Learning Instance-Level Constraints in Folksonomies for Semi-supervised Clustering using CHR

Maged Shalaby, Slim Abdennadher, Nada Sharaf, and Ghada Fakhry

Computer Science and Engineering Department
German University in Cairo
maged.shalaby@student.guc.edu.eg, {slim.abdennadher, nada.hamed, ghada.fakhry}@guc.edu.eg

Abstract. In our modern days, huge amount of information is available to be searched and browsed by simple users. However, due to the sheer volume of information available, the task of browsing this information becomes increasingly difficult. Clustering algorithms have emerged as an automated tool to organize information for easier browsing. In this paper, we employ constraint reasoning to reason about the different pairwise constraints present, such as must-link and cannot-link constraints. These constraints are used to aid in semi-supervised clustering algorithms, and they are used to guide the algorithm whether a pair of items should or should not be placed in the same cluster. This implementation was done using CHR, a declarative rule-based language. We prove via experimental evaluation that inferring instance-level constraints in the domain of folksonomies yields an improvement in clustering performance.

Keywords: constraint reasoning, semi-supervised clustering, CHR

1 Introduction

The modern rise of the World Wide Web has led to an explosion in the amounts of data available to users. It is now a tedious task to find items of interest in this growing web of data. Folksonomies have risen to be a possible solution to the recommendation problem. Folksonomies are websites where users can tag items (artists, pictures, etc..) with relevant tags. An example for tags for a photo of a sandy beach are sea and beach. Users can view, add and tag items of interest, and get personalized recommendations out of those systems.

Clustering techniques have emerged as one possible means of recommending items for easier searching and browsing. Among those clustering techniques, there is a new generation of algorithms known as semi-supervised algorithms.

Semi-supervised algorithms utilize pairwise and group constraints that aid in the clustering process. An example of pairwise constraints are must-link and cannot-link constraints that enforce that items must be in the same cluster or different clusters, respectively [14]. These constraints can be added manually by domain experts, or inferred from the available data. Inferring these constraints using procedural languages is more difficult than using rule-based languages.
In this paper, we choose to use Constraint Handling Rules (CHR) to implement a constraint handler tuned for this problem. We found that CHR rules are easier to be written and implemented than procedural approaches due to their declarativity. CHR also gives a naturally incremental approach to adding constraints to semi-supervised algorithms. Therefore, we can add even more instance-level constraints than must-link and cannot-link constraints.

We employ constraint inference from folksonomies in our work to improve upon unsupervised clustering performance. We use tagging information about items to extract constraints. An example is generating must-link constraints for a pair of items if this pair has been tagged by many users. Another example is generating cannot-link constraints if the pair of items has less than a specific number of tags in common.

This paper is organized as follows: Section 2 reviews the relevant background about folksonomy systems, semi-supervised clustering, and CHR (our programming language of choice). Section 3 discusses the constraint inference module implemented. Finally, we conclude with a summary and discussion of future work in Section 4.

2 Background

In our work, we employ constraint inference to learn rules for folksonomies, to aid in semi-supervised clustering algorithms. We do this using CHR, a rule-based declarative language. Therefore we shall introduce first the notions of folksonomies, semi-supervised clustering and CHR.

2.1 Folksonomies

One arising application of Web 2.0, the new look for websites and webpages that started in the early 2000’s, is that of folksonomies. Folksonomies are the collection of users, items, tags in social tagging websites and applications, such as Flickr and del.icio.ous. Users can create user profiles, add or show interest in certain items (such as pictures, songs or bookmarks), and tag these items. Formally, a folksonomy is a tuple $F := (U, T, R, Y)$, where $U$, $T$, and $R$ are finite sets, whose elements are called users, tags and resources, respectively, and $Y$ is a ternary relation between them, i. e., $Y \subseteq U \times T \times R$ [8].

We will focus in our work on clustering items in folksonomies. An example of this is grouping photos together in a social photo-sharing website, such as Flickr. This approach is opposite to tag clustering, where tags are grouped into semantically meaningful groups, often used for tag disambiguation. Both are used in the process of recommendation, either recommending new items previously unseen to the users [11], or in recommending new tags to tag resources with.
2.2 Semi-supervised Clustering

In more recent times, there has been an interest in what is called constrained, or semi-supervised clustering. However, we ought to define supervised and unsupervised learning first.

In the field of machine learning, it is well known that there are two broad classes of problems, namely supervised and unsupervised learning. In supervised learning, given a set of training data, each item in the training data being compromised of $x$ features and a label. Given this training data, a model should be trained, that given a new previously unseen instance that has the $x$ features, will predict correctly its label. In unsupervised learning, the luxury of the labeled data does not exist. Instead, the goal is to find hidden patterns in the data from instances of only $x$ features.

As mentioned above, classical clustering is viewed as an unsupervised learning problem. However, with the use of some extra data inferred from domain knowledge or crowdsourced data, as we will do in our work, the problem of clustering can be categorized as a new class of problems: semi-supervised learning.

So what are the extra pieces of information that will transform our problem? The most commonly used are what are known as instance-level constraints. These can be categorized into either must-link constraints or cannot-link constraints between pairs of items. The presence of a must-link constraint forces, or at least advises, an algorithm to place the two items in the same cluster. The presence of a cannot-link constraint is the opposite, it advises an algorithm to place the two items in different clusters. The use of instance-level constraints can tremendously help any unsupervised clustering algorithm to produce better results, if only this information is somehow available.

There is an important difference between hard constraints and soft constraints. The difference between the two is that hard constraints force the algorithm to satisfy the constraint, placing the items in the same cluster for must-link constraints and placing the items in different clusters for cannot-link constraints. On the other hand, soft constraints are merely a guide to the algorithm that it is simply better to follow the constraints, with some sort of penalty for breaking the constraints.

It is possible to infer the instance link constraints, using some sort of domain knowledge. One example in folksonomies is generating must-link constraints for items that share at least $t$ tags in common. This sort of inference is possible in the domain of folksonomies, and is rather done using soft constraints, not hard constraints [3]. Another example in the fields of information retrieval and natural language processing is inferring must-link constraints from text documents that share $x$ n-grams [2].

There are more examples related to natural life. A third example is using GPS data for automatic lane detection, where there are two heuristics used in [14]. The first is for trace contiguity, where a must-link constraint is generated for all data points originating from the same vehicle in the absence of lane changes. The second is for maximum separation, which limits how far apart two points can be (perpendicular to the centerline) while still being in the same lane. If
two points are separated by at least four meters, then a cannot-link constraint is generated that will prevent those two points from being placed in the same cluster.

2.3 CHR

Constraint Handling Rules (CHR) is a high level language that was introduced for writing constraint solvers. CHR is a committed choice language based on multi-headed and guarded rules. CHR programs works by transforming constraints into simpler ones until they are solved. Over the last decades, CHR has matured into a general purpose language. CHR usually does not exist on its own, but is built on top of a host language. The most popular host language, and the one we shall use in our work, is Prolog, but there have also been CHR implementations for other languages, such as Haskell and Java. While originally intended for constraint solvers, CHR has extended to other problems and fields such as semantic web (description logics) [6], and natural language processing [5].

In CHR, two types of constraints are available. Built-in constraints provided through the host language (Prolog) and user-defined constraints that are defined through the rules of a CHR program [7]. Each rule consists of a head, which contains a set of constraints, an optional guard and a body. A rule is fired when constraints from the constraints store match the constraints in the rule head and the guard is satisfied. In general, there are three types of rules in CHR. The first type is simplification rule, such rules replace constraints by simpler ones. In simplification rules the head of the rule \( H_r \), which comes before the (\( \Leftrightarrow \)) are removed on executing the rule. (\( G \)) is the optional guard that consists of built-in constraints. The body (\( B \)) could contain both CHR and built-in constraints. A simplification rule thus has the following format:

\[
H_r \Leftrightarrow G \mid B.
\]

The second rule type is propagation rules. In a propagation rule, all head-constraints \( H^k \) are kept after the rule is executed adding the constraints in the body to the constraint store. This may cause further simplification afterwards. Propagation rules have the following format:

\[
H^k \Rightarrow G \mid B.
\]

The third type of rules, defined as simpagation rules, is a hybrid between the simplification and propagation rules. The elements of \( H^k \) are the constraints that are kept after the rule is executed. On the other hand, the constraints in \( H^r \) are removed after executing the rule. Simpagation rules take the following format:

\[
H^k \setminus H^r \Leftrightarrow G \mid B.
\]

An illustrative example can be viewed in the less-than-or-equal simple constraint solver The. The constraint \texttt{leq(X,Y)} represent the relation between the
two numbers $X$ and $Y$ respectively. The first rule adds a new inequality constraint between items $X$ and $Z$ given that $X \leq Y$ and $Y \leq Z$. Thus, the first rule is a propagation rule. The second rule denotes that if $X \leq Y$ and $Y \leq X$, then $X$ and $Y$ must be equal. Thus, the second rule is a simplification rule.

\[
\text{leq}(X,Y), \text{leq}(Y,Z) \Rightarrow \text{leq}(X,Z).
\]

\[
\text{leq}(X,Y), \text{leq}(Y,X) \Leftrightarrow X=Y.
\]

### 2.4 Tag Reasoning in CHR

A notable effort done previously is the work done by Sharaf et al [12] where a CHR tag reasoning system was provided. Due to its declarative nature, with CHR, different properties were easily encoded. The system was able to capture different properties including application specific features such as “absence of a tag in the presence of another”. This encodes the fact that in some of the cases if a tag is absent (fixed for example) in the presence of another tag (bug for example), a new tag (todo) should be added. Some, more general, rules were included to deal with inconsistent items (annotated with contradicting tags), and inconsistent users (annotating the same item with contradicting tags). With CHR, it was also possible to easily generate the co-occurrence graph of tags using the number of times the tags appear together.

### 3 Learning Instance-Level Constraints in Folksonomies

In this section we will discuss our constraint reasoning techniques, to infer must-link and cannot-link constraints in the field of folksonomies, and to reason about those constraints. We generate soft constraints here, where each constraint is a triple $I_1, I_2, V$ where $V$ is the value associated to the strength of the constraint. It is then easier to use them as soft constraints, or simply transform them to hard constraints by ignoring $V$.

To represent the tagging information, we use the CHR constraint \text{annotation}(U, I, T), which means that user $U$ has tagged item $I$ with tag $T$.

#### 3.1 Inferring Must-Link Constraints

The technique we apply to generate must-link constraints is to link pairs of items that have been co-tagged by $X$ users. To do this, we need to count how many users have commonly tagged this pair of items, and generate a must-link constraint only once this reaches a certain threshold. We show an example in the following rules:

\[
\text{annotation}(U, I_1, _), \text{annotation}(U, I_2, _), \text{possiblePair}(I_1, I_2)
\Rightarrow I_1 \not\leq I_2 \mid \text{userInCommon}(I_1, I_2, U).
\]

\[
\text{userInCommon}(I_1, I_2, U) \backslash \text{userInCommon}(I_1, I_2, U) \Leftrightarrow \text{true}.
\]
userInCommon(I1, I2, _, possiblePair(I1, I2) => I1 @< I2 | usersInCommon(I1, I2, 1).

usersInCommon(I1, I2, X1), usersInCommon(I1, I2, X2) <=> X is X1 + X2 | usersInCommon(I1, I2, X).

usersInCommon(I1, I2, X) \ possiblePair(I1, I2) => X > 10 | mustLink(I1, I2, 1).

In the first rule, we generate all users that have commonly tagged a pair of items. In the second rule, we remove duplicate instances of the same constraint. In the third and fourth rules, we count how many users in common a pair of items have. In the fifth rule, we generate a must-link constraint when the users in common has exceeded a certain threshold (here we set it to 10).

In the first rule, using a "_" in the place of the tag variable indicates that we don’t care whether the same tag is used or not. Also, the guard @< is used between I1 and I2 to force that I1 must be lexicographically smaller than I2. This is done to prevent the rule to fire for each pair of items twice, and to prevent it to fire for two annotations of the same item.

We use the constraint possiblePair(I1, I2) to stop the constraint generation procedure early if found that this pair of items has exceeded the threshold. This speeds up computation. The generation of the constraint possiblePair(I1, I2) is simple to do using either CHR or a procedural language. It can be generated by parsing an input file containing all the items, or from the constraint annotation(I, U, T).

To compare against a procedural approach, it would’ve needed a double for-loop for each pair of items, then computing the set intersection for the users that have tagged each item, which is a non trivial task. Instead, we need only five rules in the rule-based approach.

3.2 Inferring Cannot-Link Constraints

For generating cannot-link constraints between pairs of items, we count how many tags they have in common, and if it is less than a certain threshold, we generate a cannot-link constraint. Here are the rules for this task:

annotation(_, I1, T), annotation(_, I2, T), possiblePair(I1, I2) => I1 @< I2 | tagInCommon(I1, I2, T).

tagInCommon(I1, I2, T) \ tagInCommon(I1, I2, T) <=> true.

tagInCommon(I1, I2, _), possiblePair(I1, I2) => I1 @< I2 | tagsInCommon(I1, I2, 1).

tagsInCommon(I1, I2, X1), tagsInCommon(I1, I2, X2) <=> X is X1 + X2 | tagsInCommon(I1, I2, X).
tagsInCommon(I1, I2, X), possiblePair(I1, I2) ==> X < 9 |
cannotLink(I1, I2, 1).

tagsInCommon(I1, I2, X) \ possiblePair(I1, I2) <= X >= 9 |
okLink(I1, I2, 1).

okLink(I1,I2,1) \ cannotLink(I1, I2, 1) <= true.

The first rule is used to generate all tags in common between all pairs of items. This is equivalent to building the item-tag bipartite graph. The second rule is used to remove duplicate constraints. The third and fourth rules count how many tags in common are between this pair of items. The fifth rule generates a cannot-link constraint if the tags in common are less than a certain threshold. The sixth rule generates an ok-link constraint if the tags in common is greater than a certain threshold. The undesired cannot-link constraints are removed by the seventh rule.

To compare against a procedural approach, it would’ve needed two for loops to enumerate all pairs of items, and then computing the set intersection of the tags for each item. The rule based approach is again much simpler.

3.3 Generating Transitive Closure

One task often done before the start of a semi-supervised clustering algorithm is to find the transitive closure. The transitive closure is defined on a graph (V, E), and it is finding the reachability of all nodes in V. Therefore, for must-link and cannot-link constraints, we try to generate all possible constraints that can be inferred from the original constraint. For example, if I1, I2 must be linked together, and I2, I3 must be linked together, then it follows that I1, I3 must be linked together. We present now the two rules we used to generate the transitive closure:

mustLink(A, B, Cnt1), mustLink(B, C, Cnt2) ==> allDifferent(A,B,C), minimum(Cnt1, Cnt2, Cnt) | mustLink(A,C, Cnt).

mustLink(A, B, Cnt1), cannotLink(B, C, Cnt2) ==> allDifferent(A,B,C), minimum(Cnt1, Cnt2, Cnt) | cannotLink(A, C, Cnt).

The predicate allDifferent(A, B, C) is a Prolog predicate we implemented which simply checks that the three inputs are distinct.

To compare against a procedural approach, we would’ve needed to generate a graph of the constraints, and traverse it using any (O(n + m) traversal such as breadth-first search or depth-first search, and find the connected components in this graph, then generate must-link constraints for all pairs of items in a connected component. The rule based approach is much simpler and needed only two rules.
3.4 Evaluation

In our work, we use a publicly available Last.fm [1] dataset [4]. This dataset contains listening and tagging information from the music website Last.fm for 1892 users. We work with the tagging, not the listening information. It contains 17632 artists, 186479 tag assignments, i.e. tuples [user, tag, artist]. Since some artists are sparsely tagged, we consider only the 100 most annotated items in our evaluation.

To evaluate whether inferred instance-level constraints using the rules above improve clustering performance, we use the constraints in a semi-supervised clustering algorithm, namely COP K-Means [14]. We use a simple variation of the algorithm, which employs soft constraints rather than hard constraints. If there is a must-link constraint between an item $I_1$ and an already assigned item $I_2$ in cluster $C$, we increase distance of $I_1$ to $C$ by 1. Similarly, we decrease the distance by 1 for cannot-link constraints.

We represent items by their tag vectors. Each item has a tag vector of length equal to the number of unique tags, and with values indicating how many times this item has been tagged with this tag. Our distance measure is the cosine similarity. [13]

We compare the silhouette coefficient (SC) [9] and mean-square-error (MSE) for regular K-Means (without the inferred constraints) and COP K-Means, to evaluate clustering performance. Both are “internal indices”, which means they don’t use a labeled dataset to assess the clustering performance [10]. We resorted to using internal indices because after extensive research, we could not find a labeled dataset for item-clustering in folksonomies.

The silhouette coefficient is always between [-1, 1]: higher values closer to 1 indicate a better clustering performance. The mean-square error is the opposite, however, where lower values indicate a better clustering performance.

To account for the possible difference in results due to picking random seeds in the clustering process, we choose a set of seed points using the following strategy. We find the $k$ most used tags, where $k$ is the desired number of clusters for items, and pick the most tagged artist by each tag as the seed points.

There are two variables that influence the results: the must-link threshold and the cannot-link threshold. The must-link threshold is the number of users $u$, where a pair of items must have more than $u$ users in common to generate a must-link constraint between them. The cannot-link threshold is the number of tags $t$, where a pair of items must have less than $t$ tags in common to generate a cannot-link constraint between them. We note that we did not apply the transitive closure on the constraints as this would have taken a large amount of time.

Table 1 shows the results for different runs. The first run is regular K-Means, without constraints. The second and third run are the ones that optimize the SC for must-link and cannot-link only, respectively. The last run is the optimal run, both in terms of the MSE and the SC (highlighted in bold), and it incorporates both must-link and cannot-link constraints.

This indicates that inferring instance-level constraints can improve item clustering in folksonomies.
Table 1. Clustering Performance

<table>
<thead>
<tr>
<th>ML Threshold</th>
<th>CL Threshold</th>
<th># ML Constraints</th>
<th># CL Constraints</th>
<th>MSE</th>
<th>SC</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1754.09</td>
<td>0.21</td>
</tr>
<tr>
<td>-</td>
<td>8</td>
<td>0</td>
<td>325</td>
<td>1772.56</td>
<td>0.26</td>
</tr>
<tr>
<td>68</td>
<td>-</td>
<td>9</td>
<td>0</td>
<td>1779.69</td>
<td>0.24</td>
</tr>
<tr>
<td>68</td>
<td>8</td>
<td>9</td>
<td>325</td>
<td><strong>1713.64</strong></td>
<td><strong>0.29</strong></td>
</tr>
</tbody>
</table>

We note, however, the reason behind why the optimal must-link threshold was relatively high, giving a small number of constraints. This is because our distance measure for clustering does not include user information; it only relies on tag information. Therefore, it is natural that generating must-link constraints from user information could be detrimental, or in this case, giving a slight improvement but only when a small number of constraints is used.

4 Conclusion and Future Work

In this paper we used the declarative language Constraint Handling rules for improving semi-supervised clustering by generating must-link and cannot-link constraints. As a result of the work, it was found that rule based approaches are much simpler than their procedural counterparts, and are more natural to write. Our approach focused on the domain of folksonomies, where annotations are the primitive constraints that we used to infer new constraints. We found that inferring instance-level constraints could help improve item-clustering performance in folksonomies.

In the future, we intend to apply the approach over different domains, and show that rule based constraint reasoning is generally superior to procedural approaches. In addition, we intend to further incorporate the notion of popularity in our rules to produce better constraints. For example, a popular tag that is in common between a pair of items should have less impact on the cannot-link constraint than an unpopular tag in common.

We also would like to evaluate the clustering performance using a labeled dataset, which we would create or use a combination of datasets from folksonomies and clustering datasets. Optimally, we should test using different semi-supervised clustering algorithms. Using other distance measures (incorporating user information), as well as using dimensionality reduction techniques should be investigated. Another related problem is investigating more advanced techniques for seed-point initialization in folksonomies and their effect on clustering performance.

References

6. Thom Frühwirth. Description logic and rules the chr way. 2007.