Compressing Tags to Find Interesting Media Groups

Matthijs van Leeuwen
Dept. of Computer Science
Universiteit Utrecht
Utrecht, The Netherlands
mleeuwen@cs.uu.nl

Francesco Bonchi
Yahoo! Research
Barcelona, Spain
bonchi@yahoo-inc.com

Börkur Sigurbjörnsson
Yahoo! Research
Barcelona, Spain
borkur@yahoo-inc.com

Arno Siebes
Dept. of Computer Science
Universiteit Utrecht
Utrecht, The Netherlands
arno@cs.uu.nl

ABSTRACT

On photo sharing websites like Flickr and Zooomr, users are offered the possibility to assign tags to their uploaded pictures. Using these tags to find interesting groups of semantically related pictures in the result set of a given query is a problem with obvious applications. We analyse this problem from a Minimum Description Length (MDL) perspective and develop an algorithm that finds the most interesting groups. The method is based on Krimp, which finds small sets of patterns that characterise the data using compression. These patterns are sets of tags, often assigned together to photos.

The better a database compresses, the more structure it contains and thus the more homogeneous it is. Following this observation we devise a compression-based measure. Our experiments on Flickr data show that the most interesting and homogeneous groups are found. We show extensive examples and compare to clusterings on the Flickr website.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Clustering; H.2.8 [**Database Applications**]: Data Mining

General Terms

Algorithms, Experimentation, Theory

1. INTRODUCTION

Collaborative tagging services have become so popular that they hardly need any introduction. Flickr, del.icio.us, Technorati, Last.fm, or citeulike – just to mention a few – provide their users with a repository of resources (photos, videos, songs, blogs, urls, scientific papers, etc.), and the capability of assigning tags to these resources. Tags are freely chosen keywords and they are a simple yet powerful tool for organising, searching and exploring the resources.

Suppose you're fond of *high dynamic range* (*HDR* for short, a digital photo technique) and you want to see what other photographers are doing in HDR. You type the query *hdr* in Flickr and you get a list of more than one million pictures (all pictures tagged with *hdr*). This is not a very useful representation of the query result, neither for exploring nor for discovery. In a situation like this, it is very useful to have the resulting pictures automatically grouped on the basis of their other tags. For instance, the system might return groups about natural elements, such as {*sea, beach, sunset*} and {*cloud, winter, snow*}, a group about urban landscape {*city, building*}, or it may localise the pictures geographically, e.g. by means of the groups {*rome, italy*} and {*california, unitedstates*}. Grouping allows for a much better explorative experience.

Presenting the results of a query by means of groups may also help discovery. Suppose you search for *pyramid.* Instead of a unique long list of pictures, you might obtain the groups: {*chichenitza, mexico*}, {*giza, cairo*}, {*luxor, lasvegas*}, {*france, museum, louvre, glass*}, {*rome, italy*}, {*glastonbury, festival, stage*}, and {*alberta, edmonton, canada*}. Thus you discover that there are pyramids in Rome, Edmonton, and that there is a pyramid stage at the Glastonbury festival. You would have not discovered this without groupings, as the first photo of Rome's pyramid might appear very low in the result list.

Grouping also helps to deal with ambiguity. E.g. you type *jaguar* and the system returns to you groups about the animal, the car, the guitar, and the airplane.

In a nutshell, the problem at hand is the following: given the set of photos belonging to a particular tag, find the most interesting groups of photos based only on their other tags. Since we focus only on tags, we may consider every photo simply as a set of tags, or a *tagset.*

But, what makes a group of photos, i.e. a bag of tagsets, "interesting"? The examples just described suggest the following answer: large groups of photos that have many tags in common. Another word for having many tags in common is homogeneous, which results in the following informal problem statement:

> *For a database $\mathcal{D}_Q$ of photos containing the given query Q in their tagsets, find all significantly large and homogeneous groups $G \subseteq \mathcal{D}_Q$.*

Actually, Flickr has already implemented *Flickr clusters*[1], a tag clustering mechanism which returns 5 groups or less. The method proposed in this paper is rather different: (1) it finds groups of photos, i.e. groups of tagsets, instead of groups of tags, (2) it is based on the idea of *tagset compression*, and (3) it aims at producing a much more fine grained grouping, also allowing the user to fine-tune the grain.

Difficulties of the problem. All collaborative tagging services, even if dealing with completely different kinds of resources, share the same setting: there is a large database of user-created resources linked to a tremendous amount of user-chosen tags. The fact that tags are freely chosen by users means that irrelevant and meaningless tags are present, many synonyms and different languages are used, and so on. Moreover, different users have different intents and different tagging behaviours [1]: there are users that tag their pictures with the aim of classifying them for easy retrieval (maybe using very personal tags), while other users tag for making their pictures visited by as many other users as possible. There are users that assign the same large set of tags to all the pictures of a holiday, even if they visited different places during the holiday; thus creating very strong, but fake, correlations between tags. This all complicates using tags for any data mining/information retrieval task.

Another difficulty relates to the dimensions of the data. Even when querying for a single tag, both the number of different pictures and the number of tags can be prohibitively large. However, as pictures generally only have up to tens of tags, the picture/tag matrix is sparse. This combination makes it difficult to apply many of the common data mining methods. For instance, clustering methods like k-means [8] attempt to cluster all data. Because the data matrices are large and sparse, this won't give satisfactory results. Clustering all data is not the way to go: many pictures simply don't belong to a particular cluster or there is not enough information available to assign it to the right cluster.

Our approach and contribution. We take a very different approach to the problem by using MDL, the Minimum Description Length principle [4]. The general idea of MDL is that given the data and a set of models, the best model is that one that minimises the size of both the compressed data and the model. In the current context, this means that we are looking for groups of pictures that can be compressed well. We will make this more formal in the next section.

We consider search queries Q consisting of a conjunction of tags and we denote by $\mathcal{D}_Q$ the database containing the results of a search query. Each picture in $\mathcal{D}_Q$ is simply represented by the set of tags it contains. Our goal is to find all large and coherent groups of pictures in $\mathcal{D}_Q$. We do not require all pictures and/or tags to be assigned to a group.

For this, we build upon the Krimp [11] algorithm, which uses MDL to characterise data with small sets of patterns. Krimp has previously been shown to capture data distributions very well [14].

The algorithm presented in this paper uses Krimp to compress the entire dataset to obtain a small set of descriptive patterns. These patterns act as candidates in an agglomerative grouping scheme. A pattern is added to a group if it contributes to a better compression of the group. This results in a set of groups, from which the group that gives the largest gain in compression is chosen. This group is taken from the database and the algorithm is re-run on the remainder until no more groups are found. The algorithm will be given in more detail in Section 3.

We collect data from Flickr for our experiments. To address the problems specific for this type of data (mentioned above), we apply some pre-processing. Among this is a technique based on Wikipedia redirects (see Section 4).

Experiments are performed on a large number of queries. To demonstrate the high quality of the results, we show extensive examples of the groups found. For a quantitative evaluation, we introduce a compression-based score which measures how good a database can be compressed. Using this score, we compare our method to the groups that can be found by *Flickr clusters*. The results shows that our method finds groups of tagsets that can be compressed better than the clusters on Flickr. Even more important, pictures not grouped by our method cannot be compressed at all, while pictures not grouped in the current Flickr implementation can be compressed and thus contain structure.

2. THE PROBLEM

2.1 Preliminaries: MDL for Tagsets

Given are a query Q and some mechanism that retrieves the corresponding set of pictures $\mathcal{D}_Q$. We represent this set of pictures as a bag of tagsets (each picture represented by its associated tagset). The situation is identical to that of frequent itemset mining [5], where a bag of sets is given (usually dubbed *transactions*), and the patterns of interest are again sets of items (usually dubbed *itemsets*). Since in this case they consist of tags we call them *tagsets*.

We denote our input database by $\mathcal{D}$, and let $\mathcal{T}$ represent the vocabulary of all tags appearing in $\mathcal{D}$. A transaction $t \in \mathcal{D}$ (i.e. a picture) is a subset $t \subseteq \mathcal{T}$, a group G is a subset of transactions in $\mathcal{D}$, i.e. $G \subseteq \mathcal{D}$. $|\mathcal{D}|$ denotes the number of transactions in $\mathcal{D}$. A tagset X is a set of tags, i.e. $X \subseteq \mathcal{T}$, X occurs in a transaction t iff $X \subseteq t$, and the length of X is the number of tags it contains. The support of a tagset X in database $\mathcal{D}$, denoted by $sup_{\mathcal{D}}(X)$, is the number of transactions in $\mathcal{D}$ in which X occurs. That is, $sup_{\mathcal{D}}(X) = |\{t \in \mathcal{D} \mid X \subseteq t\}|$.

For a given minimal support threshold *minsup*, a tagset X is called frequent if its support exceeds *minsup* on $\mathcal{D}$, i.e. $sup_{\mathcal{D}}(X) \geq minsup$. Due to the A Priori property, $X \subseteq Y \rightarrow sup_{\mathcal{D}}(X) \geq sup_{\mathcal{D}}(Y)$, all frequent tagsets can be discovered efficiently [5].

MDL (Minimum Description Length) [4] is a practical version of Kolmogorov Complexity. Both embrace the slogan *Induction by Compression*. For MDL, this principle can be roughly described as follows.

Given a set of models $\mathcal{H}$, the best model $H \in \mathcal{H}$ is the one that minimises

$$L(H) + L(\mathcal{D}|H)$$

in which

- $L(H)$ is the length, in bits, of the description of H, and
- $L(\mathcal{D}|H)$ is the length, in bits, of the description of the data when encoded with H.

In order to use this principle for our problem statement, we need to define our collection of models and how to encode

[1]see www.flickr.com/photos/tags/jaguar/clusters/

Algorithm 1 The Cover Algorithm

```
1: Cover($CT, t$) :
2: $Y$ := smallest $X \in CT$ in coding order for which $X \subseteq t$
3: if $t \setminus Y = \emptyset$ then
4:     $Result = \{Y\}$
5: else
6:     $Result = \{Y\} \cup$ Cover($CT, t \setminus Y$)
7: end if
8: return $Result$
```

the data with such a model. Moreover, we need to determine how many bits are necessary to encode a model and how many are necessary for the coded data.

The remainder of this subsection is mostly taken from [11] and provided here for the convenience of the reader. We use (sets of) code tables as our models. Such a code table is defined as follows.

Definition 1. *Let $\mathcal{T}$ be a set of tags and $\mathcal{C}$ a set of code words. A code table CT for $\mathcal{T}$ and $\mathcal{C}$ is a two column table such that:*

1. *The first column contains tagsets over $\mathcal{T}$, contains at least all singleton tagsets, and is ordered descending on 1) tagset length, 2) support and 3) lexicographically.*

2. *The second column contains elements from $\mathcal{C}$, such that each element of $\mathcal{C}$ occurs at most once.*

A tagset $X \in \mathcal{P}(\mathcal{T})$ occurs in CT, denoted by $X \in CT$ iff X occurs in the first column of CT, similarly for a code $C \in \mathcal{C}$. For $X \in CT$, $code_{CT}(X)$ denotes its code, i.e. the corresponding element in the second column. $|CT|$ denotes the number of tagsets X in CT.

To encode a transaction database $\mathcal{D}$ over $\mathcal{T}$ with code table CT, we use the Cover algorithm from [11] given in Algorithm 1. Its parameters are a code table CT and a transaction t, the result is a set of elements of CT that cover t. Note that Cover is a well-defined function on any code table and any transaction t, since CT contains at least the singletons.

To encode database $\mathcal{D}$, we simply replace each transaction by the codes of the tagsets in its cover: $t \rightarrow \{code_{CT}(X) \mid X \in Cover(CT, t)\}$. Note, to ensure that we can decode an encoded database uniquely, we assume that $\mathcal{C}$ is a prefix code.

Since MDL is concerned with the best compression, the codes in CT should be chosen such that the most often used code has the shortest length. That is, we should use an optimal prefix code, i.e. the Shannon code. To define this for our code tables, we need to know how often a certain code is used. We call this the *usage* of a tagset in CT. Normalised, this usage represents the probability that that code is used in the encoding of an arbitrary $t \in \mathcal{D}$,

$$P(X|\mathcal{D}) = \frac{usage_{\mathcal{D}}(X)}{\sum_{Y \in CT} usage_{\mathcal{D}}(Y)}$$

The optimal code length is then $-\log$ of this probability and the coding table is optimal if all its codes have their optimal length. That is, a code is optimal for $\mathcal{D}$ iff $|code_{CT}(X)| = -\log(P(X|\mathcal{D}))$ and CT is code-optimal for $\mathcal{D}$ if all its codes $C \in CT$ are optimal for $\mathcal{D}$. From now on, we assume that code tables are code-optimal, unless we state differently.

For any database $\mathcal{D}$ and code table CT, we can now compute $L(\mathcal{D}|CT)$. The encoded size of a transaction is simply the sum of the sizes of the codes of the tagsets in its cover, $l(t|CT) = \sum_{X \in Cover(CT,t)} -\log(P(X|\mathcal{D}))$. The size of a $\mathcal{D}$, denoted by $L(\mathcal{D}|CT)$, is simply the sum of the sizes of its transactions, $L(\mathcal{D}|CT) = \sum_{t \in \mathcal{D}} l(t|CT)$.

In the remainder of the paper, we sometimes slightly abuse notation by denoting $L(\mathcal{D}|CT)$ with shortcut $CT(\mathcal{D})$.

The remaining problem is to determine the size of a code table. For the second column this is clear as we know the size of each of the codes. For encoding the first column, we use the simplest code table, i.e, the code table that contains only the singleton elements. This code table, with optimal code lengths for database $\mathcal{D}$, is called the *standard code table* for $\mathcal{D}$, denoted by ST. With this choice the size of CT, denoted by $L_{\mathcal{D}}(CT)$, is given by $L_{\mathcal{D}}(CT) = \sum_{X \in CT} |code_{ST}(X)| + |code_{CT}(X)|$.

With these results we know the total size of our encoded database. It is simply the sum of the size of the encoded database plus the size of the code table. The total size of the encoded database, denoted by $L(\mathcal{D}, CT)$, is given by $L(\mathcal{D}, CT) = L(\mathcal{D}|CT) + L_{\mathcal{D}}(CT)$.

Clearly, two different code tables will yield different encoded sizes. The lower the total encoded size, the better the code table captures the structure of the database. An *optimal code table* is one that minimises the total size.

Definition 2. *Let $\mathcal{D}$ be a database over $\mathcal{T}$ and let $\mathcal{CT}$ be the set of code tables that are code-optimal for $\mathcal{D}$. Code table $CT \in \mathcal{CT}$ is called* optimal *iff*

$$CT = \underset{CT \in \mathcal{CT}}{\operatorname{argmin}}\ L(\mathcal{D}, CT)$$

If we compare the encoded size reached by an optimal code table CT with the encoded size reached by the standard code table ST, we get insight into how much structure has been found. Or, how much structure is present in the database.

Definition 3. *Let $\mathcal{D}$ be a database over $\mathcal{T}$ and CT its optimal code table, we define* compressibility *of $\mathcal{D}$ as*

$$compressibility(\mathcal{D}) = 1 - \frac{L(\mathcal{D}, CT)}{L(\mathcal{D}, ST)}$$

The higher compressibility, the more structure we have in the database. If compressibility is 0, there is no structure discernible by a code table.

2.2 Problem Statement

Recall that our goal is to find all significantly large and homogeneous groups in the data. Both *significantly large* and *homogeneous* are vague terms, but luckily both can be made more precise in terms of compression. Homogeneous means that the group is characterised by a, relatively, small set of tags. That is, a group is homogeneous if it can be compressed well *relative* to the rest of the database. Hence, we should compare the performance of an overall optimal code table with a code table that is optimal on the group only. For this, we define *compression gain*:

Definition 4. *Let $\mathcal{D}$ be a database over $\mathcal{T}$, $G \subseteq \mathcal{D}$ a group and $CT_{\mathcal{D}}$ and CT_G their respective optimal code tables. We define* compression gain *of group G, denoted by $gain(G, CT_{\mathcal{D}})$, as*

$$gain(G, CT_{\mathcal{D}}) = CT_{\mathcal{D}}(G) - CT_G(G)$$

If the gain in compression for a particular group is large, this means that it can be compressed much better on its own than as part of the database. Note that compression gain is not strictly positive: negative gains indicate that a group is compressed better as part of the database. This could, e.g. be expected for a random subset of the data.

Compression gain is influenced by two factors: (1) homogeneity of the group and (2) size of the group. The first we already discussed above. For the second, note that if two groups G_1 and G_2 have the same optimal code table CT and G_1 is a superset of G_2, then $L(G_1, CT)$ will necessarily be bigger than $L(G_2, CT)$. Hence, bigger groups have potentially a larger compression gain. Since we look for large, homogeneous groups, we can now define the best group.

Problem 1 (Maximum Compression Gain Group). *Given a database $\mathcal{D}$ over a set of tags $\mathcal{T}$ and its optimal code table CT, find that group $G \subseteq \mathcal{D}$ that maximises $gain(G, CT)$.*

We do not want to find only one group, but the set of *all* large homogeneous groups. Denote this set by $\mathcal{G} = \{G_1, \ldots, G_n\}$. $\mathcal{G}$ contains all large homogeneous groups if the remainder of the database contains no more such groups. That is, if the remainder of the database has compressibility 0. Since we want our groups to be homogeneous, we require that the G_i are disjoint. We call a set of groups that has both these properties a *grouping*, the formal definition is as follows.

Definition 5. *Let $\mathcal{D}$ be a database over $\mathcal{T}$ and $\mathcal{G} = \{G_1, \ldots, G_n\}$ a set of groups in $\mathcal{D}$. $\mathcal{G}$ is a* grouping *of $\mathcal{D}$ iff*

1. *compressibility*$\Big(\mathcal{D} \setminus \Big(\bigcup_{G_i \in \mathcal{G}} G_i\Big)\Big) = 0$
2. $i \neq j \rightarrow G_i \cap G_j = \emptyset$

The grouping we are interested in is the one that maximises the total compression gain.

Problem 2 (Interesting Tag Grouping). *Given a database $\mathcal{D}$ over a set of tags $\mathcal{T}$ and its optimal code table CT, find a grouping $\mathcal{G}$ of $\mathcal{D}$ such that $\sum_{G_i \in \mathcal{G}} gain(G_i, CT)$ is maximal.*

3. THE ALGORITHM

In this section, we propose a new algorithm for the *Interesting Tag Grouping* problem. Our method is built upon a heuristic algorithm called Krimp that approximates the optimal code table for a database [11]. For this, Krimp needs a database and a set of candidate tagsets as input. As candidates, frequent tagsets up to a given *minsup* are used. The candidate set is ordered first descending on support, second descending on tagset cardinality and third lexicographically. Krimp starts with the standard code table ST. One by one, each pattern in the candidate set is added to the code table to see if it helps to improve database compression. If it does, it is kept in the code table, otherwise it is removed. After this decision, the next candidate is tested. In all experiments reported in this paper, pruning is applied, meaning that each time a tagset is kept in the code table all other elements are tested to see whether they still contribute to compression. Elements that do not are permanently removed.

3.1 Code Table-based Groups

For the *Maximum Compression Gain Group* problem, we need to find the group $G \subseteq \mathcal{D}$ that maximises $gain(G, CT)$. Unfortunately, $gain(G, CT)$ is neither a monotone nor an anti-monotone function. If we add a small set to G, the gain can both grow (a few well-chosen elements) or shrink (using random, dissimilar elements). Given that $\mathcal{D}$ has $2^{|\mathcal{D}|}$ subsets, this means that computing the group that gives the maximal compression gain is infeasible. A similar observation holds for the *Interesting Tag Grouping* problem.

Hence, we have to resort to heuristics. Given that the tagsets in the code table CT characterise the database well, it is reasonable to assume that these tagsets will also characterise the maximum compression gain group well. In other words, we only consider groups that are characterised by a set of code table elements.

Each code table element $X \in CT$ is associated with a bag of tagsets, viz., those tagsets which are encoded using X. If we denote this bag by $G(X, \mathcal{D})$, we have

$$G(X, \mathcal{D}) = \{t \in \mathcal{D} \mid X \in Cover(CT, t)\}$$

For a set g of code table elements we simply take the union of the individual bags, i.e.

$$G(g, \mathcal{D}) = \bigcup_{X \in g} G(X, \mathcal{D})$$

Although a code table generally doesn't have more than hundreds of tagsets, considering all $2^{|CT|}$ such groups as candidates is still infeasible. In other words, we need further heuristics.

3.2 Growing Homogeneous Groups

Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two databases over the same set of tags $\mathcal{T}$, with code tables CT_1 and CT_2, respectively. Moreover, let $CT_1 \approx CT_2$, i.e. they are based on more or less the same tagsets and the code lengths of these tagsets are also more or less the same. Then it is highly likely that the code table $CT_\cup$ of $\mathcal{D}_1 \cup \mathcal{D}_2$ will also be similar to CT_1 and CT_2. In other words, it is highly likely that

$$L(\mathcal{D}_1 \cup \mathcal{D}_2, CT_\cup) < L(\mathcal{D}_1, CT_1) + L(\mathcal{D}_2, CT_2)$$

This insight suggests a heuristic: we *grow* homogeneous groups. That is, we add code table elements one by one to the group, as long as the group stays homogeneous.

This strategy pre-supposes an order on the code table elements: which code table element do we try to add first? Given that the final result will depend on this order, we should try the best candidate first. In the light of our observation above, the more this new tagset has in common with the current set of code table elements the better a candidate it is. To make this precise, we define the notion of coherence.

Definition 6. *Let $\mathcal{D}$ be a database over the tagset $\mathcal{T}$ and let CT be its code table. Moreover, let $X \in CT$ and $g \subset CT$. Finally, let $U(g) = \bigcup_{Y \in g} Y$. Then the* coherence *of X with g is defined by*

$$coherence(X, g, \mathcal{D}) = \sum_{i \in (X \cap U(g))} usage_{G(g,\mathcal{D})}(\{i\})$$

Given a set of candidate tagsets $Cand$, the best candidate to try first is the one with the highest coherence with the current group, i.e.

$$bestCand(g, \mathcal{D}, Cand) = \underset{X \in Cand}{\operatorname{argmax}}\, coherence(X, g, \mathcal{D})$$

Algorithm 2 The GrowGroup Algorithm

```
1: **GrowGroup**($g, \mathcal{D}, CT, gMinsup$) :
2: $Cand := CT$
3: while $Cand \neq \emptyset$ do
4:    $best :=$ bestCand($g, \mathcal{D}, Cand$)
5:    $Cand := Cand \setminus best$
6:    if AcceptCandidate($best, g, \mathcal{D}, CT, gMinsup$) then
7:       $g := g \cup \{best\}$
8:    end if
9: end while
10: return $g$
```

```
1: AcceptCandidate($best, g, \mathcal{D}, CT, gMinsup$) :
2: $G := G(g, \mathcal{D})$
3: $G' := G(g \cup \{best\}, \mathcal{D})$
4: $\delta := G' \setminus G$
5: $CT_G :=$ Krimp($G$, MineCandidates($G, gMinsup$))
6: $CT_{G'} :=$ Krimp($G'$, MineCandidates($G', gMinsup$))
7: return $CT_{G'}(G') < CT_G(G) + CT(\delta)$
```

Next to this order, we need a criterion to decide whether or not to accept the candidate. This is, again, based on compression.

Definition 7. *Let g be the set of tagsets that define the current group $G = G(g, \mathcal{D})$. Consider candidate X, with candidate cluster $G' = G(g \cup \{X\}, \mathcal{D})$ and $\delta = G' \setminus G$. Let $CT_{\mathcal{D}}$, CT_G and $CT_{G'}$ be the optimal code tables for respectively $\mathcal{D}$, G and G'. We now accept candidate X iff*

$$CT_{G'}(G') < CT_G(G) + CT_{\mathcal{D}}(\delta)$$

When a candidate helps to improve compression of all data in the new group, we decide to keep it. Otherwise, we reject it and continue with the next candidate. The algorithm that does this is given in Algorithm 2.

3.3 Finding Interesting Tag Groups

The group growing algorithm given in the previous subsection only gives us the best candidate given a non-empty group. In line with its hill-climbing nature, we consider each code table element $X \in CT$ as starting point. That is, we grow a group from each code table element and choose the one with the maximal compression gain as our *Maximum Compression Gain Group*. Note that this implies that we only need to consider $|CT|$ possible groups.

To solve the *Interesting Tag Grouping* problem we use our solution for the *Maximum Compression Gain Group* problem iteratively. That is, we first find the Maximum Compression Gain Group G on $\mathcal{D}$, then repeat this on $\mathcal{D} \setminus G$, and so on until no group with gain larger than 0 can be found. This simple recursive scheme is used by the FindTagGroups algorithm presented in Algorithm 3.

The algorithm has four parameters, with database $\mathcal{D}$ obviously being the most important one. The second parameter is the minimum number of code table elements a group has to consist of to get accepted. Krimp requires a minimum support threshold to determine which frequent tagsets are used as candidates and we specify these separately for the database (*dbMinsup*) and the groups (*gMinsup*). We will give details on parameter settings in Section 5.

The result of the FindTagGroups algorithm is a set of pairs, each pair representing a group of the grouping. Each pair contains (1) the code table elements that were used to construct the group and (2) the transactions belonging to the group. The former can be used to give a group description that can be easily interpreted, as these are the tagsets that characterise the group. E.g. a group description could be the k most frequent tags or a 'tag cloud' with all tags.

Algorithm 3 The FindTagGroups Algorithm

```
1: FindTagGroups($\mathcal{D}, minElems, dbMinsup, gMinsup$) :
2: $groups := \emptyset$
3: loop
4:    $CT :=$ Krimp($\mathcal{D}$, MineCandidates($\mathcal{D}, dbMinsup$))
5:    $bestGain := 0$
6:    $best := \emptyset$
7:    for all $X \in CT$ do
8:       $cand :=$ GrowGroup($\{X\}, \mathcal{D}, CT, gMinsup$)
9:       if $gain(G(cand, \mathcal{D}), CT) > bestGain$ and
         $|cand| >= minElems$ then
10:         $bestGain := gain(G(cand, \mathcal{D}), CT)$
11:         $best := cand$
12:      end if
13:   end for
14:   if $best \neq \emptyset$ then
15:      $groups := groups \cup \{(best, G(best, \mathcal{D}))\}$
16:      $\mathcal{D} := \mathcal{D} \setminus G(best, \mathcal{D})$
17:   else
18:      break
19:   end if
20: end loop
21: return $groups$
```

4. DATA PRE-PROCESSING

Our data collection consists of tagsets from Flickr photos. We evaluate the performance of the algorithm on a diverse set of datasets. Each dataset consists of all photos for a certain query, i.e. all photos that have a certain tag assigned. Table 1 shows a list of the queries used to evaluate the algorithm. We use a wide range of different topic types, ranging from locations to photography terms, with general and specific, as well as ambiguous and non-ambiguous queries.

To reduce data sparsity we limit our attention to a subset of the Flickr tag vocabulary, consisting of roughly a million tags used by the largest number of Flickr users. Effectively this excludes only tags that are used by very few users.

Another source of data sparsity is that Flickr users use different tags to refer to the same thing. The source of this sparsity can have several roots: (1) singular or plural forms of a concept (e.g. scyscraper, skyscrapers); (2) alternative names of entities (e.g. New York City, NYC, New York NY); (3) multilingual references to the same entity (e.g. Italy, Italia, Italie); or (4) common misspellings of an entity (e.g. Effel Tower, Eifel Tower). We address this problem using Wikipedia redirects. If a user tries to access the 'NYC' Wikipedia page she is redirected to the 'New York City' Wikipedia page. We downloaded the list of redirects used by Wikipedia and mapped them to Flickr tags using exact string matching. This results in a set of rewrite rules we used to normalise the Flickr tags. E.g. all occurrences of the tag 'nyc' were replaced by the tag 'new york city'.

Figure 1 shows some examples of the rewrite rules that were gathered using Wikipedia redirects. The figure shows all strings that were transformed to the given normalised

new york city city new york, new york, city of new york, new york skyline, nyc, new yawk, ny city, the city that never sleeps, new york new york

eiffel tower eiffle tower, iffel tower, effel tower, tour eiffel, eifel tower, eiffel tour, the eiffel tower, la tour eiffel, altitude 95

skyscraper skyscrapers, office tower, tall buildings, skyskraper, skycrappers

Figure 1: Examples of tag transformations using Wikipedia redirects.

string (bold). We can see that using the redirects we address, to some extent, the problem of singular/plural notation, alternative names, multilinguality and common misspellings.

A very common 'problem' in Flickr data is that users assign a large set of tags to entire series of photos. For example, when someone has been on holiday to Paris, *all* photos get the tags *europe, paris, france, eiffeltower, seine, notredame, arcdetriomphe, montmartre*, and so on. These tagsets are misleading and would negatively influence the results. As a workaround, we make all transactions in a dataset unique; after all, we cannot distinguish photos using only tag information if they have exactly the same tags.

Another issue pointed out by this example is that some items are not informative: e.g. if we query for *eiffeltower*, many of the transactions contain the tags *europe*, *paris* and *france*. If one were to find a large group, including these high-frequent tags would clutter the results. Therefore, we remove all items with frequency $\geq 15\%$ from each dataset. One final pre-processing step is to remove all tags containing either a year in the range 1900-2009 or any camera brand, as both kinds of tags introduce a lot of noise.

5. EXPERIMENTS

5.1 Experimental Setup

To assess the quality of the groups produced by the FINDTAGGROUPS algorithm, a large number of experiments was performed on the datasets for which basic properties are given in Table 1. The experiments reported on in this section all use the same parameter settings:

dbMinsup = 0.33% This is the *minsup* parameter used by KRIMP to determine the set of candidate frequent tagsets (see Algorithm 3). This parameter should be low enough to enable KRIMP to capture the structure in the data, but not too low as this would allow very infrequent tagsets to influence the results. For this application, we empirically found a *minsup* of 0.33% to always give good results.

gMinsup = 20% This parameter is used when running KRIMP on a group (see Algorithm 2 and 3). As groups are generally much smaller than the entire database, this parameter can be set higher than *dbMinsup*. We found a setting of 20% to give good results.

minElems = 2 This parameter determines the minimum number of code table elements required for a group. In practice, it acts as a trade-off between fewer groups that are more conservative (low value) and more groups that are more explorative (high value). Unless mentioned otherwise, this parameter is fixed to 2.

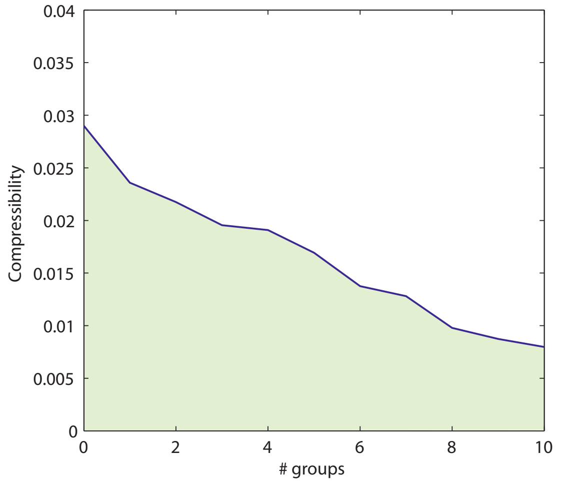

Figure 2: Database compressibility for *Bicycle* when applying FINDTAGGROUPS (minElems = 2).

5.2 An Example Query

To demonstrate how the algorithm works and performs, let us first consider a single dataset: *Bicycle*. As we can see from Table 1, it contains 98,304 photos with a total of 10,126 different tags assigned, 6.1 per photo on average.

When FINDTAGGROUPS is applied on this dataset, it first constructs a code table using KRIMP. Then, a candidate group is built for each tagset in the code table (using the GROWGROUP algorithm). One such tagset is {*race, tourofcalifornia*} and the algorithm successfully adds coherent tags to form a group: first the tagset {*race, racing, tourofcalifornia, atoc, toc*}, then {*race, racing, tourofcalifornia, atoc*}, {*race, racing, tourofcalifornia, stage3*}, and so on until 22 code table elements have been selected to form a group.

Compression gain is computed for each of the candidate groups constructed and the group with the maximal gain is chosen. In this case, the group with most frequent tags {*race, racing, tourofcalifornia, atoc, cycling*} has the largest gain (57,939bits) and is selected. All 3,564 transactions belonging to this group are removed from the database, a new code table is built on the remainder of the database and the process repeats itself.

As explained in Subsection 2.1, we can use compressibility to measure how much structure is present in a database. If we do this for the group we just found, we get a value of 0.05, so there is some structure but not too much. However, it is more interesting to see whether there is any structure in the remainder of the database. Compressibility of the original database was 0.029, after removing the group it is 0.024, so we removed some structure. Figure 2 shows database compressibility computed each time after a group has been removed. This shows that we remove structure each time we form a group and that there is hardly any structure left when the algorithm ends: compressibility after finding 10 groups is 0.008.

So far we assumed the *minElems* parameter to have its default value 2. However, Figure 3 shows the resulting groupings for different settings of *minElems*. Each numbered item represents a group and the groups are given in the order in which they are found. For each group, the union of the code table elements defining the group is taken and the tags are ordered descending on frequency to obtain a group description. For compact representation, only the 5 most frequent

minElems=1
1. race racing tourofcalifornia atoc cycling
2. bicicleta bici fahrrad cycling
3. freestyle bmx
4. netherlands amsterdam holland nederland fiets
5. fixedgear fixie fixed gear trackbicycle
6. travel trip
7. street tokyo bw japan people
8. cycling cycle mountainbike mtb london
9. newyorkcity city urban china beijing
10. portland oregon bikeportland
11. p12 pro racing race
12. canada bc toronto
13. black white
14. blue sky red cloud
15. sanfrancisco california sf
16. unitedkingdom england
17. winter snow
18. paris france
19. shanghai china
20. child kid

minElems=2
1. race racing tourofcalifornia atoc cycling
2. bicicleta bici fahrrad cycling
3. netherlands amsterdam holland nederland fiets
4. bmx freestyle oldschool
5. fixedgear fixie fixed gear trackbicycle
6. tokyo china japan street people
7. cycling cycle mountainbike mtb london
8. newyorkcity city urban manhattan street
9. portland oregon bikeportland
10. sky blue cloud red

minElems=6
1. race racing tourofcalifornia atoc cycling
2. netherlands amsterdam holland fahrrad nederland
3. street tokyo bw japan people
4. cycling city urban cycle street
5. fixedgear fixie fixed gear track

minElems=12
1. race racing tourofcalifornia atoc cycling
2. netherlands amsterdam holland fahrrad nederland
3. street city urban people bw

Figure 3: Groups for *Bicycle* (different settings for *minElems*, showing the 5 most frequent tags per group).

tags are given (or less if the group description is smaller). Font size represents frequency relative to the most frequent tag in that particular group description.

Clearly, most groups are found when *minElems = 1*. Groups with different topics can be distinguished: sports activities (1, 3, 8, 11), locations (4, 7, 9, 10, 12, 15, 16, 18, 19), 'general circumstances' (6, 14, 17). The second group is due to a linguistic problem which is not solved by the pre-processing.

Increasing *minElems* results in a reduction in the number of groups. This does not mean that only the top-k groups from the *minElems = 1* result set are picked. Instead, a subset of that result set is selected and sometimes even new groups are found (e.g. *minElems = 12* group 3). Unsurprisingly, increasing the *minElems* value results in fewer groups with larger group descriptions. These are often also larger in the number of transactions and can generally be regarded more confident but less surprising. For the examples in Figure 3, for *minElems = 1* a group on average contains 1,181 transactions, 1,828 transactions for *minElems = 2*, 3,034 transactions for *minElems = 6* and 3,325 transactions for *minElems = 12*.

We found *minElems = 2* to give a good balance in the number of groups, the size of the groups and how conservative/surprising these are. We therefore use this as the default parameter setting in the rest of the section.

5.3 More Datasets

Table 1 shows quantitative results for all datasets. From this table, we see that the algorithm generally finds 10-20 groups, but sometimes more, with *fun* as extreme with 50 groups. This can be explained by the fact that 'fun' is a very general concept, so it is composed of many very different conceptual groups and these are all identified. On average, between 5 and 8 code table elements are used for a group. The average number of transactions (photos) that belong to a group depends quite a lot on the dataset. Average group sizes range from only 73 for *fun* up to 17,899 for *art*. However, the size relative to the original database does not vary much, from 1.3% up to 2.7%. The complete groupings usually give a total coverage between 20 and 40%.

The three rightmost columns show compressibility values: for the groups found (averaged), for the database remaining after the algorithm is finished and for the initial database (called *base*). The compressibility of the initial database can be regarded as a baseline, as it is an indication of how much structure is present. In general, there is not that much structure present in the sparse tag data so values are not far above 0. However, the groups identified have more structure than baseline: for all datasets, average group compressibility is higher than baseline. Even more important is that compressibility of the database remaining after removing all groups is always very close to 0 (<= 0.01). Hence, all structure that was in the database has been captured.

To give some more insight in the groups found by the algorithm, Figure 4 shows additional results for 4 datasets. In general, the resulting groups clearly identify specific photo collections that are conceptually different. Sometimes there appears to be some redundancy (from a semantic point of view), but in most cases additional tags reveal subtle differences. Note that not all tags that are part of the group descriptions are shown (see Subsection 5.2).

Some groups are surprising. For example, the second group for *black and white*, with *dewolf* as most frequent tag. At first sight, it looks like a single user got chosen as second-most important concept for this query. However, performing a Flickr search for *blackandwhite* and *dewolf* quickly learns us that there is a Nick DeWolf photo archive project and this archive contains over 13,000 black&white pictures taken by this co-founder of Teradyne in Boston. After all, it is thus by no means strange that this comes up as second group.

5.4 Compared to Flickr

As mentioned in the introduction, Flickr offers its visitors the possibility to browse through 'clusters' of photos given a certain tag. However, Flickr clusters are conceptually different from our groupings: (1) Flickr clusters consist of tags, not pictures, (2) Flickr only gives 1 to 5 clusters and (3) clusters are usually quite general and conservative (hardly

Dataset					**Group average**		**Compressibility**		
Query	$\lvert\mathcal{T}\rvert$	$\lvert db\rvert$	*#tags/photo*	*#groups*	*#elems*	*#transactions*	*group avg*	*remainder*	*base*
Architecture	13657	541810	8.0	22	5.5	8507 (1.6%)	0.064	0.006	0.047
Art	15498	861711	8.5	15	7.9	17899 (2.1%)	0.034	0.002	0.032
Bicycle	10126	98304	6.1	10	7.1	1828 (1.9%)	0.061	0.008	0.029
Black and white	14042	567554	7.4	17	6.5	10815 (1.9%)	0.056	0.004	0.037
Eiffeltower	3994	13087	4.8	12	6.7	309 (2.4%)	0.072	0.001	0.039
Florence	5299	42528	5.3	18	6.9	731 (1.7%)	0.078	0.007	0.048
Fun	9162	5533	14.2	50	6.3	73 (1.3%)	0.213	0.009	0.137
HDR	12788	210389	7.3	14	5.4	3608 (1.7%)	0.043	0.005	0.032
Hollywood	9090	51176	7.7	24	8.1	731 (1.4%)	0.112	0.004	0.081
Jaguar	9283	10996	7.0	11	9.3	251 (2.3%)	0.132	0.012	0.067
Manhattan	9328	167087	6.8	19	8.6	3125 (1.9%)	0.081	0.005	0.063
Niagara falls	3144	11307	4.8	18	5.8	227 (2.0%)	0.117	0.006	0.048
Night	13366	790277	7.4	19	5.0	13011 (1.6%)	0.043	0.004	0.034
Portrait	12982	846530	7.6	19	6.2	15477 (1.8%)	0.047	0.002	0.037
Pyramid	7288	15657	6.9	20	11.2	369 (2.4%)	0.115	0.004	0.066
Reflection	13157	437769	7.5	17	6.3	8660 (2.0%)	0.036	0.005	0.033
Rock	16305	249790	8.0	13	15.9	6635 (2.7%)	0.092	0.005	0.044
Skyscraper	9298	42292	8.5	38	8.8	731 (1.7%)	0.059	0.009	0.083
Spain	9376	324682	6.9	19	6.3	5670 (1.7%)	0.079	0.008	0.054
Windmill	8474	23838	5.9	13	7.6	574 (2.4%)	0.066	0.003	0.036

Table 1: Dataset properties and quantitative results obtained with the FindTagGroups algorithm. For each dataset, the number of groups found, the average number of code table elements and transactions per group are given. Compressibility is shown for all groups (averaged), the database remaining after the algorithm is finished and the initial database.

surprising). An objective semantic comparison of the groups we find to the Flickr clusters is therefore impossible; it would come down to subjective preference of the person asked.

To give an example, Flickr gives the following clusters for *bicycle* (`www.flickr.com/photos/tags/bicycle/clusters/`):

1. bike street bw cycling city urban cycle red road bikes
2. amsterdam netherlands holland
3. fixie fixed fixedgear gear
4. england uk

It is up to personal taste whether you prefer this or the groupings we presented in Figure 3. What we can do though, is to 'simulate' Flickr clusters with the data we have and compute compressibility values for both the clusters and the unclustered photos. To this end, we first retrieve the tagsets that identify the clusters from the Flickr website and pre-process these the same way we pre-processed our data. Next, take the pre-processed datasets also used for the previous experiments and assign each transaction to one of the clusters or to the database with 'remaining' photos. Each cluster is constructed as follows: each photo is assigned to the tagset with which it has the largest tag intersection. (In case of a tie, the transaction is assigned to the first tagset with the largest intersection, clusters ordered as on the website.) Any transactions that do not intersect with any of the cluster's tagsets are assigned to the remainder database.

For all queries used in this paper, Flickr presents 3.5 clusters on average. Average compressibility for all Flickr clusters we obtained is 0.014. This means that there is hardly any structure in the clusters that can be captured in a code table by Krimp, much less than the groups the FindTagGroups algorithm finds.

More important are the compressibility values computed for the database containing all photos not belonging to any cluster: is there any structure left? The average value we obtained for all datasets is 0.035, clearly indicating that there is structure not yet captured in one of the clusters. Figure 5 shows a comparison of remaining database compressibility between the Flickr method and our method. It is easy to see that our method is better at finding all structure in the database than Flickr's method is.

5.5 Running Times

All experiments have been performed on a machine with a 3GHz Intel Xeon CPU and 2Gb of memory. Running times depend on a large number of factors, amongst which the number of tags $|\mathcal{T}|$ and transactions $|\mathcal{D}|$. The amount

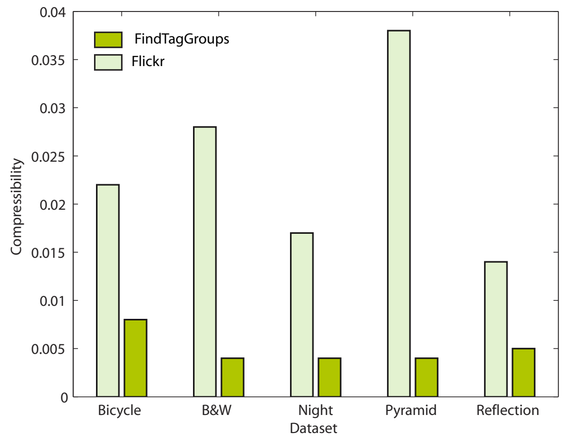

Figure 5: Remaining database compressibility compared to Flickr, for 5 datasets.

Black and white

1. black art girl kid
2. dewolf blackwhite nick boston
3. bn biancoenero blancoynegro italy portrait
4. woman girl portrait female face
5. film 35mmfilm format '120' ilford
6. noiretblanc france paris nb blancoynegro
7. selfportrait me 365days portrait self
8. portrait people face black man
9. unitedkingdom england london monochrome blackwhite
10. white black blackwhite
11. street urban city newyorkcity manhattan
12. tree cloud nature sky winter
13. cat pet animal dog kitty
14. black white light
15. child kid portrait girl boy
16. beach sand ocean sea water
17. flower macro nature

Night

1. california sanfrancisco sanfranciscobayarea city urban
2. newyorkcity manhattan lights christmas newyork
3. france paris nuit europe
4. japan tokyo light geotagging city
5. thestand sky cloud luna tree
6. london unitedkingdom england riverthames lights
7. party people friends recreation portrait
8. neon sign lights
9. sky cloud sunset star tree
10. italy notte rome roma
11. light darkness street color red
12. city lights urban street light
13. long exposure longexposure light lights
14. lights black darkness white street
15. water bridge reflection river lights
16. australia sydney melbourne
17. germany nacht berlin
18. building lights light architecture tree
19. canada ontario toronto

Reflection

1. selfportrait me mirror 365days self
2. beach sunset sea cloud sky
3. lake tree landscape nature mountain
4. nature tree bird pond green
5. night light river lights bridge
6. building architecture glass sky blue
7. tree autumn river leaf pond
8. light shadow color glass mirror
9. unitedkingdom england london
10. sky cloud blue tree
11. mirror portrait self automobile girl
12. macro drop closeup
13. black white bw blackandwhite
14. window glass shop street
15. red blue green yellow color
16. newyorkcity manhattan newyork
17. sunset sun sky

Pyramid

1. itza chichenitza maya mexico chichen
2. france museum glass europe architecture
3. sanfrancisco transamerica california francisco san
4. giza cairo sphinx khufu africa
5. mexico ruins chichenitza mayan maya
6. mexico ruins mexicocity teotihuacan puebla
7. mexico teotihuacan guatemala maya tikal
8. mountain switzerland niesen hurni christopher
9. luxor lasvegas nevada casino hotel
10. sky cloud architecture blue building
11. camel cairo saqqara sand desert
12. france night bw pyramide blackandwhite
13. rome italy piramide roma museo
14. glastonbury festival stage glastonburyfestival
15. mexico travel chichen itza trip
16. alberta edmonton canada muttartconservatory conservatory
17. poodle babyboy royalcanin keops love
18. night light lights
19. galveston texas moodygardens
20. france architecture fountain

Figure 4: Results for *Black and white*, *Reflection*, *Night* and *Pyramid* (showing at most 5 tags per group).

of structure and the number of groups also play an important role. 9 datasets required less than 20 minutes, with *Eiffeltower* being the quickest taking only 3 minutes. The remaining datasets took longer than that, with timings ranging from 1 hour for *Fun* and *HDR*, up to 20 hours for *Architecture* and *Portrait.*

These running times indicate that the method should be used offline, not online at query time. For most queries, stability of the resulting groupings should be high, implicating that the algorithm does not need to be re-run often.

5.6 Discussion

Using compression, the FindTagGroups algorithm aptly finds significantly large and homogeneous groups in tag data. The groupings usually contain between 20% and 40% of the photos in the database, a fair amount given the sparsity of the data. Compressibility shows that the groups are more homogeneous than the original database, while there is no discernible structure left in the remaining database.

Manual inspection of the groups shows that the quality of the groupings is high; a manageable number of semantically coherent groups is found. As each group is characterised by a small set of tagsets, it is easy to give intuitive group descriptions using tag clouds. The grain of the groups returned by the algorithm can be fine-tuned by the end-user: he may either wish for more, smaller and explorative groups or prefer less, larger and conservative groups.

A qualitative comparison to Flickr clusters is difficult, but by simulating these clusters we showed that Flickr clusters leave more structure in the database than our method.

Data preparation is very important, due to the amount of noise present in user-generated tagsets. The method based on Wikipedia redirects solves many problems with synonyms and other linguistic issues.

In the experiments we focus on pictures, but our algorithm can find groups for any type of tagged resource. Apart from improving usability of tag searching, groupings could also be used for applications like tag recommendation [12, 13].

6. RELATED WORK

The main reference for our work is *Flickr clusters.* Even if the technical details are not public, by observing the results we can infer (1) that it clusters tags, as the same tag can not belong to more than one group, and (2) that the maximum number of clusters is 5. As said in the introduction our method finds groups of pictures, i.e. groups of tagsets, instead of groups of tags, and it aims at producing a much more fine grained grouping. A consequence of having a larger number of groups is that they have larger cohesiveness.

Begelman *et al.* [2] propose to first build a graph of tags and then to apply a spectral bisection algorithm in combination with modularity measure optimization. Similar to Flickr clusters, they cluster the tags and not the resources (URLs). Another difference with our work is that they try to cluster the whole tag space, e.g. for creating multiple, more cohesive, *tag clouds* instead of a unique large tag cloud. Instead we focus on the result set of a given query.

Recently, three other papers [3, 10, 15] have studied the use of tags from large-scale social bookmarking sites (such as del.icio.us) for clustering web pages in semantic groups. Ramage *et al.* [10], explore the use of clusterings over vector space models that includes tags and page text. In this vector space k-means is compared to a novel generative clustering algorithm based on latent Dirichlet allocation. Zhou *et al.* [15] investigate the possibility of devising generative models of tags and documents contents in order to improve information retrieval. Their holistic approach combines a language model of the resources and the tags with user domain categorisation. Another work considering resources, tags, and users in an unifying framework is by Grahl *et al.* [3]. They present a conceptual clustering where first tags are clustered by k-means, then the FolkRank [6] algorithm is applied to discover resources and users that are related to each cluster of tags. Yeung *et al.* [9] also focus their analysis on resources-tags-users triplets, in particular for tag disambiguation.

Subspace clustering [7] tries to find clusters in 'subspaces' of the data, where a subspace is usually defined as subsets of both the data points and attributes. There are two important differences with our method. First, tags are not dimensions and the fact that a tag is not present may mean different things – we only look at the tags that are present. Secondly, we do not attempt to group all data, which subspace clustering methods still do.

7. CONCLUSIONS

We propose an algorithm that addresses the problem of finding homogeneous and significantly large groups in tagged resources, such as photos. The method is based on the MDL principle and uses the KRIMP algorithm to characterise data with small sets of patterns. The best group is the group that gives the largest gain in compression, i.e. it can be compressed much better as a group than as part of the entire database.

We perform experiments on a large collection of datasets obtained from Flickr; the data consists of tagsets belonging to photos. A pre-processing technique based on Wikipedia redirects solves many problems typical for this type of data. Experiments show that semantically related groups are identified and no structure is left in the database.

8. REFERENCES

[1] M. Ames and M. Naaman. Why we tag: motivations for annotation in mobile and online media. In *CHI 2007: Proc. of the SIGCHI conf. on Human factors in computing systems*, pages 971–980. ACM, 2007.

[2] G. Begelman, P. Keller, and F. Smadja. Automated tag clustering improved search and exploration in the tag space. In *Proc. of Collaborative Web Tagging Workshop at WWW 2006*, 2006.

[3] M. Grahl, A. Hotho, and G. Stumme. Conceptual clustering of social bookmarking sites. In *LWA 2007: Lernen - Wissen - Adaption*, 2007.

[4] P. D. Grünwald. Minimum description length tutorial. In P. Grünwald and I. Myung, editors, *Advances in Minimum Description Length.* MIT Press, 2005.

[5] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM SIGMOD Conference*, pages 1–12, 2000.

[6] A. Hotho, R. Jäschke, C. Schmitz, and G. Stumme. Information retrieval in folksonomies: Search and ranking. In *ESWC 2006: Proc. of the 3rd European Semantic Web Conference*, 2006.

[7] H.-P. Kriegel, P. Kröger, and A. Zimek. Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. *TKDD*, 3(1), 2009.

[8] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. of the 5th Symposium on Mathematical Statistics and Probability*, 1967.

[9] C. man Au Yeung, N. Gibbins, and N. Shadbolt. Understanding the semantics of ambiguous tags in folksonomies. In *Proc. of the First Int. Workshop on Emergent Semantics and Ontology Evolution, ESOE 2007*, 2007.

[10] D. Ramage, P. Heymann, C. D. Manning, and H. Garcia-Molina. Clustering the tagged web. In *WSDM 2009: Proc. of the 2nd ACM Int. Conference on Web Search and Data Mining*, 2009.

[11] A. Siebes, J. Vreeken, and M. van Leeuwen. Item sets that compress. In *SDM 2006: SIAM Conference on Data Mining*, pages 393–404, 2006.

[12] B. Sigurbjörnsson and R. van Zwol. Flickr tag recommendation based on collective knowledge. In *WWW 2008*, 2008.

[13] Y. Song, Z. Zhuang, H. Li, Q. Zhao, J. Li, W.-C. Lee, and C. L. Giles. Real-time automatic tag recommendation. In *ACM SIGIR*, pages 515–522, 2008.

[14] J. Vreeken, M. van Leeuwen, and A. Siebes. Characterising the difference. In *SIGKDD 2007: ACM Conf. on Knowledge Discovery and Data Mining*, pages 765–774, 2007.

[15] D. Zhou, J. Bian, S. Zheng, H. Zha, and C. L. Giles. Exploring social annotations for information retrieval. In *WWW 2008: Proc. of the 17th int. conf on World Wide Web*, 2008.