Mining Antagonistic Communities from Social Networks

Kuan Zhang, David Lo, and Ee-Peng Lim

School of Information Systems, Singapore Management University
{kuang.zhang.2008, davidlo, eplim}@smu.edu.sg

Abstract. During social interactions in a community, there are often sub-communities that behave in opposite manner. These antagonistic sub-communities could represent groups of people with opposite tastes, factions within a community distrusting one another, etc. Taking as input a set of interactions within a community, we develop a novel pattern mining approach that extracts for a set of antagonistic sub-communities. In particular, based on a set of user specified thresholds, we extract a set of pairs of sub-communities that behave in opposite ways with one another. To prevent a blow up in these set of pairs, we focus on extracting a compact lossless representation based on the concept of closed patterns. To test the scalability of our approach, we built a synthetic data generator and experimented on the scalability of the algorithm when the size of the dataset and mining parameters are varied. Case studies on an Amazon book rating dataset show the efficiency of our approach and the utility of our technique in extracting interesting information on antagonistic sub-communities.

1 Introduction

We form opinions and at times strong convictions on various issues and questions. Based on similarity in opinions and ideals, it is common that sub-groups or communities of users are formed. As members support or uphold a particular view or even conviction, we also observe the dynamics of human social interaction of *antagonistic groups*, i.e., two groups that consistently differ in opinions.

Opposing groups and their nature have been studied in the sociology domain [17, 5, 4, 14, 10, 6]. Understanding the formation of these groups and widespread-ness of opposing communities are of research interest. They could potentially signify signs of disunity in the larger community and point to sub-communities that oppose one another. If these issues could be detected early, unwanted tensions between communities could potentially be averted. Identification of antagonistic communities is also the first step to further studies on: e.g., how the antagonistic communities are formed, why they are formed, how does the antagonistic communities grow over time, when do the communities stop being antagonistic, etc.

Aside from enriching studies on dynamics of social interactions, information on groups of people having opposing opinions could potentially be used for: designing better marketing/product survey strategy by deeper understanding on the nature of each sub-community and potentially an opposing one, better recommendation of friends, or even recommendation of "non-friends", e.g., those whose reviews one could ignore, etc.

In this study, our goal is to discover antagonistic communities automatically from their history of interactions. We design a novel pattern mining algorithm to directly mine antagonistic communities. We take as input a database of user opinions/views over things, bucketized into high/medium/low or positive/neutral/negative. From this database, we extract every two sets of users that are antagonistic over enough number of common items/issues with a high likelihood. Each mined pattern identifies a group of users that oppose another group over a sufficient number of common issues/items of interest (i.e., enough *support*) with a high likelihood (i.e., enough *confidence*).

Our approach explores the search space of all possible opposing communities and prunes those that appear with not enough frequency/support. An apriori-like anti-monotonicity property is used to prune the search space. Eventually the patterns mined are checked if the confidence is sufficient. If it is, it would then be reported. As a frequent antagonistic pattern would have many sub-patterns that are frequent we only report patterns that are *closed*. A pattern is closed if there exists no super-pattern having the same support and confidence.

To show the scalability of our approach, we developed a synthetic data generator in a similar fashion as the IBM data generator used for mining association rules [2]. The data generator is used to test the scalability of our approach on several dimensions of interest. The result shows that our algorithm is able to run well on various parameter settings. We also investigates a rating dataset from Amazon. Our algorithm is able to run on the real dataset and extract antagonistic communities. A few hundred communities are mined from the dataset.

The contributions of our work are as follows:

1. We propose a new problem of mining antagonistic communities from social network. Mined antagonistic communities could potentially be used to shed better light on social interactions, prevent unwanted tensions in the communities, improve recommendations and marketing strategies, etc.
2. We propose a new algorithm to mine for antagonistic communities that is shown to be scalable.
3. We extract antagonistic communities from real datasets shedding light to the extent of consistent antagonistic behavior in real rating datasets.

The structure of this paper is as follows. Section 2 describes some related work. Section 3 formalizes some concepts and the semantics of antagonistic communities. Section 4 describes our mining algorithm. Experiments and case studies are described in Section 5. We finally conclude and describe future work in Section 6.

2 Related Work

There have been a number of studies on finding communities in social network [3, 9, 8]. In this study we enrich past studies by discovering not cohesive communities but ones with opposing sub-communities. We believe these two source of information could give more light to the social interactions among users in Web 2.0.

Antagonistic communities is also related to the concept of homophily. Members of a pair of antagonistic communities, intuitively share more preferences with those in the same community and share less preferences with others from the opposing community. There have been a number of studies on homophily in social networks [16]. In this work, our mined communities not only express similar preferences but also opposing preferences. Homophily and trust are closely related as users with similar preferences are more likely to trust each other [11]. In this sense, our work enriches existing studies on homophily and trust [12, 15, 13].

In the sociology, economics, and psychology communities, the concept of inter-group antagonism has been studied by various work [17, 5, 4, 14, 10, 6]. We extend this interesting research question by providing a computation tool to automatically identify opposing communities from a history of their behaviors. We believe our tool could potentially be used to help sociologist in understanding the behaviors of communities from the wealth of available data of user interactions in Web 2.0.

Our algorithm belongs to a family of pattern mining algorithms. There have been a number of pattern mining algorithms including those mining association rules (e.g., [2]), frequent sequences (e.g., [18]), frequent repetitive sequences (e.g., [7]), frequent graphs, etc. The closest to our study is the body of work on association rule mining [2]. Association rule mining also employs the concept of support and confidence like us. However, association rule mining extracts frequent transactions, and relationship between transactions. On the other hand, we extract two sets of opposing users that share many common interests/form opinions/commonly rated items but oppose each other with high likelihood. This problem is inherently different from association rule mining. We show that a similar apriori-like anti-monotonicity property holds but we employ a different algorithm to mine for antagonistic communities. Similar to the work in [18], we do not report all frequent and confident groups rather only the *closed* ones.

3 Antagonistic Group

We formalize past histories of user social interactions in terms of ratings to items which can be objects, views, or even ideas. Hence there is a bipartite graph between users and objects where the arrows are labeled with rating scores. We divide all rating scores to be high, medium, low **rating polarities** depending on the score ranges. For example in Epinions where there is a 5-point scale assigned to an item by a user, we bucketize rating scores of $1-2$ to be of low rating polarity, 3 to be of medium rating polarity, and $4-5$ to be of high rating polarity. We formalize our input as a database of ratings, defined in Definition 1. We refer to the size of a rating database DB_R as $|DB_R|$ which is equal to the number of mapping entries in the database.

Definition 1. *Consider a set of users U and a set of items I. A database of ratings consists of a set of mappings of item identifiers to a set of pairs, where each pair consists of user identifier and rating score. There are three types of rating scores considered: high (hi), medium (mid), and low (lo). The database of ratings could be formally represented as:*

$$DB_R = \{it_{id} \mapsto \{(us_{id}, rtg), \ldots\} | it_{id} \in I \wedge us_{id} \in U \wedge rtg \in \{hi, mid, lo\} \wedge \\ us_{id} \textit{ gives } it_{id} \textit{ a rating of } rtg\}$$

Two ratings are said to be common between two users if the ratings are assigned by the two users on the same item. A set of ratings is said to be common among a set of users if these ratings are on a common set of items rated by the set of users.

Definition 2. ***(Antagonistic Group):*** *Let U_i and U_j be two disjoint sets of users. (U_i, U_j) is called an antagonistic group (or simply, a-group) if at least σ of their common ratings satisfy all the following conditions:*

- *Users from U_i share the same rating polarity p_i;*
- *Users from U_j share the same rating polarity p_j; and*
- *p_i and p_j are opposite polarities.*

The number of common ratings between two sets of users U_i and U_j is known as their **support count** and is denoted by $count(U_i, U_j)$. The **support** of the two user sets $support(U_i, U_j)$ is defined as $\frac{count(U_i,U_j)}{|I|}$ where I represents the set of all items.

The number of common ratings between U_i and U_j that satisfy the three conditions in the antagonistic group definition (see Definition 2) is called the **antagonistic count**, denoted by $antcount(U_i, U_j)$. Obviously, $antcount(U_i, U_j) \leq count(U_i, U_j)$. The **antagonistic support** of the two user sets $asupport(U_i, U_j)$ is defined as $\frac{antcount(U_i,U_j)}{|I|}$. We also define the **antagonistic confidence** of a a-group (U_i, U_j) to be $aconf(U_i, U_j) = \frac{antcount(U_i,U_j)}{count(U_i,U_j)}$.

Definition 3. ***(Frequent Antagonistic Group):*** *An antagonistic group (U_i, U_j) is frequent if $support(U_i, U_j) \geq \lambda$ and $asupport(U_i, U_j) \geq \lambda \times \sigma$ where λ is the support threshold ($\in (0, 1)$), and σ is the* ***antagonistic confidence threshold*** *($\in (0, 1)$).*

We consider (U_i, U_j) **to subsume** (U'_i, U'_j) if: (a) $U'_i \subset U_i$ and $U'_j \subseteq U_j$; or (b) $U'_i \subseteq U_i$ and $U'_j \subset U_j$. We denote this by $(U'_i, U'_j) \subset (U_i, U_j)$.

Frequent a-groups satisfy the important Apriori property as stated below. Due to space constraint, we move the proof to [1].

Property 1. **(Apriori Property of Freq. A-group):** Every size $(k-1)$ a-group (U'_i, U'_j) subsumed by a size-k frequent a-group (U_i, U_j) is frequent.

Definition 4. ***(Valid Antagonistic Group):*** *An a-group (U_i, U_j) is valid if it is frequent and $aconf(U_i, U_j) \geq \sigma$.*

Definition 5. ***(Closed Antagonistic Group):*** *A valid a-group* (U_i, U_j) *is closed if* $\neg\exists(U_i', U_j').(U_i, U_j) \subset (U_i', U_j')$, $count(U_i', U_j') = count(U_i, U_j)$ *and* $antcount(U_i', U_j') = antcount(U_i, U_j)$.

Example 1. Consider the example rating database in Table 1 (left). Suppose $\lambda = 3$ and $\sigma = 0.5$. Both (a, d) and (a, bd) are valid a-groups. However, since $count(a, d) = count(a, bd) = 3$ and $antcount(a, d) = antcount(a, bd) = 2$, (a, d) is not a closed a-group and is subsumed by (a, bd). Hence, (a, d) is considered as redundant. On the other hand, both (a, b) and (a, bc) are closed a-groups even though both $aconf(a, b)$ and $aconf(a, bc)$ has the same value which is $\frac{2}{3}$. This is so as $count(a, b) \neq count(a, bc)$ and $antcount(a, b) \neq antcount(a, bc)$.

Table 1. Example Rating Database 1 (DB_{EX1}), 2, and 3

Item	User ratings
i_1	a-hi, b-lo, d-lo
i_2	a-hi, b-lo, d-lo
i_3	a-hi, b-hi, d-hi
i_4	a-hi, b-lo, c-lo
i_5	a-hi, b-lo, c-lo
i_6	a-hi, b-hi, c-lo

Item	User ratings
i_1	a-hi, b-lo, c-lo
i_2	a-hi, b-lo, c-lo
i_3	a-hi, b-lo, c-hi
i_4	d-hi, e-lo, f-lo
i_5	d-hi, e-hi

Item	User ratings
i_1	a-hi, b-lo, d-lo
i_2	a-hi, b-lo, d-lo
i_3	a-hi, b-hi, d-hi

Note that $count(U_i, U_j) = count(U_i', U_j')$ does not imply that $antcount(U_i, U_j) = antcount(U_i', U_j')$ for any $(U_i, U_j) \subset (U_i', U_j')$, and vice versa. We can show this using the rating database example in Table 1 (middle). In this example, we have $count(a, b) = count(a, bc) = 3$ but $(antcount(a, b) = 3) > (antcount(a, bc) = 2)$. We also have $antcount(d, e) = antcount(d, ef) = 1$ but $(count(d, e) = 2) > (count(d, ef) = 1)$.

Definition 6. ***(Antagonistic Group Mining Problem):*** *Given a set of items* I *rated by a set of users* U*, the antagonistic group mining problem is to find all closed antagonistic groups with the given support threshold* λ *and antagonistic confidence threshold* σ.

4 A-Group Mining Algorithm

We develop a new algorithm to mine for antagonistic groups from a database of rating history. The database could be viewed as a cleaned representation of people opinions or views or convictions on various items or issues. Our algorithm systematically traverses the search space of possible antagonistic groups using a search space pruning strategy to remove unfruitful search spaces.

The a-group mining algorithm runs for multiple passes. In the initialization pass, we calculate the *count* and *antcount* of all the size-2 a-group candidates and determine which of them are frequent. In the next pass, with the set of frequent a-groups found in the previous pass, we generate new potential frequent a-groups, which are called *candidate* set. We then count the actual *count* and *antcount* values for these candidates. At the end of this pass, we determine the frequent candidates, and they are used to generate frequent a-groups for the next pass. After that, we filter the previous frequent a-group set with the newly generated frequent a-group set by removing non-closed a-groups. Then we move on to the next pass. This process continues until no larger a-groups are found. After successful mining of all frequent a-groups, we derive the valid a-groups from them.

```
   Input: $\lambda$; $\sigma$; rating database
   Output: valid and closed a-group of all size
1  $L_1$ = frequent user set;
2  $C_2 = \{(\{u_i\}\{u_j\})|i < j, u_i \in L_1, u_j \in L_1\}$;
3  for $k = 2; k \leq |U|$ and $|L_{k-1}| \neq 0$; $k{+}{+}$ do
4      if $k > 2$ then
5          $C_k$=antGrpMining-gen($L_{k-1}$);
6      end
7      root$\leftarrow$ buildHashTree($k$,$C_k$);
8      foreach item $t \in \mathcal{D}$ do
9          $C_t$=subset($t$,root);
10         foreach candidate $c$ in $C_t$ do
11             update count and antcount of $c$;
12         end
13     end
14     $L_k=\{g_k \in C_k | \frac{count(g_k)}{|I|} \geq \lambda$ and $\frac{antcount(g_k)}{|I|} \geq \lambda \times \sigma\}$;
15     $L_{k-1}$=prune($L_{k-1}, L_k$);
16 end
17 $G=\{g \in \bigcup_k L_k | \frac{antcount(g)}{count(g)} \geq \sigma\}$;
18 Output $G$;
```

Algorithm 1: Mining Algorithm – Clagmine(λ,σ,DB_R)

Algorithm 1 shows the a-group mining algorithm known as Clagmine. Two basics data structures are maintained namely L_k the intermediary set of frequent a-groups of size k and C_k a candidate set of size k for valid a-groups checking. The first two lines of the algorithm derives size-2 candidates to get the frequent size-2 a-groups. It forms the base for subsequent processing. A subsequent pass, say pass k, consists of three phases. First, at line 5, the a-groups in L_{k-1} found in $k-1$ pass are used to generate the candidate a-group set C_k, using the antGrpMining-gen method in Algorithm 2. Next, the database is scanned and the count and antcount of candidates in C_k is updated (lines 7 to 13). We make use of the hash-tree data structure described in [2] to hold C_k and we then use a subset function to find the candidates overlap with the raters of an item. After we marked all the overlapped candidates, we update the count and antcount of them. Frequent a-groups can be determined by checking count and antcount against the support threshold and $\lambda \times \sigma$ thresholds respectively. Following that, L_{k-1} is filtered with the newly generated a-groups to remove non-closed a-groups (line 15). After all the passes, the valid a-group is determined from the frequent a-group set (line 17). The following subheadings zoom into the various components of the mining algorithm in more detail.

```
Input: size-$(k-1)$ frequent a-group set $L_{k-1}$
Output: size-$k$ candidate frequent a-group set
foreach $p, q \in L_{k-1}$ do
    $g_k \leftarrow merge(p, q)$;
    add $g_k$ to $C_k$;
    forall $(k-1)$-subsets $s$ of $g_k$ do
        if $s\neg \in L_{k-1}$ then
            delete $g_k$ from $C_k$;
        end
    end
end
return $C_k$;
```

Algorithm 2: antGrpMining-gen(L_{k-1})

Candidate Generation and Pruning. The antGrpMining-gen function described in Algorithm 2 takes L_{k-1}, the set of all frequent size-$(k-1)$ a-groups as input. It returns a superset of all frequent size-k a-groups. It works as below. First, we merge all the elements in L_{k-1} that share the same sub-community of size-(k-2). Each of them can be merged into a size-k candidate a-group consisting of the common sub-community and the two differing members. We add the candidate a-groups to C_k. Next, in the pruning stage, we delete $g_k \in C_k$ if some $(k-1)$ subset of g_k is not in L_{k-1}.

The pruning stage's correctness is guaranteed by Property 1. From the property, if g_k is frequent, all its $(k-1)$ subsets must be frequent. In other words, if any one $(k-1)$ subset of an a-group g_k is not frequent, g_k is not frequent too. We thus prune such g_ks. The correctness of antGrpMining-gen function follows from Lemma 1. Due to space constraint, we move the proofs of all lemmas and theorems to [1].

Lemma 1. *For $k \geq 3$, given a set of all size-$(k-1)$ frequent a-group, i.e., L_{k-1}, every size-k frequent a-group, i.e., L_k, is in the candidate set, i.e., C_k, output by Algorithm 2.*

An example to illustrate the process of candidate generation via merging and deletion is given below.

Example 2. Let L_3 be $\{(u_1, u_2u_3),(u_5, u_2u_3),(u_1u_4, u_2),(u_1u_5, u_2),(u_4u_5, u_2)\}$. After the merge step, C_4 will be $\{(u_1u_5, u_2u_3), (u_1u_4u_5, u_2)\}$. The deletion step serving as apriori-based pruning, will delete the a-group (u_1u_5, u_2u_3) because the a-group (u_1u_5, u_3) is not in L_3. We will then left with only $\{(u_1u_4u_5, u_2)\}$ in C_4.

Subset Function. Candidate a-groups are stored in a hashtree as mentioned in line 7 of Algorithm 1. Each node of the hashtree contains either a hashtable (interior node), or a list of candidates (leaf). Each node is labeled with a user identifier representing the user associated with this node. The hashtable at interior nodes contains mappings to nodes at the next level, with each hash key being the corresponding user identifier. Every candidate is sorted according to the user identifier, and is then inserted into the hashtree.

The subset function in line 9 of Algorithm 1 finds all the candidate a-groups among raters of item t. The raters of item t is first sorted by their user identifiers. The raters are then traversed one by one. A pointer list is kept to maintain a list of nodes which are visited, which initially has only the root of the hashtree. For a rater u, we traverse through all the nodes in the pointer list, if a child node of the current node is found with label u, the child node is further checked to see whether it is interior or leaf. If it is an interior node, we add it to the pointer list and if it is a leaf, every a-group stored in the leaf is marked as a subset of raters of t. A node is removed from the pointer list if all of its child nodes are in the list (i.e., are visited). The process is repeated through all the raters of item t. At the end, all the candidates which are subset of raters of t will be marked.

Filtering Non-Closed A-Group. The filtering of non-closed a-groups corresponds to line 15 in Algorithm 1. The function works as follows. For each a-group g_k in L_k, we traverse through every a-group g_{k-1} in L_{k-1}. If g_k subsumes g_{k-1}, and the count and antcount of the two groups are equal, g_{k-1} can be filtered. This step ensures all the a-groups in L_{k-1} are closed. By iterating through k, we can have all the non-closed a-group of any size filtered. Note that a closed a-group could potentially subsumes a combinatorial number of sub-groups. Removal of non-closed a-group potentially reduces the number of reported a-groups significantly.

Correctness of the algorithm. The correctness of the algorithm is guaranteed by Theorems 1 & 2 stated below.

Theorem 1. *Mined a-group set G contains all valid and closed a-groups.*

Theorem 2. *Mined a-group set G contains only valid and closed a-groups.*

Scalability Variant: Divide and Conquer Strategy. At times, the main memory required to generate all the candidates could be prohibitive. If there are too many L_2 patterns, storing all of them in the memory would not be feasible. To address this issue, we perform a divide and conquer strategy by partitioning the database, mining for each partition, and merging the result. We first state some new definitions and describe a property.

Definition 7 (User Containment). *Consider a member $m = it_{id} \mapsto PairSet$ in a database of ratings DB_R. We say that a user u_i is contained in the entry, denoted by $u_i \in m$, iff $\exists$ (u_i, rtg) where $rtg \in \{hi, lo, mid\}$ and (u_i, rtg) is in $PairSet$. We also say that a user u_i is in an a-group $a = (S_1, S_2)$ iff ($u_i \in S1$ $\vee$ $u_i \in S2$)*

Example 3. To illustrate, consider the first entry etr in the example rating database shown in Table 1(left). The first entry etr contains users a, b and d: $a \in etr$, $b \in etr$, and $d \in etr$.

Definition 8 (Database Partition). *Consider a user u_i and a database of ratings DB_R. The partition of the database with respect to user u_i, denoted as $DB_R[u_i]$ is defined as: $\{etr | u_i \in etr \wedge etr \in DB_R\}$*

Example 4. To illustrate, projection of the database shown in Table 1(left) with respect to user d is the database shown in Table 1(right).

Using the two definitions above, Lemma 2 describes the divide and merge mining process.

Lemma 2 (Divide and Merge). *Consider a database of ratings DB_R, support threshold λ, and confidence threshold σ. Let U_{set} be the set of users in DB_R and Cm be the shorthand of the Clagmine operation in Algorithm 1. The following is guaranteed:*
$Cm(\lambda, \sigma, DB_R) = \bigcup_{u_i \in U_{Set}} \{g | u_i \in g \wedge g \in Cm(\frac{\lambda \times |DB_R|}{|DB_R[u_i]|}, \sigma, DB_R[u_i])\}$

Based on Lemma 2, our algorithm to perform divide and conquer is shown in Algorithm 3. The algorithm partitions the database one item at a time and subsequently calls the original closed antagonistic group mining algorithm defined in Algorithm 1. Theorem 3 guarantees that the mined result is correct and a complete set of a-groups are mined by Algorithm 3.

Theorem 3. *Algorithm 3 would return a complete set of closed and valid a-groups and all returned a-group would be closed and valid.*

Note that the divide and conquer algorithm reduces memory costs however it could potentially increase the runtime cost since the database would now need to be scanned more number of times. In Section 5, we show the results of running the two algorithms over a number of datasets.

5 Performance & Case Studies

In this section we describe our performance study using various data generated from our synthetic data generator with various parameter values. We then describe a case study from a real book rating dataset.

Performance Study. As a summary, our synthetic data generator accepts as input I (in '000)(the number of items), U (in '000)(the number of users), P (the expected number of users rating an item), N_G (average size of maximal potential large a-group), and N_L (in '000) (number of maximal potential large a-group). We use the following datasets:

DS_1	I=100, U=10, P=20, N_G=6, N_L=2
DS_2	I=100, U=50, P=20, N_G=6, N_L=2

The result for dataset DS_1 when varying the support threshold from 0.002 to 0.006 with σ=0.7 is shown in Figure 1. The first graph shows the runtime needed to execute the algorithm at various support thresholds. "Non-split" and "Split" correspond to Algorithms 1 & 3 respectively. We only include 3 data points for

```
  Input: $\lambda$; $\sigma$; rating database
  Output: valid and closed a-group of all size
1 $U_{Set}$ = Set of all users in $DB_R$;
2 $G = \{\}$;
3 foreach $u_i \in U_{Set}$ do
4     $G = G \cup \{ag | u_i \in ag \wedge ag \in$ Clagmine($\frac{\lambda \times |DB_R|}{|DB_R[u_i]|}$,$\sigma$,$DB_R[u_i]$)$\}$;
5 end
6 Output $G$;
```

Algorithm 3: Clagmine-partitional(λ,σ,DB_R)

"Non-split", as mining at lower thresholds are too long to complete. The second graph shows the numbers of a-groups mined at various support thresholds.

The result shows that the time taken grows larger when support threshold is reduced. This growth is accompanied by the growth in the number of a-groups mined. Also, many longer patterns are mined as the support threshold is lowered.

For DS_2, we consider a larger number of users. The results for various support thresholds with σ=0.7 are shown in Figure 2. We have also conducted additional performance studies and their results can be found in our technical report [1].

The performance study has shown that the algorithm is able to run well on various settings. The lower the support threshold the more expensive it is to mine. Also, the larger the number of users (or items or expected number of users rating an item – see [1]), the more expensive it is to mine.

Case Study. For the case study, we consider a dataset of book ratings from Amazon. There are a total of 99,255 users ratings 108,142 books in 935,051 reviews. The experiment is run with σ=0.5. The result is shown in Figure 3.

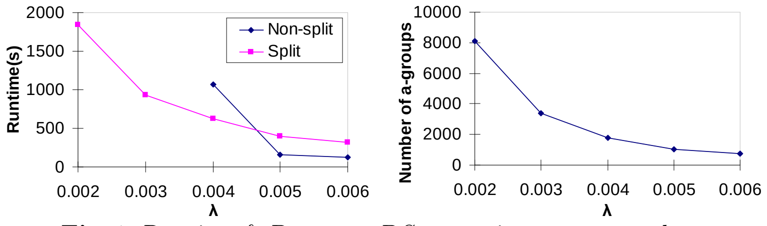

Fig. 1. Runtime & Patterns: DS_1 at various support values.

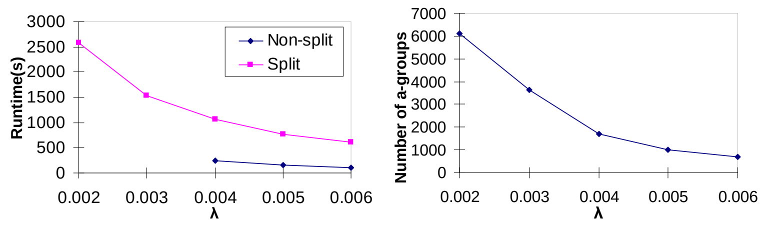

Fig. 2. Runtime & Patterns: DS_2 at various support values.

The number of mined a-groups in the real dataset is small even on much lower support threshold. Interestingly, we find that antagonistic behavior is not

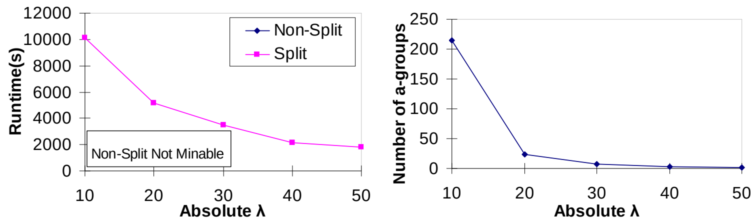

Fig. 3. Runtime & Patterns: Book ratings dataset at various support values.

so much apparent on item ratings. This might be the case since the objects rated are not “sensitive” items that tend to divide people into opposing groups.

Several interesting a-groups are discovered from the Amazon dataset by running the mining algorithm with absolute support (i.e., $\lambda \times |I|$) of 10 and σ=0.5. Out of 167 a-groups generated, 147 are of size 2, 18 of them are of size 3, and 2 of them are of size 4. We post-process to retain those with $aconf > 0.7$, and at least one user has (commonly-rated-items/ totally-rated-items) > 0.6.

ID	Antagonistic Groups	Commonly Ratings	Ratings by User 1	Ratings by User 2	Ratings by User 3
1	({Johnston},{Weissgarber})	12	56	13	-
2	({Johnston, Jump},{Weissgarber})	10	56	61	13
3	({Johnston, Hill},{Weissgarber})	10	56	106	13
4	({Leeper},{Weissgarber})	10	137	13	-
5	({Kern},{Sklarski})	14	452	22	-

Table 2. Interesting Examples from Amazon Book Rating Dataset

After post-processing, we note 5 of the most interesting a-groups. We select those having highest $aconf$ and average (common-item/total item) over all constituent users. They are represented in table 2. We select the first a-group and observe the following:

– *High antagonistic level*: We observe that the two users in the first a-group rated with a high level of antagonism. Among Jason Johnston’s 56 rated books, 12 have ratings opposite to the ratings by Luke Weissgarber. Similarly for Weissgarber, 12 of all his 13 rated books have ratings opposite to those by Johnston, which means more than 92% of Weissgarber’s ratings are opposite to Johnston’s. It is a significantly high figure.
– *Antagonistically rated books*: We found that for books with opposite ratings from Weissgarber and Johnston are some novels with similar story background. These books are clearly liked by Johnston but not by Weissgarber.
– *Antagonistically behaved users*: It is interesting that Weissgarber appears in four a-groups. His ratings are opposite to other 4 users for at least 10 books.

6 Conclusion & Future Work

In this study, we proposed a new pattern mining algorithm to mine for antagonistic communities. Our algorithm traverses the search space of possible antagonistic groups and uses several pruning strategies to remove search space containing no antagonistic pattern. We also propose a new variant of the algorithm that adopts a divide and conquer strategy in mining when the first algorithm becomes prohibitively expensive to run. A performance study is conducted on various synthetic datasets to show the scalability of our approach on various parameter values. We also mine from an Amazon book rating dataset. The result shows that antagonistic communities exists but are not particularly many or large in the Amazon dataset. In the future, we plan to investigate more "sensitive" datasets and further speed up the mining algorithm.

Acknowledgement. This work is supported by National Research Foundation of Singapore under project NRF2008IDM-IDM004-036. We would like to thank Paolo Massa for sharing his Epinions dataset. We would also like to thank Bing Liu for sharing the book ratings dataset.

References

1. *www.mysmu.edu/phdis2008/kuan.zhang.2008/agroup.pdf*, 2009.
2. R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of International Conference on Very Large Data Bases*, 1994.
3. D. Cai, Z. Shao, X. He, X. Yan, and J. Han. Community mining from multi-relational networks. In *PKDD*, 2005.
4. I. Dasgupta. 'living' wage, class conflict and ethnic strife. *Journal of Economic Behavior & Organization*, 2009. in press.
5. I. Dasgupta and R. Kanbur. Community and class antagonism. *Journal of Public Economics*, 91(9):1816–1842, Sep 2007.
6. J. Denrell. Why most people disapprove of me: Experience sampling in impression formation. *Psychological Review*, 112(4):951–978, 2005.
7. B. Ding, D. Lo, J. Han, and S.-C. Khoo. Efficient mining of closed repetitive gapped subsequences from a sequence database. In *ICDE*, 2009.
8. G. Flake, S. Lawrence, C. Giles, and F. Coetzee. Self-organization and identification of web communities. *Computer*, 35(3):66–71, 2002.
9. D. Gibson, J. Kleinberg, and P. Raghavan. Inferring web communities from link topology. In *Hypertext*, 1998.
10. M. Giles and A. Evans. The power approach to intergroup hostility. *The Journal of Conflict Resolution*, 30(3):469–486, Sep 1986.
11. J. Golbeck. Trust and nuanced profile similarity in online social networks. *ACM TWeb*, 3(4):1–33, 2009.
12. R. Guha, R. Kumar, P. Raghavan, and A. Tomkins. Propagation of trust and distrust. In *WWW*, 2004.
13. J. Kunegis, A. Lommatzsch, and C. Bauckhage. The slashdot zoo: Mining a social network with negative edges. In *WWW*, 2009.
14. S. Labovitz and R. Hagedorn. A structural-behavioral theory of intergroup antagonism. *Social Forces*, 53(3):444–448, Mar 1975.
15. H. Liu, E.-P. Lim, H. Lauw, M.-T. Le, A. Sun, J. Srivastava, and Y. Kim. Predicting trusts among users of online communities: an epinions case study. In *EC*, 2008.
16. M. McPerson, L. Smith-Lovin, and J. Cook. Birds of a feather: Homophily in social networks. *Annual Review of Sociology*, 27(3):415–444, 2001.
17. Tolsma, Jochem, N. D. Graaf, and L. Quillian. Does intergenerational social mobility affect antagonistic attitudes toward ethnic minorities? *British Journal of Sociology*, 60(2):257–277, June 2009.
18. J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *ICDE*, 2004.