	78	325	127	1981
M_{tgt}	75	106	308	129	1951

Let DQN_{src} and DQN_{tgt} be agents that are defined for the source and target partitions of a dataset, respectively. Based on the experimental results in the previous section, both of DQN_{src} and DQN_{tgt} are based on *GIM-RL-Fusion* that stably yields high performances. In addition, as with the previous section, DQNs with the same structure are used for extracting HUIs and ARs, and simpler DQNs are for FIs. Our idea of agent transfer is very simple. All the parameters of DQN_{src} trained on the source partition are transferred to DQN_{tgt} , except the first and output layers that need to treat the difference of distinct items between the source and target partitions. That is, DQN_{src} and DQN_{tgt} have the same structure with the same parameters except their first and output layers. Each unit in DQN_{src} ’s first layer has a weight for each item. More precisely, it is used to weight s_k ’s value, which represents the usefulness of changing the item’s inclusion in $X(b_k)$ (the itemset defined by b_k). Thus, weights of DQN_{src} ’s first layer for items that are included in both the source and target partitions are replicated on DQN_{tgt} ’s first layer. On the other hand, weights of DQN_{src} ’s first layer for items included only in the source partition, are discarded. For items included only in the target partition, new randomly initialised weights are added to DQN_{tgt} ’s first layer. The same replication, discarding and initialisation of weights are carried out for DQN_{src} ’s and DQN_{tgt} ’s output layers, which individually output $Q(s_k, a; \Theta)$ to select an action for changing the inclusion of an item in $X(b_k)$. After the above-mentioned agent transfer, DQN_{tgt} is re-trained on the target partition. Assuming that the source and target partitions have similar relations among items and parameters transferred from DQN_{src} to DQN_{tgt} are useful for capturing those relations, retraining DQN_{tgt} is expected to be much faster than training a DQN from scratch on the target partition. For simplicity, the latter DQN is called $DQN_{scratch}$.

In Fig. 3, DQN_{tgt} is compared to $DQN_{scratch}$ on the target partition of each dataset. Here, the increase of the number of unique itemsets extracted by DQN_{tgt} over 500 episodes is plotted in orange, and such an increase for $DQN_{scratch}$ is drawn in blue. That is, Fig. 3 visualises how fast each of DQN_{tgt} and $DQN_{scratch}$ is

trained to extract itemsets in the target partition. As illustrated in this figure, in all the 12 cases, DQN_{tgt} finally extracts more itemsets or at least the same number of itemsets to $DQN_{scratch}$. Especially, the 7 cases depicted by the dotted-line rectangles objectively indicate that DQN_{tgt} extracts many itemsets of a target type more quickly than $DQN_{scratch}$. This suggests the potential effectiveness of agent transfer to realise DQN_{tgt} ’s efficient itemset extraction with the help of DQN_{src} .

In addition, let us focus on Chess, Accidents_10%, Connect in the HUI extraction and Pumsb and Connect in the AR extraction in Fig. 3. In each of these cases, DQN_{tgt} extracts no or few itemsets at the beginning, but once it starts to extract itemsets, it becomes to extract many itemsets very fast. This can be thought as follows: Although parameters transferred from DQN_{src} to DQN_{tgt} are useful, they are not directly compatible with a target partition. But, once these parameters are adapted to the target partition, their usefulness brings in extracting many itemsets very quickly. To improve agent transfer in terms of this parameter adaptation, we plan to explore model-based RL that uses an internal model summarising an environment (i.e., dataset) and adaptively updates this model depending on changes of the environment [13].

4.3 Runtime Analysis

Table 6 presents the runtimes of *GIM-RL-Fusion* for each itemset extraction task. These runtimes are measured on a desktop PC equipped with Intel Core i9-9900K (3.60GHz), 32GB RAM and NVIDIA GeForce RTX 2080Ti. *GIM-RL-Fusion* implemented with Pytorch is run on Ubuntu 20.04-LTS. All the source codes for *GIM-RL-Fusion* as well as the other agents are available online, as described in Section C. As can be seen from Table 6, *GIM-RL-Fusion* requires at least more than 30 minutes to finish one task. But, we believe that this slowness is surpassed by *GIM-RL-Fusion*’s great flexibility that any type of itemsets can be extracted as long as a reward for the type can be defined. In other words, *GIM-RL-Fusion* can extract any user-defined type of itemsets with no need to develop a specialised data structure or algorithm for the type.

Table 6: *GIM-RL-Fusion*’s runtimes expressed in the form of “hh:mm:ss”. Column “Acc. / Pumsb” means that Accident_10% is used for the HUI extraction, and Pumsb is used for the FI and AR extractions.

Type	Chess	Mushroom	Acc. / Pumsb	Connect
HUI	00:34:09	00:31:42	04:40:53	12:45:10
FI	00:31:56	00:33:34	00:50:06	00:37:29
AR	00:50:20	00:50:10	01:07:26	00:55:21

The main reason for *GIM-RL-Fusion*’s slow runtime is that it needs to scan a dataset to compute a reward and a state, as illustrated by the non-filled arrows in Fig. 1. For this, we will not develop a specialised data structure or algorithm to accomplish fast scan for a specific purpose. Instead, inspired by the work in [3], we will explore to encode the dataset into a neural network. Its input is an itemset given as a query, and the output is an approximate reward or approximate interestingness measure value for the itemset. This kind of speed-up of *GIM-RL-Fusion* is essential for extending

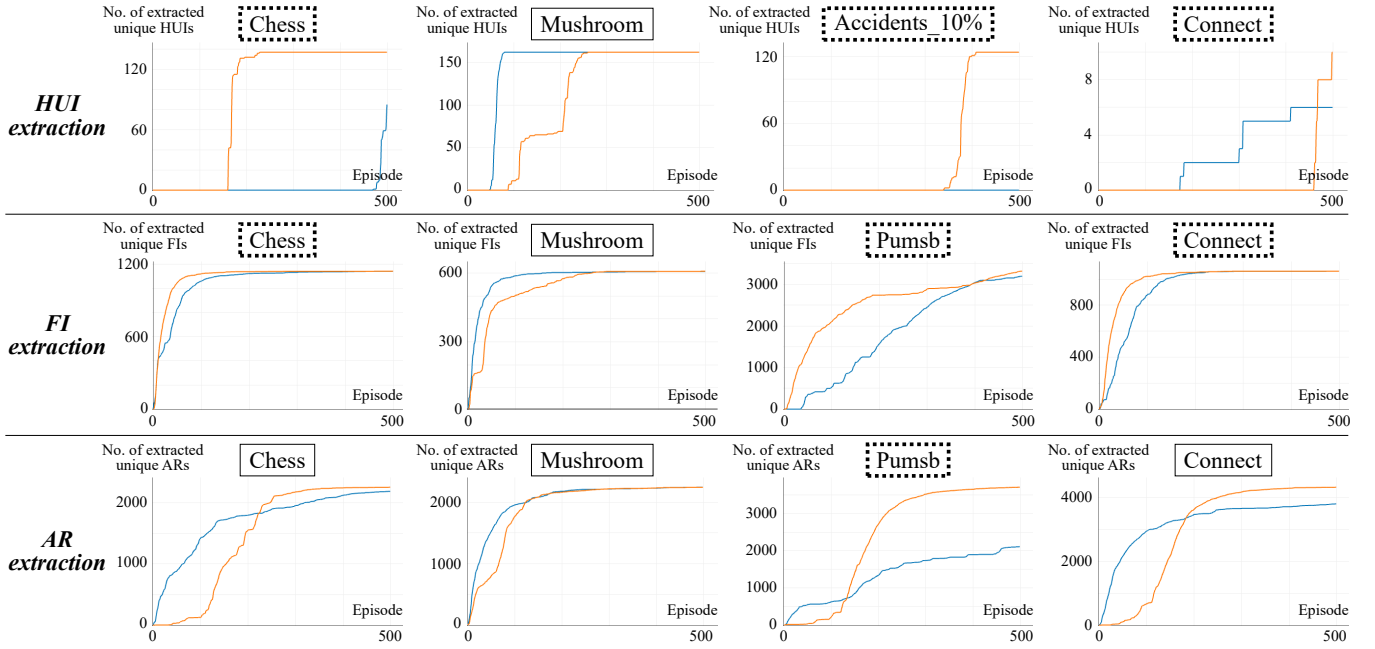


Figure 3: Comparison between the increase of the number of unique itemsets extracted by a transferred agent DQN_{tgt} (orange) and the one by a normally trained agent $DQN_{scratch}$ (blue) over the passage of 500 episodes for each dataset.

it to more complex patterns like sequential or trajectory patterns. While an itemset can be defined by a bit-vector, the representation of a sequential pattern needs a “bit-map” where one row is a bit-vector representing an itemset observed at one time point, and the one of a trajectory pattern requires a “bit-cube” where each bit-matrix at one time point indicates the x-y location of an object. The above dataset encoding is considered necessary for fast querying a dataset in terms of these sequential or trajectory patterns.

5 CONCLUSION AND FUTURE WORK

In this paper, we introduced GIM-RL that offers a unified RL framework to extract various types of itemsets only by changing a reward definition. The general effectiveness of GIM-RL is verified through the experiments on the HUI, FI and AR extractions. The experimental results also suggest one remarkable potential of GIM-RL, namely agent transfer, which realises efficient itemset mining on a dataset with the help of the agent trained on another related dataset. We believe that GIM-RL opens a new direction towards so-called “learning-based itemset mining” and involves many interesting research topics to be studied. In addition to model-based RL in Section 4.2 and dataset encoding in Section 4.3, it is possible to design a reward for extracting a previously unexplored type of itemsets. Moreover, a more intelligent agent may be trained by adopting a planning mechanism to infer itemsets that will be possibly obtained at the multiple-step-ahead future [25].

REFERENCES

- [1] Rakesh Agrawal and Ramakrishnan Srikant. 1994. Fast Algorithms for Mining Association Rules in Large Databases. In *Proc. of VLDB 1994*. 487–499.
- [2] Chowdhury Farhan Ahmed, Syed Khairuzzaman Tanbeer, Byeong-Soo Jeong, and Young-Koo Lee. 2009. Efficient Tree Structures for High Utility Pattern Mining in Incremental Databases. *IEEE Trans. Knowl. Data Eng.* 21, 12 (2009), 1708–1721.
- [3] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. 2017. Neural Combinatorial Optimization with Reinforcement Learning. In *Proc. of ICLR Workshop 2017*.
- [4] Mario Boley, Sandy Moens, and Thomas Gärtner. 2012. Linear Space Direct Pattern Sampling Using Coupling from the Past. In *Proc. of KDD 2012*. 69–77.
- [5] Luca Cagliero and Paolo Garza. 2014. Infrequent Weighted Itemset Mining Using Frequent Pattern Growth. *IEEE Trans. Knowl. Data Eng.* 26, 4 (2014), 903–915.
- [6] Toon Calders, Nele Dexters, Joris J.M. Gillis, and Bart Goethals. 2014. Mining frequent itemsets in a stream. *Inf. Syst.* 39 (2014), 233–255.
- [7] Vladimir Dzyuba, Matthijs Leeuwen, and Luc Raedt. 2017. Flexible Constrained Sampling with Guarantees for Pattern Mining. *Data Min. Knowl. Discov.* 31, 5 (2017), 1266–1293.
- [8] Philippe Fournier-Viger, Jerry Chun-Wei Lin, Bay Vo, Tin Truong Chi, Ji Zhang, and Hoai Bac Le. 2017. A Survey of Itemset Mining. *WIREs Data Min. Knowl. Discovery* 7, 4 (2017), e1207.
- [9] Philippe Fournier-Viger, Cheng-Wei Wu, and Vincent S. Tseng. 2012. Mining Top-K Association Rules. In *Proc. of Canadian AI 2012*. 61–73.
- [10] Philippe Fournier-Viger et al. 2016. The SPMF Open-Source Data Mining Library Version 2. In *Proc. of PKDD 2016*. 36–40.
- [11] Vincent Francois-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. 2018. An Introduction to Deep Reinforcement Learning. *arXiv e-prints* (2018), arXiv:1811.12560.
- [12] Liqiang Geng and Howard J. Hamilton. 2006. Interestingness Measures for Data Mining: A Survey. *ACM Comput. Surv.* 38, 3 (2006), 1–32.
- [13] David Ha and Jürgen Schmidhuber. 2018. Recurrent World Models Facilitate Policy Evolution. In *Proc. of NeurIPS 2018*. 2450–2462.
- [14] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. 2004. Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach. *Data Min. Knowl. Discov.* 8 (2004), 53–87.
- [15] R. Uday Kiran et al. 2018. Efficient Discovery of Weighted Frequent Itemsets in Very Large Transactional Databases: A Re-visit. In *Proc. of Big Data 2018*. 723–732.
- [16] Yun Sing Koh and Sri Devi Ravana. 2016. Unsupervised Rare Pattern Mining: A Survey. *ACM Trans. Knowl. Discov. Data* 10, 4, Article 45 (2016).
- [17] Carson Kai-Sang Leung, Quamrul I. Khan, Zhan Li, and Tariqul Hoque. 2007. CanTree: A Canonical-order Tree for Incremental Frequent-pattern Mining. *Knowl. Inf. Syst.* 11, 3 (2007), 287–311.
- [18] Frédéric Li, Kimiaki Shirahama, Muhammad Adeel Nisar, Xinyu Huang, and Marcin Grzegorzczek. 2020. Deep Transfer Learning for Time Series Data Based

- on Sensor Modality Classification. *Sensors* 20, 15, Article 4271 (2020).
- [19] Mengchi Liu and Junfeng Qu. 2012. Mining High Utility Itemsets without Candidate Generation. In *Proc. of CIKM 2012*. 55–64.
 - [20] Claudio Lucchese, Salvatore Orlando, and Raffaele Perego. 2006. Fast and Memory Efficient Mining of Frequent Closed Itemsets. *IEEE Trans. Knowl. Data Eng.* 18, 1 (2006), 21–36.
 - [21] Jacinto Mata, José-Luis Alvarez, and José-Cristobal Riquelme. 2002. Discovering Numeric Association Rules via Evolutionary Algorithm. In *Proc. of PKDD 2002*. 40–51.
 - [22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari With Deep Reinforcement Learning. In *NIPS Deep Learning Workshop*.
 - [23] Volodymyr Mnih *et al.* 2015. Human-level Control through Deep Reinforcement Learning. *Nature* 518, 7540 (2015), 529–533.
 - [24] Maxime Oquab, Leon Bottou, Ivan Laptev, and Josef Sivic. 2014. Learning and Transferring Mid-level Image Representations Using Convolutional Neural Networks. In *Proc. of CVPR 2014*. 1717–1724.
 - [25] Razvan Pascanu *et al.* 2017. Learning Model-based Planning from Scratch. *arXiv e-prints* (2017), arXiv:1707.06170.
 - [26] Wei Song and Chaomin Huang. 2018. Discovering High Utility Itemsets Based on the Artificial Bee Colony Algorithm. In *Proc. of PAKDD 2018*. 3–14.
 - [27] Wei Song and Chaomin Huang. 2018. Mining High Utility Itemsets Using Bio-Inspired Algorithms: A Diverse Optimal Value Framework. *IEEE Access* 6 (2018), 19568–19582.
 - [28] Sarika Surampudi. 2017. *Oracle Data Mining Concepts*. Oracle. <https://docs.oracle.com/database/121/DMCON/toc.htm>.
 - [29] Akbar Telikani, Amir H. Gandomi, and Asadollah Shahbahrani. 2020. A Survey of Evolutionary Computation for Association Rule Mining. *Inf. Sci.* 524 (2020), 318–352.
 - [30] Vincent S. Tseng, Cheng-Wei Wu, Bai-En Shie, and Philip S. Yu. 2010. UP-Growth: An Efficient Algorithm for High Utility Itemset Mining. In *Proc. of KDD 2010*. 253–262.
 - [31] Takeaki Uno, Masashi Kiyomi, and Hiroaki Arimura. 2004. LCM ver.2: Efficient Mining Algorithms for Frequent/Closed/Maximal Itemsets. In *Proc. of FIMI 2004*.
 - [32] Geoffrey Webb. 2011. Filtered-top-k Association Discovery. *WIREs Data Min. Knowl. Discovery* 1, 3 (2011), 183–192.

SUPPLEMENTAL MATERIALS

A PSEUDO-CODE OF GIM-RL

Algorithm 1 shows a pseudo-code of GIM-RL to train an agent $Q(s, a; \Theta)$ that extracts itemsets satisfying $\varphi(X) \geq \xi$ from a dataset \mathcal{D} . Overall, as expressed by the double for-loop in lines 6–22, the agent is trained through E episodes consisting of K steps. As seen at lines 10–12, at each step, the agent receives a state s_k and computes $Q(s_k, a; \Theta)$ to decide an action a_k for updating the bit-vector b_k into b_{k+1} . After checking the quality of the itemset $X(b_{k+1})$ defined by b_{k+1} at lines 13–15, the environment generates a reward r_k and a new state s_{k+1} at lines 16 and 17. The tuple (s_k, a_k, r_k, s_{k+1}) is then stored into a replay memory \mathcal{P} as one experience of the agent at line 18. Subsequently, the agent’s parameters Θ are updated at line 19, while as written at line 21 the target network’s parameters Θ^- are updated every five episodes by copying Θ to Θ^- . For *GIM-RL-Fusion* in Section 4, $Q(s, a; \Theta)$ at lines 10 and 11 is replaced with $Q'(s_k, a; \Theta) = \lambda s'_k + (1 - \lambda)Q(s_k, a; \Theta)$.

Algorithm 1 GIM-RL (Generic Itemset Mining based on Reinforcement Learning)

Input Dataset \mathcal{D} , interestingness measure $\varphi(X)$, threshold ξ

Output A set \mathcal{X} containing itemsets meeting $\varphi(X) \geq \xi$

```

1: Initialise  $Q(s, a; \Theta)$  with He’s parameter initialisation3
2: Initialise a target network as  $Q(s, a; \Theta^-) = Q(s, a; \Theta)$ 
3: Initialise a replay memory as  $\mathcal{P} \leftarrow \{\}$ 
4: Filter out items that individually have no possibility to be an
   element of itemsets of the target type.
5:  $\mathcal{X} \leftarrow \{\}$ 
6: for  $e = 1, \dots, E$  do
7:   Randomly initialise the bit-vector  $b_1$ 
8:   Generate the first state  $s_1$  based on  $b_1$  and  $\mathcal{D}$ 
9:   for  $k = 1, \dots, K$  do
10:    Compute  $Q(s_k, a; \Theta)$  by feeding  $s_k$  into  $Q(s, a; \Theta)$ 
11:    Decide an action  $a_k$  by the  $\epsilon$ -greedy using  $Q(s_k, a; \Theta)$ 
12:    Update  $b_k$  into  $b_{k+1}$  by  $a_k$ 
13:    if  $\varphi(X(b_{k+1})) \geq \xi$  then
14:       $\mathcal{X} \leftarrow \mathcal{X} \cup \{X(b_{k+1})\}$  // An itemset is extracted
15:    end if
16:    Compute a reward  $r_k$  based on  $b_{k+1}$  and  $\mathcal{D}$  (Section 3.2)
17:    Compute  $s_{k+1}$  based on  $b_{k+1}$  and  $\mathcal{D}$ 
18:     $\mathcal{P} \leftarrow \mathcal{P} \cup \{(s_k, a_k, r_k, s_{k+1})\}$ 
19:    Update  $\Theta$  of  $Q(s, a; \Theta)$  using experiences randomly sam-
       pled from  $\mathcal{P}$ 
20:   end for
21:   Update  $\Theta^-$  of  $Q(s, a; \Theta^-)$  as  $\Theta^- = \Theta$  (every 5 episodes)
22: end for
23: return  $\mathcal{X}$ 

```

We mention the following three implementation details: First, the search space reduction before extracting itemsets is done at line 4. Specifically, for the extraction of HUIs, items whose upper bound utilities (transaction weighted utilisations [2, 19, 26, 30]) are

³K. He, X. Zhang, S. Ren and J. Sun: Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, In *Proc. of ICCV 2015*, pp. 1026–1034 (2015)

less than the threshold ξ are discarded, and for the extractions of FIs and ARs, items whose supports are less than the minimum support threshold (ξ for FIs, and ξ_1 for ARs) are eliminated. Second, at line 7, b_1 in each episode is obtained by the random bit-vector initialisation described in Section 3.1. We assume that the more frequently an item occurs in \mathcal{D} , the more likely it is to be included in HUIs, FIs or ARs. The above-mentioned search space reduction and random bit-vector initialisation need to be modified when targeting another type of itemsets like infrequent itemsets. Last, to help the agent explore a variety of itemsets, line 11 shows that a_k is decided using the ϵ -greedy strategy. Here, a_k is chosen as the action corresponding to the highest output of $Q(s_k, a; \Theta)$ with the probability $1 - \epsilon$, while a_k is set to the action to change the value of a randomly selected dimension in b_k with the probability ϵ .

B DQN ARCHITECTURES

Fig. 4 (a) and (b) show the DQN architecture for extracting HUIs and ARs and the one for FIs, respectively. These architectures are common with respect to the input and output layers. The input layer accepts an M -dimensional state vector $s_k = (s_{k,1}, \dots, s_{k,M})^T$ where $s_{k,m}$ ($1 \leq m \leq M$) represents an estimated usefulness of changing the m th item’s inclusion in $X(b_k)$. The output layer produces an $(M+1)$ -dimensional vector $Q(s_k, a; \theta) = (q_{k,1}, \dots, q_{k,M}, q_{k,M+1})^T$ where $q_{k,m}$ indicates an approximate quality of the action to change the m th item’s inclusion in $X(b_k)$, and the last $q_{k,M+1}$ expresses an approximate quality of the action to randomly initialise b_k . As illustrated by the three dotted arcs in Fig. 4 (a), DQNs used for the HUI and AR extractions have three blocks each of which consists of a Fully-Connected (FC) layer, a batch normalisation layer and an activation layer defined by leaky ReLU. Similarly, as shown in Fig. 4 (b), DQNs for the FI extraction is comprised of one block with an FC layer having a large number of units. Based on our preliminary experiments, the structural complexity of a DQN may be related the complexity of a target type. That is, FIs are simpler than HUIs and ARs, so simpler DQNs seem enough for extracting FIs compared to the ones for HUIs and ARs.

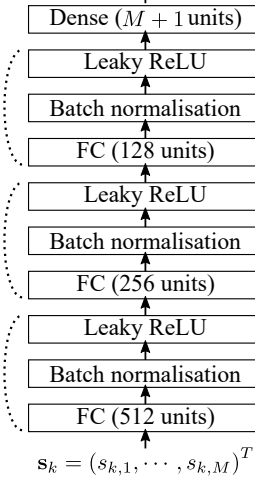
We describe details of how to compute $s_{k,m}$ in s_k . First of all, $\varphi(X)$ has a huge value range. For example, assuming that $\varphi(X)$ outputs the support of an itemset X , one itemset may have a support of 1000 while the support of a rare itemset may be 1. Thus, if $s_{k,m}$ is defined directly as $\varphi(X(b'_k))$ where $X(b'_k)$ is the itemset created by changing the m th item’s inclusion in $X(b_k)$ (please see Section 3.1), $s_{k,m}$ ’s value range is huge, and s_k is often very biased in the sense that only some dimensions have very large values. To alleviate this bias, $s_{k,m}$ is computed by normalising $\varphi(X(b'_k))$ as follows:

$$s_{k,m} = \log \left(\frac{\varphi(X(b'_k))}{Z} + 1 \right), \quad (4)$$

where Z is the normalisation factor to put $\varphi(X(b'_k))/Z$ between 0 and 1. For this, using Z is common in itemset mining, for instance, in FI extraction, Z is the number of transactions in \mathcal{D} to compute the “relative” support of an itemset [1, 8, 14], and in HUI extraction, relative utilities are calculated by setting Z to the sum of utilities for all transactions (i.e., sum of transaction utilities) in \mathcal{D} [19, 26, 27, 30]. These definitions of Z are also used in Eq. 4, which then

a) HUI and AR extractions

$$Q(s_k, a; \Theta) = (q_{k,1}, \dots, q_{k,M}, q_{k,M+1})^T$$



b) FI extraction

$$Q(s_k, a; \Theta) = (q_{k,1}, \dots, q_{k,M}, q_{k,M+1})^T$$

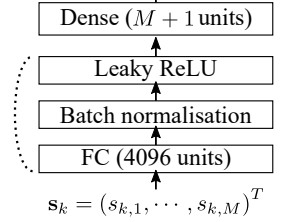


Figure 4: DQN architectures used for the HUI, FI and AR extractions.

takes \log of $(\varphi(X(b'_k))/Z + 1)$ to further reduce value differences among $s_{k,1} \dots, s_{k,M}$ in s_k .

One natural extension of s_k is performed for AR extraction involving two interestingness measures $\varphi_1(X)$ and $\varphi_2(X)$ that output the support and confidence of X , respectively. Specifically, we compute a $2M$ -dimensional state vector s_k . Here, $s_{k,m}$ in the first M dimensions is computed based on $\varphi_1(X(b'_k))$ and Z being the number of transactions in \mathcal{D} . On the other hand, $s_{k,m}$ in the second M dimensions is computed based on $\varphi_2(X(b'_k))$ and $Z = 1$, because $\varphi_2(X(b'_k))$ representing the confidence of the rule defined by $X(b_k)$ and $X(b'_k)$ already lies between 0 and 1. Finally, s_k is fed into a batch normalisation layer to normalise values by $\varphi_1(X(b'_k))$ and $\varphi_2(X(b'_k))$, and the resulting normalised s_k is used as the input of the DQN architecture in Fig. 4 (a). For *GIM-RL-Fusion*, the first M dimensions based on $\varphi_1(X(b'_k))$ are used to create s'_k .

C HYPER-PARAMETER SETTING

Referring to Algorithm 1 in Section A, the number of episodes E is set to 500 for the HUI extraction and to 1000 for the FI and AR extractions in Section 4.1, and to 500 for all the experiments of agent transfer in Section 4.2. The number of steps in one episode is always $K = 500$. The size of a replay memory \mathcal{P} is set to 10000. Each update of the agent’s parameters Θ is based on Eq. 2 where the discount factor γ is set to 0.95. This update of Θ is done using RAdam⁴ (with the initial learning rate 0.001) as an optimiser on a mini-batch of 512 experiences randomly sampled from \mathcal{P} .

Regarding λ in $Q'(s_k, a; \Theta) = \lambda s'_k + (1 - \lambda)Q(s_k, a; \Theta)$ of *GIM-RL-Fusion*, λ is dynamically changed. As described in Section 4.1, at the beginning of training, λ is large in order to facilitate finding

⁴L. Liu *et al.*: On the Variance of the Adaptive Learning Rate and Beyond, In Proc. of ICLR 2020, 2020

positive itemsets based on s'_k , and is gradually reduced to prioritise the agent’s exploitation based on the trained $Q(s_k, a; \Theta)$. Inspired by the ϵ -greedy strategy, we implement this dynamic change of λ as follows:

$$\lambda = \lambda_{end} + (\lambda_{start} - \lambda_{end}) \exp\left(-\frac{k_{total}}{\Delta}\right), \quad (5)$$

where λ_{start} and λ_{end} are the maximum and minimum values of λ , respectively. k_{total} is the total number of steps the agent has experienced, that is, $k_{total} = e \times K + k$ for the k th step in the e th episode. In Eq. 5, λ is λ_{start} when $k_{total} = 0$, and gradually converges to λ_{end} as k_{total} increases. Δ is a hyper-parameter to control the speed of this convergence.

First, Δ is set to 200 in all the experiments in Section 4. In Section 4.1, $\lambda_{start} = 0.999$ and $\lambda_{end} = 0.5$ are used for extracting HUIs and ARs, and $\lambda_{start} = 0.999$ and $\lambda_{end} = 0.6$ are for FIs. For this, we think that s_k provides “dataset-dependent” information for changing each item’s inclusion in $X(b_k)$ by actually checking transactions in a dataset, while $Q(s_k, a; \Theta)$ represents more general information obtained from a DQN. According to this thought, HUIs and ARs are more complex than FIs, so the smaller $\lambda_{end} = 0.5$ is used to put a higher priority on $Q(s_k, a; \Theta)$.

For the experiments of agent transfer in Section 4.2, λ_{start} and λ_{end} are set depending on source and target datasets, and item-set types. Specifically, for the HUI extraction, $\lambda_{start} = 0.999$ and $\lambda_{end} = 0.5$ are used to train an agent on a source dataset, and $\lambda_{start} = \lambda_{end} = 0.5$ (i.e., λ is constant at 0.5) is used on a target

dataset in order to take more advantage of $Q(s_k, a; \Theta)$ obtained from the source dataset. For the FI extraction, $\lambda_{start} = 0.999$ and $\lambda_{end} = 0.6$ are used for both source and target datasets, to take more account of dataset-dependent information for the target dataset.

For the AR extraction, we use $\lambda_{start} = \lambda_{end} = 0.5$ on a source dataset to obtain general information about items as $Q(s_k, a; \Theta)$, and then $\lambda_{start} = 0.999$ and $\lambda_{end} = 0.5$ are used to gradually adapt $Q(s_k, a; \Theta)$ to a target dataset by considering dataset-dependent information. We will explore a more sophisticated approach to define λ_{start} and λ_{end} based on statistical characteristics of source and target datasets.

Finally, all the source codes (including the ones for data download) used in this paper are available on our Github repository.

D SMALL REMARK ABOUT MUSHROOM

The Web page⁵ of SPMF library [10] provides two versions of Mushroom, one for the HUI extraction and the other for the FI and AR extractions. Regarding the number of distinct items in a source partition M_{src} and the one in a target partition M_{tgt} in Table 5 of Section 4.2, these two versions has a small difference. Specifically, Mushroom for the HUI extraction is divided into the source partition with $M_{src} = 98$ and the target one with $M_{tgt} = 109$, and the source and target partitions of Mushroom for the FI and AR extractions have $M_{src} = 78$ and $M_{tgt} = 106$, respectively. Only the latter M_{src} and M_{tgt} are shown in Table 5 for ease of understanding.

⁵<http://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php>