Mining Antagonistic Communities from Social Networks

Kuan Zhang
School of Information Systems
Singapore Management University
kuan.zhang.2008@smu.edu.sg

David Lo
School of Information Systems
Singapore Management University
davidlo@smu.edu.sg

Ee-Peng Lim
School of Information Systems
Singapore Management University
eplim@smu.edu.sg

ABSTRACT

During social interactions in a community, there are often sub-communities that behave in an opposite way with each other. These antagonistic sub-communities could correspond to various interesting information from highlighting groups of people with opposite taste with each other to factions within a community that exhibit distrust with each other. Based on a set of interactions in time, we develop a novel pattern mining approach that extract for a set of antagonistic sub-communities from an interaction database. Based on a set of user specified thresholds, we extract a set of pairs of sub-communities that behave in opposite way 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 experiment on the scalability of the algorithm when the size of the dataset is varied in multiple dimensions. Case studies on Epinion dataset and a 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. In a community, members support or upheld a particular interest, idea or even conviction. Due to the dynamics of human social interaction, often *antagonistic groups*, i.e., two groups that consistently differ in opinions are formed. Some examples of such group are many, ranging from the various social groups that held differing opinion on various topics including industrialism vs. conservation, to formal organizations that are opposing to each other. Members of opposing groups held similar opinions within the group but different with members of the opposite group.

Opposing groups and their nature have been studied in the sociology domain [19, 6, 5, 15, 11, 7]. Understanding the formation of these groups and wide-spread-ness of opposing communities are of research interest. They could potential signify signs of disunity in the 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: e.g., how the antagonistic community is formed, why they are formed, how does the antagonistic communities grow over time, when do the communities stop being antagonistic, etc.

Copyright is held by the author/owner(s).
a conference, 2010, ...
.

Aside from enriching our study on dynamics of social interactions, information on group 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 review one could ignore, etc.

With the advent of Web 2.0, people's opinion are more readily expressed and available to be analyzed. Various forums and blogs provide opinions and views on multitudes of issues. Some systems like Epinions [2] and Amazon, also provides a mechanism to rate items, or even to rate the reviews of the items themselves. These provide a wealth of data ready to be analyzed.

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/low or positive/negative. From these database, we would like to extract a set of users that are antagonistic over enough number of common items/issues with a high likelihood. The pattern that we mine would identify 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 explore the search space of all possible opposing communities and prune 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 opposing pattern would have many sub-pattern 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 [3]. These 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 investigated two real dataset which includes rat-

ing data from Epinion and another book rating data from Amazon. We extract antagonistic communities from them. The algorithm is shown to be able to run on real data and extract antagonistic communities. A few hundred of communities are mined from the dataset.

The contribution of our work is 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 interaction, 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 two real datasets shedding light to the extent of consistent antagonistic behavior on 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. Section 5 describes our synthetic data generation algorithm. Experiments and case studies are described in Section 6. We finally conclude and describe future work in Section 7.

2. RELATED WORK

There have been a number of studies on finding communities in social network [4, 10, 9]. In this study we enrich past studies by discovering not communities but opposing ones. 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 [18]. 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 [12]. In this sense, our work enriches existing studies on homophily and trust [13, 16, 14].

In the sociology, economics, and psychology communities, the concept of inter-group antagonism have been studied by various work [19, 6, 5, 15, 11, 7]. We extend this interesting research question by providing a computation tool to automatically identify opposing communities from history of their behavior. We believe our tool could potentially be used to help sociologist in understanding the behavior of communities from the wealth of available data of user interactions in Web 2.0.

On the algorithm end our algorithm belongs to a family of pattern mining algorithm. There have been many of these algorithm used to mine for association rules [3, 21], frequent sequences [23, 20], frequent repetitive sequences [17, 8], frequent graphs [22], etc. The closest to our study is the body of work on association rule mining [3]. 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/common rated items but oppose each other with high likelihood. The problem is inherently different. We show that a similar apriori-like anti-monotonicity property holds but employ a different algorithm to directly mine for antagonistic communities. Similar to the work in [21], we do not report all frequent and confident groups rather only the *closed* ones.

3. ANTAGONISTIC GROUP

In this section, we formalize how history of user opinions or views are captured. We follow by a formal definition of antagonistic group and its associated notion of support and confidence. We then describe a property on support that is used by the mining process. A formal problem statement is given at the end of the section.

We formalize past histories of user social interactions in terms of ratings to objects/views/ideas/items. 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, low **rating polarities** and middle 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, $4-5$ to be of high rating polarity and 3 to be middle rating. We formalize our input as a database of ratings, defined in Definition 3.1. 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 3.1. *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, in which 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 3.2. ***(Opposing Group):*** *Let U_i and U_j be two disjoint sets of users. (U_i, U_j) is called an opposing group (or simply, o-group).*

The number of common ratings between the two sets of users U_i and U_j of an opposing group 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 three 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.

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 3.3. ***(Frequent Opposing Group):*** *An opposing group (U_i, U_j) is a frequent opposing group (or, frequent o-group for short) 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 o-groups satisfy the important Apriori property as stated below. Due to space constraint, we move the proof to [**?**].

PROPERTY 3.1. ***(Apriori Property of Freq. O-group):*** *Every size $(k-1)$ o-group (U'_i, U'_j) subsumed by a size-k frequent o-group (U_i, U_j) is a frequent o-group.*

PROOF. Assume an o-group g_{k-1} is not frequent. This would mean $\frac{count(g_{k-1})}{|I|} < \lambda$ or $\frac{antcount(g_{k-1})}{|I|} < \lambda \times \sigma$. If an user u_k is added to either user set of this o-group, we call the resulting o-group $g_{k-1} \cup u_k$. $g_{k-1} \cup u_k$'s count can not be more than count(g_{k-1}) and its antcount can not be more than antcount(g_{k-1}). This is because the count is calculated by intersecting the g_{k-1}'s user set's rating and the u_k's rating ,count($g_{k-1} \cup u_k$) $\leq$ min{count(g_{k-1}),count(u_k)}, and similarly, the antcount is calculated by intersecting g_{k-1}'s user set's rating and u_k's rating such that intersected rating have opposite polarity, antcount($g_{k-1} \cup u_k$) $\leq$ antcount(g_{k-1}). Therefore, $\frac{count(g_{k-1} \cup u_k)}{|I|} < \lambda$ or $\frac{antcount(g_{k-1}) \cup u_k}{|I|} < \lambda \times \sigma$; that is $g_{k-1} \cup u_k$ is not frequent neither. □ □

DEFINITION 3.4. ***(Antagonistic Group):*** *An opposing group (U_i, U_j) is an antagonistic group (or, a-group for short) if it is a frequent o-group and $aconf(U_i, U_j) \geq \sigma$.*

DEFINITION 3.5. ***(Closed Antagonistic Group):*** *An 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 3.1. *Consider the example rating database in Table 1. Suppose $\lambda = 0.5$ and $\sigma = 0.5$. Both (a, d) and (a, bd) are 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 $antcount(a, b) \neq antcount(a, bc)$.*

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 2. 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 $(count(d, e) = 2) > (count(d, ef) = 1)$.

Table 1: Example Rating Database 1 - DB_{EX1}

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

Table 2: Example Rating Database 2

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

DEFINITION 3.6. ***(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. 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 frequent size-2 o-group candidates and determine which of them are frequent o-groups. In the next pass, with the set of frequent o-groups found in the previous pass, we generate new potential frequent o-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 which of the candidates are frequent o-groups, and they are used to generate frequent o-groups for the next pass. After that, we filter the previous frequent o-group set with the newly generated frequent o-group set by removing non-closed frequent o-groups. Then we move on to the next pass. This process continues until no larger frequent o-groups are found. After successful mining of all frequent o-groups, we derive the a-groups from them.

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 o-groups of size k and C_k a candidate set of size k for frequent o-groups checking. The first two lines of the algorithm derives size-2 candidates to get the frequent size-2 o-groups. It forms the base for subsequent processing. A subsequent pass, say pass k, consists of three phases. First, at line 5, the frequent o-groups in L_{k-1} found in $k-1$ pass are used to generate the candidate frequent o-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 struc-

Input: λ; σ; rating database
Output: closed a-group of all size

```
1  L_1 = frequent user set;
2  C_2 = {({u_i}{u_j})|i < j, u_i ∈ L_1, u_j ∈ L_1};
3  for k = 2;k ≤ |U| and |L_{k-1}| ≠ 0; k++ do
4      if k > 2 then
5          C_k=antGrpMining-gen(L_{k-1});
6      end
7      root← buildHashTree(k,C_k);
8      foreach record t ∈ 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 ∈ C_k| count(g_k)/|I| ≥ λ and antcount(g_k)/|I| ≥ λ × σ};
15     L_{k-1}=prune(L_{k-1}, L_k);
16 end
17 G={g ∈ ∪_k L_k| antcount(g)/count(g) ≥ σ};
18 Output G;
```

Algorithm 1: Closed Antagonistic Group Mining Algorithm – clmine(λ,σ,DB_R)

ture described in [3] 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 o-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 frequent o-groups to remove non-closed frequent o-groups (line 15). After all the passes, the a-groups are determined from the frequent o-group set (line 17). The following subheadings zoom into the various components of the mining algorithm in more detail.

Input: size-$(k-1)$ frequent o-group set L_{k-1}
Output: size-k candidate frequent o-group set

```
1  foreach p, q ∈ L_{k-1} do
2      g_k ← merge(p, q);
3      add g_k to C_k;
4      forall (k − 1)-subsets s of g_k do
5          if s¬ ∈ L_{k-1} then
6              delete g_k from C_k;
7          end
8      end
9  end
10 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-$(k-1)$ o-groups as input. It returns a superset of all frequent size-k o-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 frequent o-group consisting of the common sub-community and the two differing members. We add the

Input: a-group $(\{U_i\},\{U_j\})$; a-group $(\{U'_i\},\{U'_j\})$
Output: merged result of the two input a-group

```
1  if U_i=U'_i and diff(U_j,U'_j)=1 then
2      return (U_i,U_j ∪ U'_j);
3  end
4  if U_j=U'_j and diff(U_i,U'_i)=1 then
5      return (U_i ∪ U'_i,U'_j);
6  end
7  if U_i=U'_j and diff(U_j,U'_i)=1 then
8      return (U_i,U'_i ∪ U_j);
9  end
10 if U'_i=U_j and diff(U_i,U'_j)=1 then
11     return (U_i ∪ U'_j,U_j);
12 end
13 return null;
```

Algorithm 3: $merge((\{U_i\},\{U_j\}),(\{U'_i\},\{U'_j\}))$

candidate frequent o-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 3.1. From the property, if g_k is a frequent o-group, all its $(k-1)$ subsets must be frequent o-groups. In other words, if any one $(k-1)$ subset of a frequent o-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.

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

PROOF. From property 3.1, any subset of a frequent o-group must also be frequent o-groups. Hence, if we extend each frequent o-group in $L_{k-1}(k \geq 3)$ with all possible users and then delete all those whose $(k-1)$-subsets are not in L_{k-1}, we will be left with a superset of the frequent o-groups in L_k. In the Algorithm 2, first we perform a merge process which is equivalent to extending L_{k-1} with all possible users in the database (line 2) and then at lines 4-8, we delete o-groups whose $(k-1)$-subsets are not in L_{k-1}. Thus after we merge the and deletion steps, all frequent o-group must be a subset of the returned candidate set. □

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

EXAMPLE 4.1. *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 o-group (u_1u_5, u_2u_3) because the o-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 frequent o-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

```
Input: k:level of the tree; C_k:size-k candidate set
Output: root of the tree
1  create new node root;
2  foreach candidate c_i in C_k do
3      sort users in c_i by userID;
4      tempNode← root;
5      foreach user u in c_i do
6          if tempNode has descendant d labeled u then
7              tempNode← d;
8          else
9              create node d with label u;
10             set d as descendant of tempNode;
11             tempNode← d;
12         end
13         if u is the last user in c_i then
14             set tempNode to leaf;
15             add c_i to the leaf;
16         end
17     end
18 end
19 return root;
```

Algorithm 4: buildHashTree(k,C_k)

```
Input: t:item in database; root:root of hashtree
Output: set of candidate contained in t
1  C_t ← ∅;
2  pointerRef ← empty vector of node;
3  pointerRefSuffix ← empty vector of node;
4  add root to pointerRef;
5  foreach rater u of t do
6      foreach node node_i in pointerRef do
7          if node_i has descendant d_i with label u then
8              node_i's descendant count−−;
9              if node_i's descendant count==0 then
10                 remove node_i from pointerRef;
11             end
12             if d_i is leaf then
13                 add a-groups stored in d_i to C_t;
14             else
15                 add d_i to pointerRefSuffix;
16             end
17         end
18     end
19     append pointerRefSuffix to pointerRef;
20     pointerRefSuffix← empty vector of node;
21 end
```

Algorithm 5: subset(t,root)

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

```
Input: frequent o-group set L_{k-1}; frequent o-group set
       L_k
Output: closed frequent o-group set of size k
1  foreach g_k ∈ L_k do
2      foreach g_{k-1} ∈ L_{k-1} do
3          if g_{k-1} ⊆ g_k and count(g_{k-1})=count(g_k) and
           antcount(g_{k-1})= antcount(g_k) then
4              remove g_{k-1} from L_{k-1};
5          end
6      end
7  end
8  return L_{k-1};
```

Algorithm 6: ncfilter(L_{k-1}, L_k)

Correctness of the algorithm. The correctness of the algorithm is guaranteed by the following Theorems 4.1 & 4.2. The theorems guarantee that everything reported are correct and a complete set of closed antagonistic groups are reported.

THEOREM 4.1. *Mined a-group set G contains all the closed a-group.*

PROOF. Let's assume an arbitrary a-group g to be closed, hence, $\frac{count(g)}{|I|} \geq \lambda$ and $\frac{antcount(g)}{count(g)} \geq \sigma$. By multiplying the two, g also fulfills $\frac{antcount(g)}{|I|} \geq \lambda \times \sigma$. By definition 3.3, g is a frequent o-group. As proved in lemma 1, g will be in $C_{|g|}$. g can be captured by line 5 of algorithm 1. As g fulfills both $\frac{count(g)}{|I|} \geq \lambda$ and $\frac{antcount(g)}{|I|} \geq \lambda \times \sigma$. g will be

Table 3: **Example Rating Database 1**

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

captured by line 14 of algorithm 1. Since g is closed, g will retain in $L_{|g|}$ after step line 15 of algorithm 1. And finally, due to $\frac{antcount(g)}{count(g)} \geq \sigma$, g will be added to G by line 17 of algorithm 1. Hence, every closed a-group will be contained in G. □

THEOREM 4.2. *Mined a-group set G contains only closed a-groups.*

PROOF. Suppose an o-group $g \in G$ is not an a-group, that is, $\frac{count(g)}{|I|} < \lambda$ or $\frac{antcount(g)}{count(g)} < \sigma$. From the statement line 17 of algorithm 1, we can know $g \in \bigcup_k L_k$, and $\frac{antcount(g)}{count(g)} \geq \sigma$. However, every o-group g_k in $\bigcup_k L_k$ has $\frac{count(g_k)}{|I|} \geq \lambda$. Thus $\frac{count(g)}{|I|} \geq \lambda$. It contradicts with our condition that $\frac{count(g)}{|I|} < \lambda$ or $\frac{antcount(g)}{count(g)} < \sigma$. Thus, g must be an a-group. Hence, G contains only a-groups. The closure property of G can be guaranteed by line 15 of algorithm 1. Every a-group in G will be checked to filter out the non-closed one. The filtering method will not leave any non-closed a-group in G, by Algorithm 6. Hence G contains only closed a-group. □

Scalability Variant: Divide and Conquer Strategy. At times the memory required to generate all the candidates could be prohibitive. If there are too many L2 patterns, storing all of them in 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 4.1 (**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 4.2. *To illustrate, consider the first entry $itm-usr$ in the example rating database shown in Table 1. The first entry $itm-usr$ contains users a, b and d: $a \in itm-usr$, $b \in itm-usr$, and $d \in itm-usr$.*

DEFINITION 4.2 (**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:*

$$DB_R[u_i] = \{itm - usr | u_i \in itm - usr \wedge itm - usr \in DB_R\}$$

EXAMPLE 4.3. *To illustrate, consider the example rating database shown in Table 1. Projection of the database with respect to user d is shown in Table 3.*

Having defined the above two definitions we now define a lemma to divide and conquer the mining process in Lemma 2.

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:*

$$Cm(\lambda, \sigma, DB_R) = \bigcup_{u_i \in U_{Set}} \{g | u_i \in g \wedge g \in Cm(\frac{\lambda \times |DB_R|}{|DB_R[u_i]|}, \sigma, DB_R[u_i])\}$$

PROOF. An entry in the database could only be counted as an additional support to an a-group containing user u_i iff the entry contains an item u_i. Hence, partitioning the database with respect to a user u_i would return the relevant portion of the database that is relevant to u_i. The absolute support count of a-group containing u_i in the partitioned database $DB_R[u_i]$ would be the same as that in the original database DB_R. All a-groups reported in $Cm(\lambda_{ui}, \sigma, DB_R[u_i])$ (λ_{ui} denotes $\frac{\lambda \times |DB_R|}{|DB_R[u_i]|}$) that contains u_i would have the correct support. All a-groups containing u_i should be output by $Cm(\lambda_{ui}, \sigma, DB_R[u_i])$. However, nothing is guaranteed for a-groups that do not contain u_i in the set returned by $Cm(\lambda_{ui}, \sigma, DB_R[u_i])$ – they could have a wrong support. They should be dropped then.

Hence, it could be easily seen that the union of the mining results over the partitions with various u_is, with removal of results that does not contain u_i would be equal to the results returned by the mining operation on the entire dataset. □

Based on Lemma 2, we algorithm to perform divide and conquer is shown in Algorithm 7. The algorithm partitions the database one item at a time and subsequently calls the original closed antagonistic group mining algorithm defined in Algorithm 1. We also have a theorem to guarantee that the mined result is correct and a complete set of a-groups are mined by Algorithm 7 stated in Theorem 4.3.

THEOREM 4.3. *Algorithm 7 would return a complete set of closed a-groups and all returned a-group would be closed.*

PROOF. From Theorems 4.1 & 4.2 and Lemma 2 it is easy to see that the above theorem holds. □

Note that the divide and conquer algorithm would reduce memory costs but however would potentially increase the runtime cost since the database would now need to be scanned more number of times. In our experiment, we would employ Algorithm 1 unless the algorithm is prohibitively expensive to run. In this case, we run Algorithm 7.

Input: λ; σ; rating database
Output: 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 $\quad G = G \cup \{ag | u_i \in ag \wedge ag \in$ Clagmine$(\frac{\lambda \times |DB_R|}{|DB_R[u_i]|}, \sigma, DB_R[u_i])\}$;
5 **end**
6 **Output** G;

Algorithm 7: clagmine-partitional(λ,σ,DB_R)

5. SYNTHETIC DATASET

To investigate the scalability of our mining solution on various data characteristic, similar to the evaluation measure applied to mining association rules [3, 23, 20], we develop a data generator engine. The engine take some a sample dataset and a set of user-given inputs. The procedure of synthetic data generation is shown in algorithm 8

The detailed generation of synthetic data is as follows:

- Generate the set of maximal potential large anti-groups L:(line 1-10)
  The anti-groups in L have an average size of N_G. For each anti-group g in L, we pick the size of g by a Poisson distribution with mean = N_G. If the size of g picked is larger than $|U|$, we resample. Let the i-th selected size be N_i. For the 1st anti-group, we randomly select a group of users (denoted by G) from U and add the anti-group with equal splits of users in G into the opposing user sets (U_1, U_2) to L. For each subsequent anti-group G, we select $N_i * q$ users (randomly) from the previous generated anti-group denoted by G' and $N_i * (1 - q)$ users (randomly) from $(U - G')$. The q value is picked from an exponential distribution with mean = 0.5. Again, if q is larger than 1, we resample. Add the G with equal splits of users into the opposing user sets to L.

- Determine the probabilities of anti-groups:(line 11-17)
  We associate a probability p_i of picking each anti-group G_i in L. The p_i's is decided using an exponential distribution with mean = 1. P_i's are then normalized such that their sum = 1. Hence, we have N_L different p_i's.

- Assign anti-groups to each item:(line 18-27)
  For each item k, pick a number of users M_k to be included for the item. The M_k is selected using a Poisson distribution with mean equals $|U| * s$ where s is a number between 0 and 1. We pick a set of anti-groups $GSet_k$, a subset of L, using a biased dice with N_L sides and each side having a probability of p_i. Note that $M_k \leq$number of users in $GSet_k$. If $M_k <$ number of users in $GSet_k$, we can assign only subset of users from an anti-group in $GSet_k$ to item k. The assigned anti-group should not overlap on an user, which means a same user i can not exist in two anti-groups in $GSet_k$ concurrently.

- Assign ratings to each item:(line 28-40)
  The assignment of rating(i, j) (user i to item j) can be done as follow. We only consider three ratings, high (hi), low (lo) and middle (*middle*). For an anti-group (U_1, U_2) associated with item k. We assign all rating(i, k) as H for all i belongs to U_1, and all rating(j, k) as L for all j belongs to U_2. We continue the process until all the anti-groups associated with item k is processed, we assign M to the rest of rating(l, k)'s.

The next section reports a performance study experiment using data generated by this data generator, along with two case studies on real datasets.

Input: N_G: average size of maximal potential large a-group;N_L:number of maximal potential large a-group;U:user set;I:item set;s:selecting probability of each user
Output: ratings of users in U to items in I

```
1  L ← ∅;
2  N_1 ← poisson(N_G);
3  g_1 ← randomly pick N_1 users from U;
4  add g_1 to L;
5  for i from 2 to N_L do
6      N_i ← poisson(N_G);
7      q ← expo(0.5);
8      g_i ← {randomly pick N_i × q users from
       g_{i-1}} ⋃ {randomly pick N_i × (1 − q) users from
       (U-g_{i-1})};
9      add g_i to L;
10 end
11 for i from 1 to N_L do
12     p_{g_i} ← expo(1);
13 end
14 sum ← Σ_{i=1}^{N_L} p_{g_i};
15 for i from 1 to N_L do
16     p_{g_i} ← p_{g_i}/sum;
17 end
18 foreach item k do
19     GSet_k ← ∅;
20     M_k ← poisson(|U| × s);
21     while number of users in GSet_k < M_k do
22         randomly pick g_k from L and add g_k to GSet_k;
23         if number of users in GSet_k > M_k then
24             retain partial users from last picked g_k such
               that number of users in GSet_k = M_k;
25         end
26     end
27 end
28 foreach item k do
29     foreach g_k in GSet_k, suppose g_k =(U_1, U_2) do
30         forall user u from U_1 do
31             rating(u, k)← hi;
32         end
33         forall user u from U_2 do
34             rating(u, k)← lo;
35         end
36     end
37     forall user u in U − {users in GSet_k} do
38         rating(u, k)← middle;
39     end
40 end
```

Algorithm 8: Synthetic Data Generation Procedure

6. EXPERIMENTS & 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 two case studies from two real dataset: Epinions and Amazon book rating dataset.

6.1 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 (i.e., U × s, 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
DS_3	I=100, U=10, P=30, N_G=6, N_L=2
DS_4	I=10, U=50, P=20, N_G=6, N_L=2

The experiments for datasets DS_1 to DS_4 are run with σ=0.7.

The result for dataset DS_1 when varying the support threshold from 0.002 to 0.006 is shown in Figure 1. The first graph show the runtime needed to execute the algorithm at various support thresholds. "Non-split" and "Split" correspond to Algorithm 1 &7 respectively. We only include 3 data points for "Non-split", as mining at lower thresholds are too long to complete. The second graph shows the numbers of a-groups being mined by the algorithm at various support thresholds. Finally, we plot a cumulative graph showing the number of patterns of size greater than or equal to k at support threshold equals to 0.002.

The result shows that the time taken grows larger when the support threshold is reduced. This growth is accompanied by the growth in terms of 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 threshold with σ=0.7 are shown in Figure 2.

For the third dataset we decrease the number of users while increase the expected number of users rating an item. The results for various support thresholds are shown in Figure 3.

For the fourth dataset we consider a fewer number of items and a larger number of users. The results for various support thresholds are shown in Figure 4.

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, the more expensive it is to mine.

6.2 Case Studies

6.2.1 Real Data Statistics

For the case study, first we consider a rating data crawled from Epinions [2] by Paolo Massa from [1]. The dataset is a result of a 5-week crawl in November/December 2003. It contains 49,290 users that rates 139,738 different items in 664,824 reviews. The experiment is run with σ=0.5. The result for the first case study on Epinions data is shown in Figure 5.

For the second 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 also run with σ=0.5. The result for the second case study on Amazon book ratings dataset is shown in Figure 6.

It could be noted that the number of mined a-group in the two real datasets are smaller even on much lower support threshold. Interestingly, we find that antagonistic behavior is not so much apparent on data ratings item in Epinions or books. A "fan-like" attitude does not seem apparent from mining the two datasets. This might be the case since the objects rated are not "sensitive" items that tend to divide people into opposing groups.

6.2.2 Detailed Examples

In this section, we present details of several interesting examples found by our mining algorithm. This illustrates how our algorithm can effectively mine a-groups on real dataset.

Several interesting a-groups are discovered from the Amazon book rating data set. They are obtained by running the mining algorithm on Amazon data set with absolute support threshold 10 and σ=0.5. The program runs for 930 second with 167 a-groups generated. 147 of the a-groups are of size 2, 18 of them are of size 3 and 2 of them are of size 4. We post-process the a-group with the following criteria:

- Aconf: Only retain a-groups with aconf $>$ 0.7.
- commonly-rated-item/totally-rated-item:Retain a-groups if at least one user in the a-group has (number of commonly-rated-items/number of totally-rated-items of the user) $>$ 0.6.

After postprocessing, we noted 5 of the most interesting a-groups. They have highest aconf and average (common-item/total item) for all user. They are represented in table 4. Take the first a-group as an example, we examine it as follows:

- *High antagonistic level*: We observe that the two users in the first a-group rated in high level of antagonism. Among Jason Johnston's 56 rated books, 12 are of opposite ratings to the ones rated by Luke Weissgarber. Similarly for Luke Weissgarber, 12 of all its 13 rated books are rated against Jason Johnston, which means more than 92% of Luke Weissgarber's ratings are against its component. It is a significantly high figure.
- *Antagonistically rated books*: Based on our mining result, we examined the Amazon website. We found that for books: "Armageddon", "The Remnant: On the Brink of Armageddon", "Desecration: Antichrist Takes the Throne", "The Mark: The Beast Rules the World", "The Indwelling: The Beast Takes Possession", "Assassins", "Apollyon: The Destroyer Is Unleashed", "Soul Harvest: The World Takes Sides", "Nicolae: The Rise of Antichrist", "Tribulation Force: The Continuing Drama of Those Left Behind", "Left Behind: A Novel of the Earth's Last Days" and "Glorious Appearing: The End of Days", Jason Johnston rated 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 and 4 for the books respectively and Luke Weissgarber rated 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 and 1 respectively. It can be noted that the ratings of the 12 books are indeed against each other.

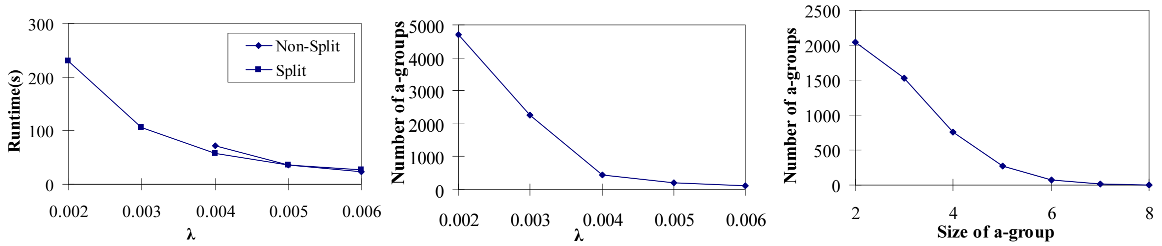

Figure 1: Runtime & Patterns: DS_1 at various support values.

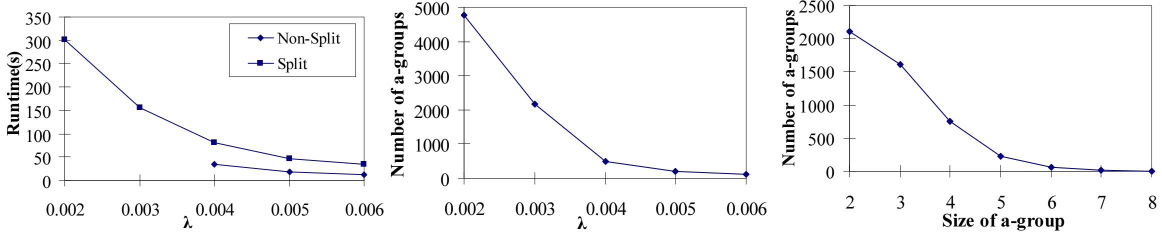

Figure 2: Runtime & Patterns: DS_2 at various support values.

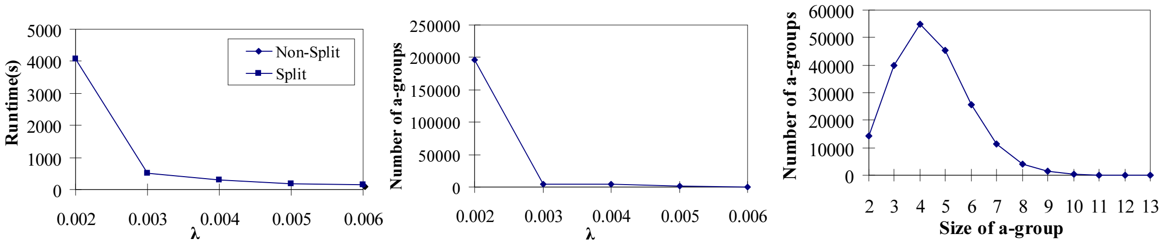

Figure 3: Runtime & Patterns: DS_3 at various support values.

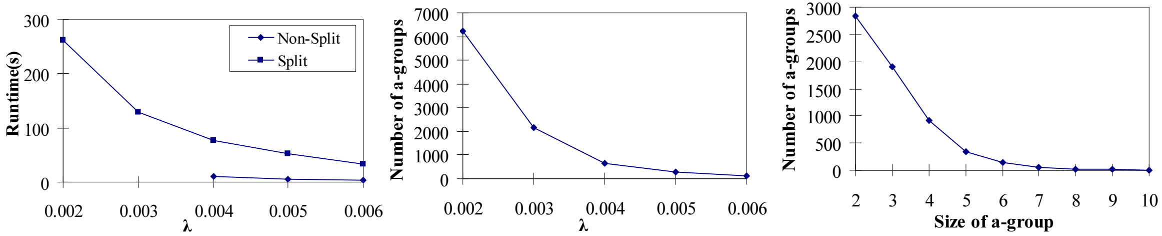

Figure 4: Runtime & Patterns: DS_4 at various support values.

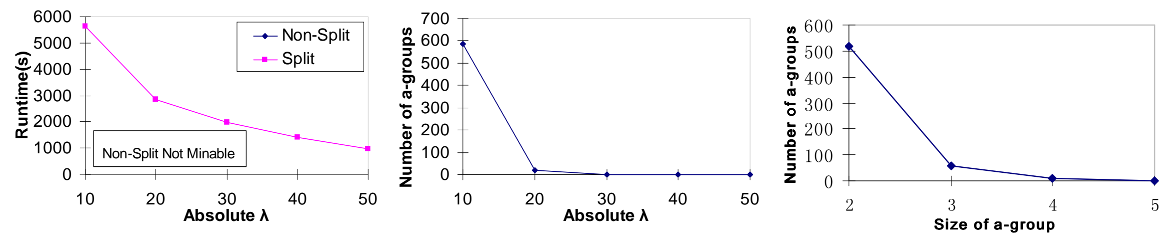

Figure 5: Runtime & Patterns: Epinions dataset at various support values.

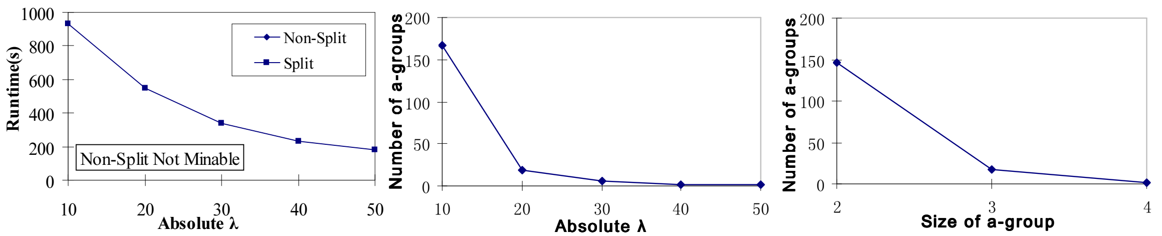

Figure 6: Runtime & Patterns: Book ratings dataset at various support values.

There is a high tendency that the books liked by Jason Johnston are disliked by Luke Weissgarber.

Table 4: **Interesting Examples from Amazon Book Rating Dataset**

ID	Antagonistic Group	Commonly Rated	Totally Rated by 1st User(% of commonly rated)	Totally Rated by 2nd User(% of commonly rated)	Totally Rated by 3rd User(% of commonly rated)
1	({Jason Johnston},{Luke Weissgarber})	12	56(21%)	13(92%)	-
2	({Jason Johnston, K.Jump},{Luke Weissgarber})	10	56(17.8%)	61(16.4%)	13(76.9%)
3	({Jason Johnston, C.Hill},{Luke Weissgarber})	10	56(17.8%)	106(9.4%)	13(76.9%)
4	({Jeffrey Leeper},{Luke Weissgarber})	10	137(7.3%)	13(76.9%)	-
5	({Konrad Kern},{T.M.Sklarski})	14	452(3.1%)	22(63.6%)	-

- *Antagonistically behaved user*: It is interesting to look at Luke Weissgarber, who appears four times in the five interesting a-groups. He tends to rate books against what others rate. His ratings are opposite to other four users on at least 10 books. It sheds some lights on the difference between Luke Weissgarber's judgement with normal people on books. It is interesting to research further on Luke Weissgarber's behavior.

7. CONCLUSION & FUTURE WORK

Antagonistic communities are important research interest well studied in sociology, psychology, economics. As a first step to understand the nature of antagonistic communities, an approach to automatically mine for these communities from ratings dataset storing user views/opinions/convictions on certain objects/items/subjects would be very useful. In this study, we proposed a new pattern mining algorithm to directly mine for antagonistic communities from rating database. Our algorithm traverse the search space of possible antagonistic groups and use several pruning strategies to remove bad search space containing no patterns. We also propose a new variant of the algorithm that perform a divide and conquer process in case 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. Two case studies have also been performed on the Epinions and book rating datasets. The result shows that antagonistic communities exists but are not particularly many or large on both datasets.

In the future, we plan to collect information from more real datasets that are more "sensitive" and tends to induce antagonistic groups to be formed. We also plan to improve the speed of the algorithm further so that even larger datasets could be analyzed more efficiently. An incremental mining process is also interesting to investigate as ratings database might change overtime.

Acknowledgement. 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.

8. REFERENCES

[1] Downloaded Epinions Dataset - Trustlet. http://www.trustlet.org/wiki/Downloaded_Epinions_dataset/ratings_data.txt.bz2.
[2] Review from Epinions. http://www.epinions.com/.
[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of International Conference on Very Large Data Bases*, 1994.
[4] D. Cai, Z. Shao, X. He, X. Yan, and J. Han. Community mining from multi-relational networks. In *PKDD*, 2005.
[5] I. Dasgupta. 'living' wage, class conflict and ethnic strife. *Journal of Economic Behavior & Organization*, 2009. in press.
[6] I. Dasgupta and R. Kanbur. Community and class antagonism. *Journal of Public Economics*, 91(9):1816–1842, Sep 2007.
[7] J. Denrell. Why most people disapprove of me: Experience sampling in impression formation. *Psychological Review*, 112(4):951–978, 2005.
[8] B. Ding, D. Lo, J. Han, and S.-C. Khoo. Efficient mining of closed repetitive gapped subsequences from a sequence database. In *ICDE*, 2009.
[9] G. Flake, S. Lawrence, C. Giles, and F. Coetzee. Self-organization and identification of web communities. *Computer*, 35(3):66–71, 2002.
[10] D. Gibson, J. Kleinberg, and P. Raghavan. Inferring web communities from link topology. In *Hypertext*, 1998.
[11] M. Giles and A. Evans. The power approach to intergroup hostility. *The Journal of Conflict Resolution*, 30(3):469–486, Sep 1986.
[12] J. Golbeck. Trust and nuanced profile similarity in online social networks. *ACM TWeb*, 3(4):1–33, 2009.
[13] R. Guha, R. Kumar, P. Raghavan, and A. Tomkins. Propagation of trust and distrust. In *WWW*, 2004.
[14] J. Kunegis, A. Lommatzsch, and C. Bauckhage. The slashdot zoo: Mining a social network with negative edges. In *WWW*, 2009.
[15] S. Labovitz and R. Hagedorn. A structural-behavioral theory of intergroup antagonism. *Social Forces*, 53(3):444–448, Mar 1975.
[16] 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 *ACM Conference on Electronic Commerce*, 2008.
[17] D. Lo, S.-C. Khoo, and C. Liu. Efficient mining of iterative patterns for software specification discovery. *Proceedings of SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2007.
[18] 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.
[19] 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.

[20] J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *Proceedings of IEEE International Conference on Data Engineering*, 2004.

[21] J. Wang, J. Han, and J. Pei. Closet+: searching for the best strategies for mining frequent closed itemsets. In *KDD*, 2003.

[22] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, 2002.

[23] X. Yan, J. Han, and R. Afhar. CloSpan: Mining closed sequential patterns in large datasets. In *Proceedings of SIAM International Conference on Data Mining*, 2003.