Quantifying Likelihood of Change through Update Propagation across Top-k Rankings

Evica Milchevski
TU Kaiserslautern
Kaiserslautern, Germany
milchevski@cs.uni-kl.de

Sebastian Michel
TU Kaiserslautern
Kaiserslautern, Germany
smichel@cs.uni-kl.de

ABSTRACT
Rankings are a widely used techniques to condense a potentially large amount of information into a concise form. However, rankings are dynamic and undergo changes, thus need to be maintained, which can be a tedious and expensive task. Given a ranking \( \tau \) that got updated to \( \tau' \), we aim at identifying those rankings \( \sigma \) that are very likely to have changed as well, as they are close in distance to the original ranking \( \tau \). We do so by modeling the expected change in form of a hypothetical ranking \( \sigma' \) and mark \( \sigma \) to require a refresh if the expected change is above a threshold. We do this for the Footrule distance and demonstrate through a preliminary evaluation the potential of our approach.

1. INTRODUCTION
We focus on the task of maintaining a set of crowdsourced entity rankings. One important characteristic of crowdsourced rankings is that although they share the same entities, they conform to different constraints, thus, a change in one ranking does not imply the same change in another ranking. We propose a framework that uses the similarity between the rankings, to reason about the degree of change in a set of rankings due to an update in one ranking. Since the distance between two rankings resembles not only structural but semantic similarity as well, it is reasonable to assume that once a ranking changes, it is more likely that similar rankings change, rather than dissimilar ones. Specifically for top-k rankings that only report on a (usually short) subset of items, if two rankings are similar, they need to share also a fraction of items. If ranking \( \tau \) changes, this means that items that are present in \( \tau \) changed, respectively their features. Such changes might or might not propagate to a ranking \( \sigma \) that is in distance \( \lambda \) to \( \tau \)—the likelihood of such a propagation is what we aim at quantifying in this work.

When considering rankings created over some objective (measurable) scoring function, like wealth in USD, the update of the rankings can be done by maintaining one global ranking, and directly updating all rankings affected by an update. However, keeping a global ranking in the case of crowdsourced rankings, where there is no measurable scoring function, but instead entities are ranked by some user perceived quality, like popularity, is not only expensive, but also unintuitive, as there is no ground truth.

1.1 Problem Statement
As input we are given a set \( T \) of top-k rankings \( \tau \). Each ranking \( \tau \) has a domain \( D_\tau \)—the items it ranks. We further know the Footrule distance between each pair of rankings in \( T \). We define an update \( u_{i,j} \) to a ranking \( \tau \) as a swap of two items \( i,j \in D_\tau \). A size of an update \( u_{i,j} \) for brevity denoted simply as \( |u_{i,j}| \), is the difference between the positions of the swapped items, \( |\tau(i) - \tau(j)| \). We denote with \( \tau' \) the ranking \( \tau \) after applying an update \( u_{i,j} \).

For a given update \( u \) over a ranking \( \tau \), the task is to compute the likelihood that \( u \) affects other rankings \( \sigma \in T, \sigma \neq \tau \) such that \( F(\sigma, \sigma') \) is larger than some user defined threshold \( \theta \). In that case, \( \sigma \) is marked to be refreshed (e.g., crowdsourced) to bring it up to date. If we believe a ranking is not affected by a change but in fact it is, we suffer loss in recall, the ranking is not refreshed and our database \( T \) is getting stale. If we, however, believe it is affected and it is not, we suffer loss in precision, which leads to wasting cost to refresh a ranking when it is not required to do so. For an overview to methods for comparing top-k rankings, see [1].

2. APPROACH
Algorithm 1 shows the procedure for determining (allegedly) affected rankings. As input the algorithm takes a specific update, a set of rankings, where the Footrule distance between all pairs is known, and a threshold \( \theta \). First, we compute the maximum distance, \( d_{max} \) that would likely result in the updated items \( i,j \) being present in both rankings \( \tau \) and \( \sigma \), i.e., \( i,j \in D_\tau \cap D_\sigma \Rightarrow F(\tau, \sigma) \leq d_{max} \), with some probability \( p \). We explain how this bound can be computed below. The next step is computing the expected change according to the distance. For this purpose, in a pre-processing step for each possible distance, for a given \( k \), we compute an average difference between the positions of the items in two rankings \( \tau \) and \( \sigma \), such that \( F(\tau, \sigma) \approx \lambda \). We call this average displacement (see Section 2.1) and it does not depend on the actual rankings in \( T \). Using the average displacements, we compute the expected change, according to the actual update. If the change is larger than a user specified threshold \( \theta \), we retrieve all distances and output all rankings within the retrieved distance to the changed ranking.

When we have a sequence of updates, we can simply accumulate the change. Note that an update over several items can also be considered as a sequence of updates of two items.

2.1 Computing the Expected Change
Since we do not know the positions, if any, of the affected items in the rankings, the first step toward quantifying the expected change in a ranking is reasoning about the most
Algorithm 1: Algorithm for determining all the rankings in $T$ affected by a change $u$, according to their distance.

1. $d_{max} = \text{get distance limit}()$
2. for each $E[\mu]$ in $\text{get expectations}(k)$ do
3.   $e_{\text{change}} = \text{get expected change}(u, \tau, E[\mu])$
4.   if $e_{\text{change}} > \theta$ then
5.     for each $d \leq d_{max}$, in $\text{get distances}(e_{\text{change}})$ do
6.       $R \leftarrow \text{get all rankings}(d, e_{\text{change}})$
7.     return $R$

3. EXPERIMENTS

We have implemented the described algorithm in Java 8. We created one synthetic dataset by first creating a small set of base rankings, by randomly choosing at least $\rho$ from $k$ items, where $k$ is the size of the rankings, and then randomly choosing the remaining $k - \rho$ items. All the other rankings are created by swapping a random number of items from the base rankings. The dataset contains 500 rankings with size $k = 10$. We compared our approach with the baseline approach—retrieving all rankings that have at least one item in common with the affected ranking. For the experiments, we randomly selected one ranking from the dataset, randomly selected a pair of items from this ranking, and then swapped their places. We then used our method and the baseline to find the affected rankings in the dataset.

Table 1 reports the average precision and recall for the two approaches over 100 trials. We report on results for two values of the threshold $\theta = 0.1$ and $0.15$. To compute $d_{max}$, we set $P(i, j) \notin D_{\tau} \cup D_{\sigma}$ to 0.9. Note that the recall of the baseline is always 1 since the relevant rankings must have at least one item in common with the changed ranking. We can see that with our approach we can achieve high precisions (much higher than the baseline) while still maintaining relatively high recall.

4. RELATED WORK

Research around crowdsourcing information usually addresses the problem of reducing the cost, while still retaining high quality results. Guo et al. [3] address the problem of finding the highest ranked object using the least number of questions, from a set of objects, in a crowdsourcing database system. Wang et al. [5] use transitive relations to reduce the number of questions asked to the crowd for the case of crowdsourced joins. Gruenhied and Kossmann [2] investigate the cost and quality trade-offs of different algorithms in a crowdsourcing environments. Polychronopoulos et al. [4] propose an algorithm for creating top-k lists using the crowd. The idea behind the algorithm is to create a high agreement top-k list for a low latency and monetary cost, by adaptively choosing the number of tasks posed to the crowd. To the best of our knowledge, there has not been any work that focuses on maintaining a set of crowdsourced rankings.

5. REFERENCES