Scaling Pair-Wise Similarity-Based Algorithms in Tagging Spaces

Damir Vandic, Flavius Frasincar, and Frederik Hogenboom

Erasmus University Rotterdam,
PO Box 1738, NL-3000 DR, Rotterdam, The Netherlands
{vandic, frasincar, fhogenboom}@ese.eur.nl

Abstract. Users of Web tag spaces, e.g., Flickr, find it difficult to get adequate search results due to syntactic and semantic tag variations. In most approaches that address this problem, the cosine similarity between tags plays a major role. However, the use of this similarity introduces a scalability problem as the number of similarities that need to be computed grows quadratically with the number of tags. In this paper, we propose a novel algorithm that filters insignificant cosine similarities in linear time complexity with respect to the number of tags. Our approach shows a significant reduction in the number of calculations, which makes it possible to process larger tag data sets than ever before. To evaluate our approach, we used a data set containing 51 million pictures and 112 million tag annotations from Flickr.

1 Introduction

Due to the ever increasing amount of data readily available on the Web, the development of applications exploiting this data flourishes as never before. However, because of the data abundance, an increasing number of these Web applications suffers from scalability issues. These developments have caused the focus of recent Web research to shift to scalability aspects. Social Web applications (e.g., in the area of products, photos, videos, links, etc.) also face these scalability issues. The reason why these systems do not scale well is because they often use pair-wise similarity measures (e.g., cosine similarity, Dice coefficient, Jaccard coefficient, etc.) [9, 10]. This introduces scalability problems as the number of pair-wise similarities that have to be computed (i.e., the number of unique pairs) grows quadratically with the number of vectors. As a result of this, the algorithms that use these pair-wise similarities have at least $O(n^2)$ time complexity, where n is the number of input vectors.

The fact that an algorithm has $O(n^2)$ complexity makes it difficult to apply it on large data sets. An area where this becomes evident are the social tagging systems, where users can assign tags to Web resources. These Web resources can be for example URLs (e.g., Delicious), images (e.g., Flickr), and videos (e.g., YouTube). Because users can use any tag they want, the number of distinct tags is enormous. Besides the number of unique tags, the number of resources is also growing fast. For example, let us consider Flickr, which is a Web site where

users can upload pictures and assign tags to them. In 2011, Flickr had 6 billion pictures in their database [11]. Now imagine that Flickr needs to compute the similarity between all (unique) pairs of pictures. Let us assume that Flickr is able to compute 100 billion pairs per second. Even at this speed, it would take a little less than 6 years to compute all combinations[1]. From this computation, we can see that there is a need to deal with pair-wise similarity computations for such large amounts of high-dimensional data.

In this paper, we focus on the scalability issue that arises with the computation of pair-wise similarities in tagging spaces (e.g., Flickr). We present an algorithm that approximately filters insignificant similarity pairs (i.e., similarities that are relatively low). The proposed algorithm is not exact but it has linear time complexity with respect to the number of input vectors and is therefore applicable to large amounts of input vectors. We report the results for the cosine similarity applied on a large Flickr data set, but our approach is applicable to any similarity measure that uses the dot product between two vectors.

The structure of this paper is as follows. We present the related work in Section 2. In Section 3, we define the problem in more detail and present our algorithm using a synthetic data set and the cosine similarity used as similarity measure. We evaluate our approach on a real data set and present the results in Section 4. Last, in Section 5, we draw conclusions and present future work.

2 Related work

The cosine similarity is a popular similarity measure that is widely used across different domains. In particular, we can find many approaches in the tagging spaces domain that are that are making use of this similarity [5, 7, 9, 10]. The reason for this is that the cosine similarity has proven to give stable results for tagging data sets. The drawback of using the cosine similarity is that it introduces scalability issues, as nowadays the number of tags and resources is growing fast. Because all similarity pairs have to be computed, the approaches that use the cosine similarity have at least $O(n^2)$ time complexity, where n is the number of tags of resources.

In the literature we can find several approaches that aim to address the scalability issue of computing pair-wise similarities. A technique that is related to our approach is the Locality Sensitive Hashing (LSH) technique, presented in [6]. LSH is a well-known approximate algorithm that is used to find clusters of similar objects. For example, it can be used to perform approximate nearest neighbour search. LSH generates n projections of the data on randomly chosen dimensions. After that, for each vector in the data set and each previously computed projection, a hash is determined using the vector features that are presented in that particular projection. The similarity pairs are constructed by finding all vector pairs that have a matching hash along the same projection.

[1] If $C = ((6 \times 10^9)^2 - (6 \times 10^9)) \times 0.5 \approx 1.8 \times 10^{19}$ unique combinations, and we can process 100×10^9 combinations per second, then it takes approximately $C/(100 \times 10^9)/(60 \times 60 \times 24 \times 365) \approx 5.71$ years to compute all similarities.

The key difference between LSH and our approach is that our approach has guaranteed linear time complexity with respect to the number of input vectors, while LSH has a polynomial pre-processing time.

The authors of [2] take a different approach to the similarity search problem. They propose an exact technique which is able to precisely find all pairs that have a similarity above some threshold. This approach uses an inverted index where the inverted indices are dynamically built and a score accumulation method is used to collect the similarity values. The difference between our approach and this approach is that we propose an approximate algorithm that gives good results with the focus on reducing the computational effort, while the authors of [2] propose an exact algorithm that works with a given threshold. Furthermore, the approach in [2] has not been evaluated on data sets obtained from tagging spaces.

Although not directly related to our research, there are approaches from the database community that address a similar problem, which are worth mentioning. For example, the authors of [3] depart from traditional database design to more flexible database design that is more suited for parallel algorithms. These parallel algorithms are used for the purpose of speeding up the computation of pair-wise similarities (e.g., cosine similarity). In this paper we focus on efficient the computation of similarities in a sequential execution.

3 Algorithm

In this section, we explain in detail the proposed algorithm, which aims at reducing the number of cosine computations. The proposed algorithm has some similar characteristics to LSH, as it also uses a hash function to cluster potentially similar objects, but differs on many aspects. For example, we use only one hash function, which results in a binary encoding of the vector that indicates where the significant parts of the vector reside. Furthermore, we have only one corresponding hash value for each vector.

As already mentioned, our algorithm is tailored and evaluated for tag spaces. Algorithms and applications for tag spaces often use a so-called tag co-occurrence matrix. A tag co-occurrence matrix is a $n \times n$ matrix C, where n is the number of tags, and C_{ij} denotes how often tag i and tag j have co-occurred in the data set (e.g., on pictures). Note that co-occurrences matrices are symmetrical, i.e., $C_{ij} = C_{ji}$.

Let us consider the tag co-occurrence matrix shown in Table 1(a). From the table, we can see for example that tag 0 and 3 occur together in total 3 times. The character "-" represents 0 as we define the co-occurrence of a tag with itself to be 0. We used the symbol "-" to differentiate from the case where two tags (not identical) do not co-occur with each other (co-occurrence labelled with 0). The total number of similarity pairs, given n tags, is $(n^2 - n) \times 0.5$. For the co-occurrence matrix in Table 1(a), we have to compute $(6 \times 6 - 6)/2 = 15$ cosine similarities.

(a)

Tag	0	1	2	3	4	5
0	-	2	1	5	2	0
1	2	-	7	1	1	0
2	1	7	-	3	0	2
3	5	1	3	-	1	0
4	2	1	0	1	-	6
5	0	0	2	0	6	-

(b)

Tag	1	2	3
0	2	1	5
1	-	7	1
2	7	-	3
3	1	3	-
4	1	0	1
5	0	2	0

(c)

Tag	0	4	5
0	-	2	0
1	2	1	0
2	1	0	2
3	5	1	0
4	2	-	6
5	0	6	-

Table 1. An example of a tag co-occurrence matrix, shown in (a). The co-occurrence matrix is split in two equally sized parts, shown in (b) and (c).

In practice, a tag co-occurrence matrix is sparse, i.e., it contains many zero values. This is because a tag on average only co-occurs with a small subset of the total set of tags. We can make use of this sparsity property in order to improve the scalability of any pair-wise similarity measure that is dependent on the dot product between two vectors. For example, the dot product between vectors **a** and **b** gives the cosine similarity, assuming the data is normalized to unit length vectors. This makes the cosine similarity a candidate for our algorithm. In the rest of this section, we explain our algorithm in the context of tag spaces, where input vectors are tag co-occurrence vectors.

The basic idea of our algorithm is to construct clusters of tag vectors from the original matrix, based on the position of the non-zero values. Because the matrix is symmetrical, it does not matter whether we cluster the columns or rows of the matrix, but for sake of clarity we assume that we cluster the columns. Tables 1(b) and 1(c) show two possible partitions (i.e., clusters) that could be obtained from the co-occurrence matrix that is shown in Table 1(a). For these two smaller matrices, we calculate the similarity only for the pairs within a cluster. For example, we do not compute the similarity between tag 3 and tag 4, as they appear in different clusters. Using this approach, the number of similarity computations that have to performed is $((3 \times 3 - 3)/2) \times 2 = 6$. This is a reduction of 60% on the total number of computations, as we have to compute the similarity only for 6 pairs instead of 15 pairs.

In order to cluster each column (i.e., tag vector) from the co-occurrence matrix, we compute a hash value for each column. This is done by first splitting each column in a predefined number of equally sized parts. Then, for each column, the relative weight of each part is computed. This is done by dividing the sum of the values in a part by the sum of the values for the whole column. After that, the columns are clustered based on the most important parts. The most important parts are defined by the smallest set of parts for which the sum of values in the parts is larger than some predefined percentage of the total column sum. This process is best explained with an example. Suppose we have the tag vector $[0, 6, 4, 0, 0, 0, 1, 0]^{\mathrm{T}}$. Now, consider we choose to split this column representation in 4 parts. For each of these 4 parts, we compute the sum of the values in that part, as shown in Table 2(a). The next step is to calculate the total sum of the

(a)

Part	Sum	Indices
1	6	0, 1
2	4	2, 3
3	0	4, 5
4	1	6, 7

(b)

Part	Score	Hash
1	0.545	1
2	0.364	1
3	0	0
4	0.091	0

Table 2. Table (a) shows the part scores and (b) the part statistics for vector $[0, 6, 4, 0, 0, 0, 1, 0]^{\mathrm{T}}$. We can see that the first and second part are the most important parts of this vector.

vector values and represent the previously determined sums as percentages of the total sum, which we call the relative score. In our example, the total sum is found by taking the sum of the values in the parts $6 + 4 + 0 + 1 = 11$. After dividing the computed sum for each part by the total sum, we obtain the relative scores, as shown in column 2 of Table 2(b).

The goal of the proposed algorithm is to cluster columns that have the same distribution of important parts, i.e., parts that have a large relative score. The algorithm pursues this goal because the similarity between two vectors will be high if the two vectors have the same important parts, assuming that the similarity measure depends on the dot product between two vectors (such as the cosine similarity). If two vectors do not share the same important parts, i.e., there are not many indices for which the vectors both have non-zero values, the similarity will be low. In order to cluster the tag columns based on this criteria, we compute a hash value based on the distribution of the relative scores of the parts. This is performed by creating a binary representation of each column of relative scores, where each part in the column is represented by a bit.

We define the parameter α as the minimum sum of relative scores for each column. We select the minimum number of parts for which the sum of the scores is larger than α. First, we sort the relative scores of each vector parts. In the previous example, if $\alpha = 0.75$ (75%), we first add part 1 (with a score of 0.545) to our list. As we do not reach 0.75 yet, we add the next largest part to our list, which is part 2 in this example. Now, we have selected $0.545 + 0.364 = 0.909$, which is larger than α. We can set the bits for parts 1 and 2 to 1, Table 2(b) shows the binary representation for our example in the third column. Again, we can observe that for this vector the most important parts are at the top, as the value for the top two parts is 1 and for the two bottom parts it is 0.

Algorithm 1 gives the previously described process in pseudo-code. The algorithm defines the function $\mathrm{s}(t, k)$, which computes the relative part scores for a tag t using k parts, and the function $\mathrm{h}(sc, \alpha)$, which is used to compute the binary hash for a vector with scores sc and using a threshold α. For the $\mathrm{h}(sc, \alpha)$ function, we currently use the quick sort algorithm to sort the k part scores after which we select the top scores based on the α threshold. Because quick sort has an average $O(p \log(p))$ time complexity, with p being the size of the sorted list, one can verify that our algorithm has time complexity $O(n(k \log k))$, where n is

Algorithm 1 Hash-based clustering algorithm

Require: The input: a tag co-occurrence matrix T
Require: The algorithm parameters:

- k, in how many parts the vector should be splitted,
- α, the minimum percentage of the total column sum required to compute the binary representation.

Require: The algorithm functions:

- $\mathrm{s}(t, k)$, computes the relative part scores for a tag t, using k parts,
- $\mathrm{h}(sc, \alpha)$, computes the hash for a vector with relative scores sc, using threshold α.

for each tag column $t \in T$ **do**
  $C = \{\}$ {C is a set of (cluster,hash) pairs}
  $score_t = \mathrm{s}(t, k)$ {vector with part scores of tag t}
  $hash_t = \mathrm{h}(score_t, \alpha)$ {binary encoding of tag t}
  if $\exists c$ s.t. $(c, hash) \in C \wedge hash_t = hash$ **then**
    $c = c \cup \{t\}$ {add t to existing cluster}
  else
    $c' = \{t\}$ {otherwise, a new cluster is created}
    $C = C \cup \{(c', hash_t)\}$ {add to set of clusters the newly created cluster}
  end if
end for

the number of tags and k is the number of parts. This means that our algorithm performs linear in time with respect to the number of tags.

There is a trade-off between the number of clusters and the accuracy of the algorithm. If we have a low number of clusters, the number of skipped high cosines will be relatively small but the reduction in the number of computations would be also small. For a small data set, splitting the matrix in 2 will give sufficient discriminant power, while a large matrix might need a split into 10 or even 20 parts. An important aspect of the algorithm is the parameter k, i.e., the number of parts a column is split into. For a given k, one can show that there are $2^k - 1$ possible binary hash representations. Because the number of clusters is dependant on the number of possible binary representations, the parameter k can be used to control the trade-off between the number of clusters and the accuracy of the algorithm.

The idea of the algorithm is to find clusters of columns such that the similarity between two tags located in different clusters is minimized. In this way, the algorithm indirectly selects column pairs for which one has to compute the similarity (intra-cluster tags) and column pairs for which the similarity is set to zero, i.e., pairs that are skipped (inter-cluster tags). The distribution of the number of columns in the clusters is important for the number of similarity pairs that are skipped. Ideally, one would like to have clusters that contain an equal number of tag columns. In this way, the number of similarity pairs that are skipped is

maximized. One can show that for $n \to \infty$, the reduction $\to 1/m$, assuming that the clusters are equally sized[2]. The maximal number of clusters is $2^k - 1$, where k is the number of parts.

In order to improve the 'recall' of the algorithm (i.e., not skipping high similarity pairs), we experiment with a heuristic that identifies clusters that potentially may have many high cosines and are difficult to be processed by our algorithm. One approach would be to remove the columns from our algorithm that have a value sum greater than some threshold. The idea behind this is that the higher the sum, the more the tag is co-occurring with other tags. For this group of tags that are removed from the data set, we have to compute all pairwise similarities with all other tags. The advantage of this approach is that less high similarities are skipped. The downside of this approach is that the number of computations that has to be performed increases. There is a clear trade-off between the number of computations and the number of skipped high similarities, as will become more clear in the evaluation.

4 Evaluation

For our experiments, we used a Flickr data set that has been gathered by the authors of [4]. The original data set contains 319,686 users, 1,607,879 tags, 28,153,045 pictures, and 112,900,000 annotations. We have selected a threshold on the number of times a tag is used. In the end, we selected the top occurring 50,000 tags. The reason for selecting the frequently occurring tags is that we want to eliminate the low-end outliers, which seldom co-occur with other tags and thus pollute the clustering process. At the same time we want to keep the data set size small enough in order to perform a brute force evaluation of all cosines for reference purposes. To be able to evaluate the performance of our algorithm, we needed to compute all cosines for this subset, which is in total 1,249,975,000 cosine computations. The used data set contains approximately 10 times more cosines between 0 and 0.1 than cosines between 0.1 and 0.2. This shows that there are many tags that are not similar to each other, which is common for data sets obtained from tagging spaces.

4.1 Experiments

In order to evaluate our algorithm, we have designed an experimental setup that covers a broad range of parameter combinations. Table 3 shows the ranges for different parameters that were used in the experiments. The total number of experiments is the number of unique combinations of the parameters, as shown in Table 3. For the parameter k, i.e., the number of parts a vector is split into, we chose a range of 3 to 50, with a step size of 1. For α, we chose the range 0.05 to 0.95, with a step size of 0.05. For the filter type (excluding a number of tags from the algorithm), we have experimented with two approaches. In the

[2] With n being the number of input vectors and m the number of clusters, we have $reduction(n,m) = \frac{\left((n/m)^2 - (n/m)\right) \times 0.5}{(n^2 - n) \times 0.5} \times m = \frac{(n-m)}{m(n-1)}$, $\lim_{n\to\infty} \frac{(n-m)}{m(n-1)} = \frac{1}{m}$

Parameter	Range
k (parts count)	min: 3, max: 50, step size: 1
α (hash threshold)	min: 0.05, max: 0.95, step size: 0.05
filter type	magnitude / sum
filter threshold	min: 50,000, max: 200,000, step size: 10,000

Table 3. Experimental setup.

first filter type approach we left out all tags that have a magnitude larger than a certain threshold and in the second approach we left out all tags that have a sum larger than a certain threshold. The last row of Table 3, the filter threshold, indicates the threshold used in the filter type. We varied the threshold for both the magnitude filter and the sum filter, from 50,0000 to 200,000 with a step size of 10,000.

In total, we performed 30,720 experiment runs (this is the total amount of unique parameter combinations). For each experiment run, we execute our clustering algorithm and store the resulting clusters. Using the clusters, we determine which tag pairs should be skipped, i.e., the similarity should be assumed to be 0. After determining which tag pair similarities are set to 0, we record the actual similarity values of these tag pairs for reference purposes. We also compute the average similarity of pairs of tags in the same cluster, which usually results in high similarities.

4.2 Results

Figure 1 gives an overview of the results. The figure shows, for different skipped cosine thresholds, the trade-off between the percentage of similarity pairs that has to be computed and the percentage of cosines that is higher than the threshold. So if a point is located on $(0.4, 0.1)$ then this means that 40% of the original number of computations has to be done (60% is skipped in total), but you skip 10% of the important cosines. The 'important' cosines are defined to be higher than the threshold used in the evaluation (0.4, 0.5, 0.6, 0.7, 0.8, and 0.9). For the tags that passed the magnitude/sum filter, we compute all pair-wise similarities to the other tags, as we do not use these tags in our clustering algorithm. The values that are reported for the x-axis include these combinations. Each point in a sub-plot of Figure 1 represents a parameter combination obtained from the experimental setup shown in Table 3. The ideal situation would be to have low values for both the x-axis as the y-axis (as close as possible to the origin), because then one has to compute relatively a small amount of the original computations while a low amount of the important cosine similarities is skipped.

We can make the two important observations from the results presented in Figure 1. First, it is clear that as the percentage of total similarity pairs that has to be computed increases, the percentage of skipped high cosines decreases. This is as expected because the probability of skipping important cosines trivially becomes smaller when more similarity pairs are computed. Second, we observe that

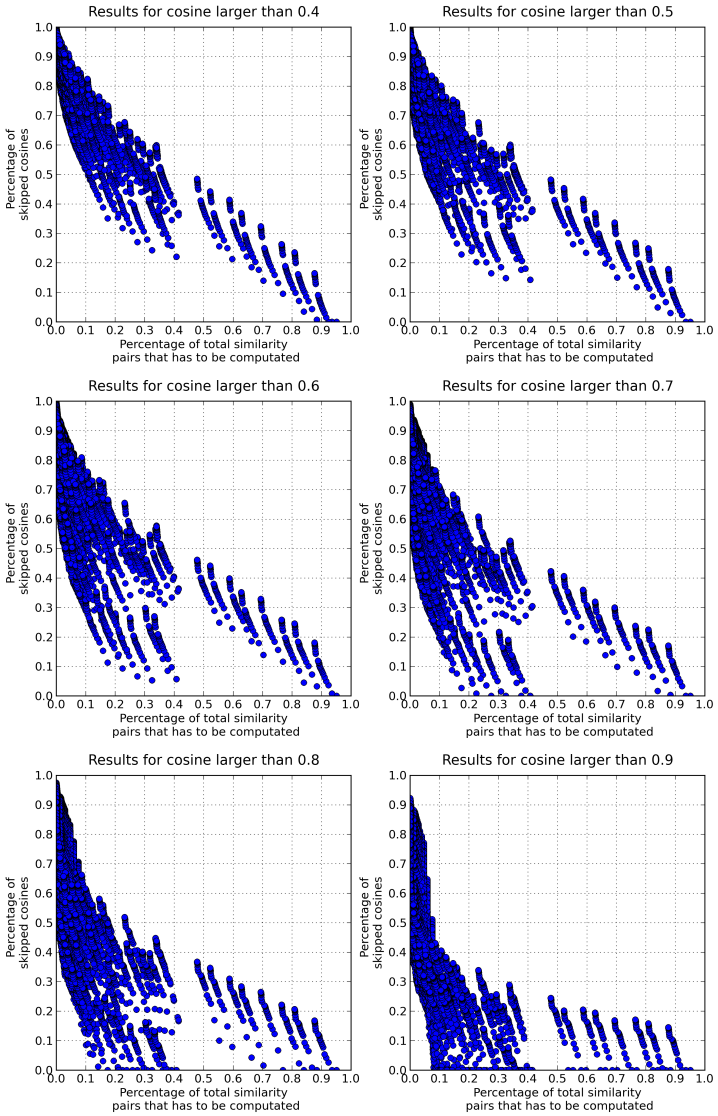

Fig. 1. In this figure we see a scatter plot of parameter combinations for different thresholds, with the x-axis showing the reduction of the total number of computations and the y-axis the percentage of skipped high cosines.

for cosine similarities ranging from 0.5 and higher, the algorithm is capable of just computing approximately 30% of the total number of computations while skipping a relatively low number of high cosines, i.e., approximately 18%. For the cases when 'high' cosines are considered to be 0.6 and higher, the results are even better. The algorithm is able to compute the significant cosines by just comput-

ing approximately 18% of the total number of combinations while just skipping 10% of the high ones. We find this already to be useful in computationally-intensive Web applications based on pair-wise similarities. When we consider higher thresholds we observe that the percentage of skipped high cosines further decreases. For example, for the case when cosines are considered to be high from 0.7 and onwards, the percentage of skipped high cosines is below 3% when computing 18% of the total number of combinations.

Although it is a bit difficult to see from Figure 1, the results show that if the user allows for less freedom in skipping high cosines (i.e., cosines higher than 0.6), the algorithm can be tuned to achieve a high cosine skipping percentage of approximately 5%, while having to compute around 30% of the original number of cosine similarities. As one can notice here, our algorithm can be tuned to meet various conditions. This shows the flexibility of our algorithm and its applicability to a wide range of Web applications (e.g., applications where a good trade-off between speed and quality is necessary, applications where speed is more important than quality, applications where similarity quality is more important than speed, etc.).

In order to understand how the parameters of our clustering algorithm influence the results, we have performed parameter sensitivity analysis. Table 4 shows information on some points (i.e., parameter combinations) from the plots given in Figure 1. We have chosen a few points that are interesting and need further explanation. The first part of Table 4 shows for each threshold the point that is closest to the origin. These points are the 'optimal' points when one gives equal weight to the percentage of computed cosines and to the percentage of skipped high cosines. For the thresholds 0.5 and 0.6, we notice that the optimal value

Threshold	Computations to be done	Skipped high cosines	Number of clusters	k	α	Filter type "sum >"	Filtered count
0.4	27.5%	27.1%	6	5	0.2	40,000	3.87%
0.5	17.5%	22.9%	29	7	0.2	40,000	3.87%
0.6	17.5%	11.3%	29	7	0.2	40,000	3.87%
0.7	14.2%	8.8%	37	8	0.2	40,000	3.87%
0.8	11.5%	5.6%	56	10	0.2	40,000	3.87%
0.9	8.0%	1.0%	1309	14	0.3	40,000	3.87%
0.4	76.9%	9.5%	7	3	0.85	40,000	3.87%
0.5	40.8%	14.3%	4	3	0.3	40,000	3.87%
0.6	22.5%	9.3%	6	5	0.2	40,000	3.87%
0.7	17.5%	2.7%	29	7	0.2	40,000	3.87%
0.8	17.5%	0.0%	29	7	0.2	40,000	3.87%
0.9	8.2%	0.0%	2803	22	0.2	40,000	3.87%

Table 4. This table shows a few interesting parameter combinations and their performance (e.g., the points that are closest to the origin) for each considered threshold. The first part of the table shows for each threshold the point that is closest to the origin. The second part of the table shows some points that might be useful in an application context where quality is more important than speed.

for parameter k is 7 and the optimal value for parameter α is 0.2. The number of clusters that is obtained using these parameter values is 29. The theoretical reduction (with the percentage of computations that have to be performed) is therefore $1/29 \approx 0.03$ for these two cases. For both the thresholds 0.5 and 0.6, we can also see that the observed reduction resulted in having to perform 17.5% of the total number of computations, which is approximately 6 times higher than the theoretical number of computations that could have been performed. This is probably due to the fact that there is still a large number of 'popular' tags present in the data set. The presence of these often occurring tags results in one or more large clusters. This makes the total reduction in the number of computations lower, as the tags are unequally distributed among the clusters.

The second part of Table 4 shows some points that might be useful in an application context. The reason for choosing these points is that they give a good trade-off between the number of computations and the skipping of high cosines, giving more weight to the latter. We can observe, for example, that for an application where high cosines are the ones that are higher than 0.4, it is necessary to compute approximately 76% of the cosines (and to have less than 10% skipped high cosines). The parameters for this situation are $k = 3$ and $\alpha = 0.85$. If we consider a different situation, where high cosines are the ones that are higher than 0.7, the optimal parameters change. With $k = 7$ and $\alpha = 0.2$ the algorithm is able to skip a relatively small amount (i.e., 2.7%) of the high cosines while computing just 17.5% of the cosines. In this way we retain most of the high cosines while performing a minimal amount of computations.

From the table we can also immediately notice that there is one filter that seems to give the best results, as for all rows this filter is found to be the optimal one. For this data set, this filtering is achieved by leaving out all tags of which the sum is greater than 40,000. The other filter, a threshold on the magnitude of a vector, seemed to give worse results. One final observation we can make is that the k parameter is more important and influential than the α parameter when considering the optimal points shown in the upper part of Table 4. The α parameter only seems to play an important role when considering a low threshold for high cosines. For the other situations, the parameter k determines the performance of the algorithm. A possible explanation of this is that the k determines the number of possible binary hash representations, and thus is the most influential parameter for the performance of the algorithm.

In order to understand in more detail how the parameters affect the performance of our algorithm, we also visualize a part of the sensitivity results. First, we focus on the reduction aspect of the algorithm, i.e., the factors that determine how many computations are skipped. Figure 2 shows a plot on the x-axis the considered values for the parameter k and on the y-axis the reduction of the number of computations as percentages. The different series in the plot each represent a value for the α parameter, as indicated by the legend. What we can observe from this figure is that in general the percentage of total combinations that has to be computed exponentially decreases as the number of parts (k) increases. We should note that an asymptotic behaviour seems to be present in

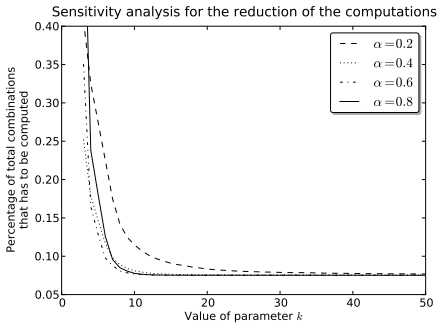

Fig. 2. A plot that shows the relationship between the values for parameters k and α and the reduction of the number of computations.

this plot, i.e., when k is larger than 10, the percentage of total combinations that has to be computed in general does not increase nor decrease. This can be explained by the fact that for $k > 10$ the number of possible binary hash representations becomes so large, that the upper bound of the actual number of unique hash representation for our data set is achieved.

We can further notice that $\alpha = 0.2$ converges the slowest. In Table 4 we already saw that 0.2 was the optimal setting for α in most cases. This stresses again the trade-off between the computation reduction power of the algorithm and the quality of the results. Figure 2 shows that even though the line $\alpha = 0.2$ is the slowest one converging, the overall reduction on the number of computations is relatively large and not much different from the other settings. One can note that for low values of k (less than 10), medium values of α (0.4 and 0.6) tend to provide for a better reduction in the number of cosines to be computed. Figure 2 also confirms our findings that the α parameter is not the ultimate determining factor of the performance of our algorithm.

Last, we have analysed how the parameters of the clustering algorithm influence the percentage of skipped high cosines, where we again consider different thresholds for the definition of a 'high' cosine similarity. Figure 3 shows 6 plots, one for each threshold, where the plots are similar to the plot in Figure 2, with the exception that the y-axis is now the percentage of skipped cosines for a particular threshold. From the figure we can observe that the percentage of skipped cosines grows with respect to the number of splits of a vector (parameter k). This is because of the number of clusters increases with k and thus the probability of skipping a high similarity increases. As a result of this, in general it is desirable to choose low values for k when it is more important not to skip high similarity pairs than to reduce the computational effort. When we consider the different α values, one can notice that a value of 0.2 gives the lowest amount of skipped high cosines, across all k values and considered cosine threshold values, something we already noticed in Table 4. Another observation is that for a threshold value of 0.7 or larger there are parameter combinations for which

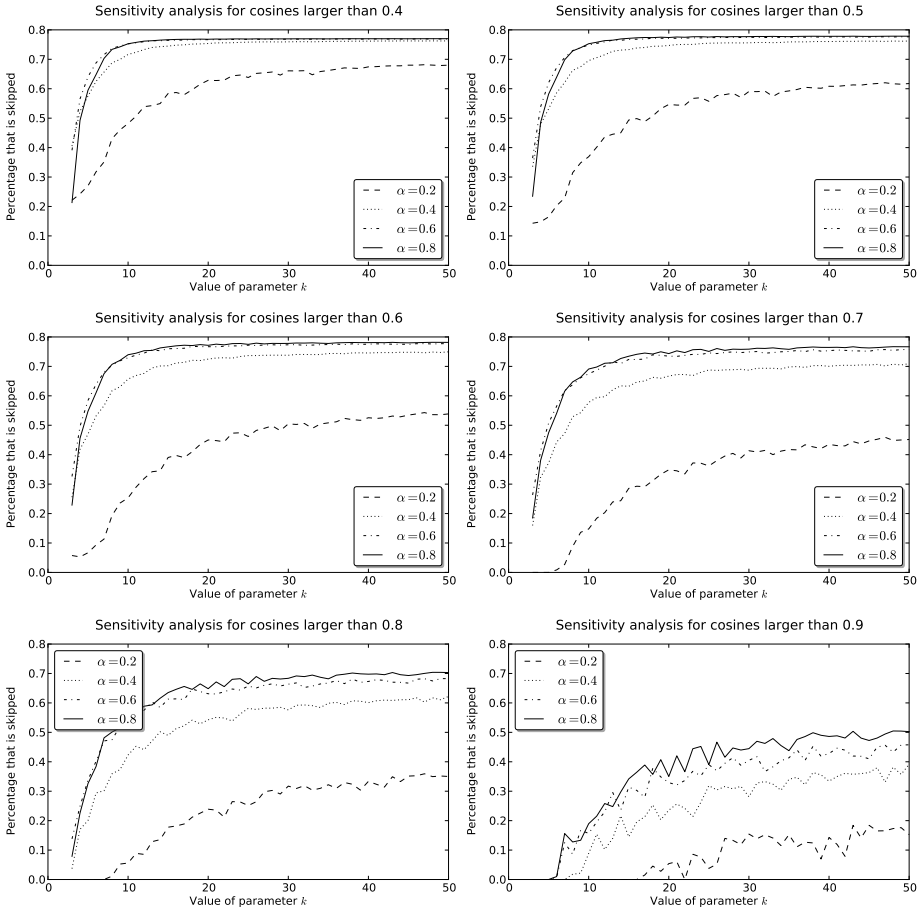

Fig. 3. An overview of how the parameters k (number of parts) and α (hash threshold), for different thresholds, influence the skipped high cosines.

no high cosines are skipped. Although the number of computations is probably high for these parameter combination, we still find it useful as it does decrease the computation time. Last, we can also observe that the asymptotic behaviour becomes less visible as we increase the threshold that defines what a 'high' cosine similarity is.

We have chosen to implement our algorithm and the experiments in Python. In order to efficiently deal with matrix algebra, we have made heavily use of the NumPy library [8]. For efficiently storing and querying large amounts of data, we have used PyTables [1]. PyTables provides an easy to use Python interface to the HDF5 file format, which is a data model for flexible and efficient input/output operations. The experiments were run on a cluster with nodes that had CPUs equivalent to a 2.5-3.5 GHz 2007 Xeon with 8 GB RAM. Each node was as-

signed to perform experiments for a set of parameter combinations. By running our code on 120 nodes, we cut down the computing time for the experiments and the computation of the cosine similarity for all pairs (our reference) from approximately 40 days to 5 hours.

5 Conclusions

In this paper we focused on the scalability issue that arises with the computation of pair-wise similarities in large tag spaces, such as Flickr. The main problem is that the number of similarity computations grows quadratically with the number of input vectors. Besides being large, the data sets in tagging space problems are usually sparse.

We have presented an algorithm that intelligently makes use of the sparsity of the data in order to cluster similar input vectors. In order to filter insignificant similarity pairs (i.e., similarities that are relatively low) we only compute the similarities between vectors that are located in the same cluster. The proposed algorithm performs the clustering of the input vectors in linear time with respect to the number of input vectors. This allows our approach to be applicable to large and sparse data sets.

For the evaluation of our solution we report the results for the cosine similarity on a large Flickr data set, although our approach is applicable to any similarity measures that are based on the dot product between two vectors. We have used an experimental setup that covers a broad range of parameter combinations. The results presented in this paper show that our algorithm can be valuable for many approaches that use pair-wise similarities based on the dot product (e.g., cosine similarity). The algorithm is, for example, capable of reducing the computational effort with more than 70% while not skipping more than 18% of the cosines that are larger than 0.5.

In order to gain more insight in how exactly the algorithm can be tuned, we performed an in-depth sensitivity analysis. From the results of this sensitivity analysis we can conclude that our proposed clustering algorithm is tunable and therefore applicable in many contexts. We have found that with respect to parameter k, the percentage of similarities that have to be computed is a decreasing function and the percentage of skipped high similarities is an increasing function. This means that if a high value for k is chosen, the algorithm produces better results with respect to the percentage of total similarities that have to be computed. When a low value for k is chosen, the algorithm skips a smaller number of high similarities. As for the parameter α, we have found that the effect on the performance of the algorithm is smaller than that of the parameter k. In general, there exists an optimal pair of values for k and α. The optimal values tend to be on the lower side of the considered values scale. Also, we find that increasing these values does not yield better results as the clustering algorithm performance seems to saturate at some point.

As future work one can consider the use of a clustering algorithm (e.g., 1-NN using kd-trees) for the binary hash representations of vectors as an extra

step in our approach. We would first like to investigate whether the overall performance of the algorithm is increased, and second, how the time complexity of the algorithm is changed by this extra step.

Acknowledgment

The authors are partially sponsored by the NWO Mozaiek project 017.007.142: Semantic Web Enhanced Product Search (SWEPS) and NWO Physical Sciences Free Competition project 612.001.009: Financial Events Recognition in News for Algorithmic Trading (FERNAT).

References

1. Alted, F., Vilata, I., et al.: PyTables: Hierarchical Datasets in Python (2012), http://www.pytables.org
2. Bayardo, R.J., Ma, Y., Srikant, R.: Scaling Up All Pairs Similarity Search. In: 16th International Conference on World Wide Web (WWW 2007). pp. 131–140. ACM Press (2007)
3. Cohen, J., Dolan, B., Dunlap, M., Hellerstein, J.M., Welton, C.: MAD Skills: New Analysis Practices for Big Data. VLDB Endowment 2(2), 1481–1492 (2009)
4. Görlitz, O., Sizov, S., Staab, S.: Pints: Peer-to-peer Infrastructure for Tagging Systems. In: 7th International Conference on Peer-to-Peer Systems (IPTPS 2008). pp. 19–19 (2008)
5. Halpin, H., Robu, V., Shepherd, H.: The Complex Dynamics of Collaborative Tagging. In: 16th International Conference on World Wide Web (WWW 2007). pp. 211–220 (2007)
6. Indyk, P., Motwani, R.: Approximate Nearest Neighbors. In: 13th Annual ACM Symposium on Theory of Computing (STOC 1998). pp. 604–613. ACM Press (1998)
7. Li, X., Guo, L., Zhao, Y.E.: Tag-Based Social Interest Discovery. In: 17th International Conference on World Wide Web (WWW 2008). pp. 675–684. ACM Press (2008)
8. Oliphant, T.E.: Python for Scientific Computing. Science & Engineering 9(3), 10–20 (2007)
9. Radelaar, J., Boor, A., Vandic, D., van Dam, J., Hogenboom, F., Frasincar, F.: Improving the Exploration of Tag Spaces using Automated Tag Clustering. In: 11th International Conference on Web Engineering (ICWE 2011). pp. 274–288. Springer (2011)
10. Specia, L., Motta, E., Franconi, E., Kifer, M., May, W.: Integrating Folksonomies with the Semantic Web. In: 4th European Semantic Web Conference (ESWC 2007), Lecture Notes in Computer Science, vol. 4519, pp. 624–639. Springer Berlin Heidelberg (2007)
11. TechRadar: Flickr reaches 6 billion photo uploads (2012), http://www.techradar.com/news/internet/web/flickr-reaches-6-billion-photo-uploads-988294