Automated Category Tree Construction in E-Commerce

Uri Avron
Tel Aviv University
uriavron@mail.tau.ac.il

Shay Gershtein
Tel Aviv University
shayg1@mail.tau.ac.il

Ido Guy
Ben-Gurion University of the Negev
idoguy@acm.org

Tova Milo
Tel Aviv University
milo@post.tau.ac.il

Slava Novgorodov
eBay Research
snovgorodov@ebay.com

ABSTRACT

Category trees play a central role in many web applications, enabling browsing-style information access. Building trees that reflect users' dynamic interests is, however, a challenging task, carried out by taxonomists. This manual construction leads to outdated trees as it is hard to keep track of market trends. While taxonomists can identify candidate categories, i.e. sets of items with a shared label, most such categories cannot simultaneously exist in the tree, as platforms set a bound on the number of categories an item may belong to. To address this setting, we formalize the problem of constructing a tree where the categories are maximally similar to desirable candidate categories while satisfying combinatorial requirements and provide a model that captures practical considerations.

In previous work, we proved inapproximability bounds for this model. Nevertheless, in this work we provide two heuristic algorithms, and demonstrate their effectiveness over datasets from real-life e-commerce platforms, far exceeding the worst-case bounds. We also identify a natural special case, for which we devise a solution with tight approximation guarantees. Moreover, we explain how our approach facilitates continual updates, maintaining consistency with an existing tree. Finally, we propose to include in the input candidate categories derived from result sets to recent search queries to reflect dynamic user interests and trends.

CCS CONCEPTS

• **Applied computing** → *Electronic commerce*; • **Information systems** → *Clustering and classification.*

KEYWORDS

Category tree construction; E-Commerce;

ACM Reference Format:
Uri Avron, Shay Gershtein, Ido Guy, Tova Milo, and Slava Novgorodov. 2022. Automated Category Tree Construction in E-Commerce. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3526124

1 INTRODUCTION

Category trees enable browsing-style information access and play a central role in document directories, news sites, question-answering services, and e-commerce platforms. The construction and maintenance of category trees are infamously challenging [34, 35], typically carried out manually by taxonomists, which is expensive and time-consuming. The latter is particularly problematic when the item repository is massive and evolves rapidly along with user interests. To address this, many works devised effective automated construction and maintenance algorithms of tree-based categorizations. These, however, mostly focus on *term taxonomies* [30], where each tree node is associated with a single term, or *topic taxonomies* [19], where each node is associated with a set of topic-indicative terms. In contrast, e-commerce product trees, which are the focus of this paper, have rarely been addressed in a non-generic manner.

The distinct nature of the e-commerce setting stems from the conjunction of two characteristics. First, there are considerations guiding the construction of the tree that are not semantic, and cannot be derived strictly from the product metadata. Namely, platforms employ large teams of taxonomists, who are domain experts that derive the categorization based on domain knowledge and various commercial requirements and measures, such as consistency with user search patterns. The second characteristic relates to combinatorial restrictions imposed by all large e-commerce platforms, including Walmart, Amazon, and eBay. Concretely, to ensure a compact categorization of a massive repository, each product must be assigned to categories on only one (and in some cases two) path(s) from a root to a leaf (e.g., eBay lists an item in a single lowermost category for free, or two categories for an extra fee [1]).

To illustrate, consider categorizing shirts. These can be classified by various criteria, even at the same level of granularity, such as brand, price, gender, season, fabric, and so on. However, it is unclear from semantic information by which of these criteria and in what order should the shirts be recursively partitioned into tree subcategories. Since the categorization must be a tree where each item belongs to one or two branches, items cannot be listed in categories corresponding to all possible combinations of their properties. Given these restrictions, it is challenging to minimize the occurrence of the scenario where items matching a commonly searched criteria are either placed in a lowermost category with too many other items or, conversely, scattered across multiple tree categories, which hinders the utility of a tree-based search session.

While taxonomists can manually and algorithmically [23] identify numerous *candidate categories*, i.e. item subsets with a shared label, most such categories cannot simultaneously exist in a category tree, due to the aforementioned combinatorial restrictions. To address this, we devise a quantitative model to capture how well a category tree reflects the complete combinatorially-unrestricted representation of an item repository by candidate categories. More generally, we aim to offer an automatic construction tool to optimize the tree w.r.t. this objective and complement, rather than supersede, taxonomists, by enabling domain knowledge integration and offering customization for explicit tweaking of the solution.

To this end, we separate in this work the problem of identifying candidate categories from that of constructing a category tree that maximally reflects these item sets. Focusing first on the latter, we propose a rigorous approach for tree construction and quality evaluation, based on a score function that measures how well a category tree matches the candidate categories. Concretely, we measure how well each input set is captured by the similarity score of the most similar category in the tree to this set. Our problem has many variants, as the model schematically applies to numerous similarity functions (with additional parameters to control more granular requirements). Specifically, we examine variations of the Jaccard index and F_1 score, a variant where a category must contain the entire input set it matches, and the *Exact variant*, where the category must be identical to the matched input set. As each candidate category represents the items that match some criteria a user may have in mind when searching the tree, some item sets may be considered more important than others. Thus, input sets are weighted to reflect how valuable it is for a solution to contain a category that closely matches each set. While we proved in [13] strong inapproximability results for this model, in the present work we provide theoretical algorithms for practical special cases and extend these to general algorithms that we empirically demonstrate to significantly exceed these bounds. Our best-performing algorithm is based on a novel approach of harnessing practical solvers for the well-known Maximum Independent Set problem, with various improvements for special cases, whereas our second algorithm is based on the traditional approach of clustering.

As for deriving candidate categories, standard methods include crowd-sourcing [31] and clustering based on shared properties (e.g., shared keywords [26]). However, two challenges are identifying categories that reflect user demand and weighting them to predict how likely these will match users' search criteria. This is particularly important for low-level categorization, which, unlike standardized high-level categories (e.g., "Electronics" or "Fashion"), is dynamic and ambiguous [18]. To this end, we propose integrating into the input high-quality result sets for frequent search queries submitted to the search engine, derived from datasets maintained by every large platform. We advocate a data-driven approach, using the result sets either as part of the input or to set the weights of input sets based on the relative frequency of the queries that target them.

The following toy example illustrates how search queries can guide the tree construction to better serve users.

Example 1.1. Consider an online electronics store that sells cameras, phones, and various accessories for these products, such as memory cards. The existing tree, depicted on the left side of Figure 1, has two separate categories for memory cards: "Cameras"⟶"Memory Cards" and "Phones"⟶"Memory Cards". However, all memory

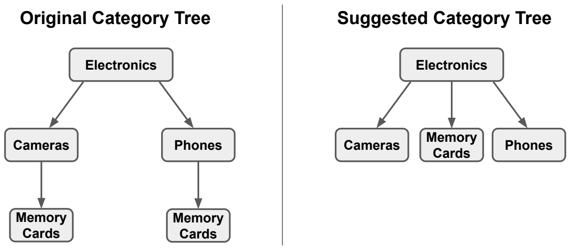


Figure 1: Sample category trees for the Electronics domain

cards are suitable for both product types. Assume that the most searched query is "memory cards", and that complete sets of all accessories for a given product type are rarely searched. The tree depicted on the right side of Figure 1 would then better serve the consumer needs, as it has a separate category containing all memory cards. The fact that these items no longer appear under the general categories has little effect, as the memory cards are rarely searched as part of more general item sets.

Apart from the model's evaluation metric, additional practical considerations must be addressed. Namely, the construction tool should be customizable, to allow tweaking the solution; the existing tree should be updated without radical modifications; the resulting categories should be semantically cohesive to allow labeling; and the tree structure should facilitate navigation. We outline in Section 2 how our approach captures the above considerations, as corroborated by both quantitative and qualitative evaluations.

The present paper builds on and extends two complementary works [8, 13]. Concretely, [13] is a theoretical paper that studies the approximation hardness of various categorization problems, including tree-based categorizations, that also capture as special cases the problem variants studied in the present paper. In particular, we showed that the problem of constructing an optimal category tree does not admit an efficient algorithm with good worst-case approximation guarantees (this is formalized in Section 2.4). Motivated by these theoretical results, we, therefore, in the current paper, devise heuristic algorithms that perform well in practice, and provide improved approximation guarantees for practical special cases. An early demonstration of a prototype of our system was accompanied by the short paper [8]. There we presented a brief overview of the system architecture, whereas the present work provides a comprehensive description of the model, algorithms and empirical analyses. Lastly, we note that our solution has been evaluated by XYZ[1] taxonomists, and we report their conclusions.

2 MODEL

We now present the formal model, discuss how it captures practical considerations, and provide inapproximability bounds.

2.1 Formal Setting

Input. The Optimal Category Tree problem (*OCT*) takes as input $\langle Q, W\rangle$, where $Q \subseteq 2^U$ is a set of n sets over a finite universe of items U, and each set in Q is assigned a non-negative weight by $W{:}Q \mapsto \mathbb{R}$. As mentioned in the introduction, each set in the input corresponds to a candidate category (e.g., a search query result set) that it is desired for the solution to contain.

[1]Company name omitted due to privacy considerations.

Problem variants. The *OCT* problem has multiple variants that differ based on the *similarity function*, $\mathcal{S}:[2^U] \times [2^U] \mapsto [0,1]$, used for the objective function, to be defined formally momentarily. The variant with similarity function $\mathcal{S}$ is denoted by $OCT(\mathcal{S})$.

Solution space. A solution to an *OCT* instance is a *category tree*, which is a rooted tree, where every node contains a subset of U and represents a *category*. A valid tree satisfies the following two requirements. First, every non-leaf category contains the union of the item sets in its child categories (and possibly other items). The root of the tree, thus, contains all the items, with the categories becoming more specific (smaller), as one moves down the tree towards the leaves. Second, each item in the tree belongs to exactly one most-specific category, along with all its ancestors. Thus, every item appears only in categories that are consecutively placed on some branch in the category tree, where a branch is a simple path from the root to a leaf. This requirement is ubiquitous in e-commerce platforms (e.g., [1]), as associating every item with a single branch ensures a compact categorization. Nevertheless, our algorithms are implemented to support a separate constant upper bound (that may exceed 1) for each item.

Objective. The goal is to have, for as many input sets as possible, a similar category in the solution, with the similarity evaluated by $\mathcal{S}$. Formally, given a set, $q \in Q$, and a category tree, T, the *similarity score* of a category $C \in T$ over q is $\mathcal{S}(q,C)$. The score of T over q is $S(q,T) = \max_{C\in T} \mathcal{S}(q,C)$. This definition captures the fact that a user seeks the category that most closely matches her (implicit) query. The overall score of T is defined as $S(Q,W,T) = \sum_{q\in Q} W(q) \cdot S(q,T)$. The weights are reflected in the score function, such that it is preferable to have matching categories for input sets of higher weight. The objective is to produce a category tree of the maximum score: $\arg\max_T S(Q,W,T)$.

2.2 Similarity Functions

Jaccard and F_1 variants. We consider variations of two prevalent set-similarity functions - the Jaccard index and the F_1 score. For completeness, we provide the definitions of these functions, w.r.t. a given input set, q, and a category, C. The Jaccard similarity is defined as $J(q,C) = \frac{|q\cap C|}{|q\cup C|}$. As for the F_1 score, it is the harmonic mean of the two more granular similarity measures: precision, $p(q,C) = \frac{|C\cap q|}{|C|}$, and recall, $r(q,C) = \frac{|C\cap q|}{|q|}$. Thus, $F_1(q,C) = 2\frac{p(q,C)\cdot r(q,C)}{p(q,C)+r(q,C)}$.

We extend these functions with a threshold parameter, $\delta \in (0,1]$, such that a similarity score below δ is rounded down to 0. This captures the fact that when the category is too dissimilar to the targeted item set, it is not identified as relevant by the user.

Correspondingly, we consider two variations of the above functions, referred to as *cutoff* and *threshold*. When $J(q,C) \geq \delta$, the *cutoff Jaccard similarity* equals $\bar{J}_\delta(q,C) = J(q,C)$, whereas the *threshold Jaccard similarity* equals $\hat{J}_\delta(q,C) = 1$. When $J(q,C) < \delta$, both functions equal 0. The cutoff and threshold variations of the F_1 score are defined analogously. Binary (threshold) functions serve to mollify possible inaccuracies in the input sets, by removing the incentive to overfit the categorization to match the input sets exactly, incentivizing instead the covering of more sets.

Perfect-Recall variant. We also study the binary Perfect-Recall function, $\mathcal{PR}_\delta$, that outputs 1 when the recall is 1 and the precision is at least δ. This variant is particularly relevant for supporting *faceted search*, where users reach a category, and then refine the item set via a filtering interface. For this search method, it is encouraged to include in the category complete input sets, as the side-effect of low precision, w.r.t. each user search that aims at a subset of this category, can be mitigated by the filtering mechanism.

Exact variant. Lastly, for $\delta = 1$, all similarity functions converge into the special case we call the *Exact variant*, where the score is 1 when the category is identical to the input set, and 0 otherwise.

Non-uniform thresholds. To simplify the presentation we focus on a uniform threshold. However, our algorithms support a separate threshold per each input set.

Cover terminology. We say that a category *covers* an input set if their similarity score exceeds the threshold. A set is *covered* if any category in the tree covers it. Thus, for threshold variants, the goal is to maximize the weight of the covered input sets.

Examples. We illustrate the *OCT* setting with the following toy examples, depicted in Figure 2. The figure presents two optimal solutions corresponding to two *OCT* variants, over the same input, provided on the left side. As the overall weight of all four sets is 5, this is also a bound on the score of any tree over this input.

For brevity and consistency, we will denote the items throughout the paper by short literal notation. Nevertheless, to provide a practical context, we show in Figure 3 a possible real-world instantiation of the items in both examples. Concretely, the 9 items depicted in the figure, that correspond to the items $\{a,b,...,i\}$ in Figure 2 (the corresponding item appears under each photo), are a sample of the shirts available in a company's catalog. These shirts have different brands ($\{a,b\}$ are Adidas, $\{c,d,e,f\}$ are Nike, $\{g\}$ is Puma, $\{h\}$ is Reebok and $\{i\}$ is Umbro), colors ($\{a,b,c,d,e\}$ are black, $\{f\}$ is red, $\{g\}$ is blue, $\{h\}$ is grey and $\{i\}$ is white) and sleeve lengths ($\{a,b,f,g,h,i\}$ have long sleeves and the rest have short sleeves). Then the sets q_1, q_2, q_3 and q_4 may correspond, respectively, to the result sets of the following 4 queries: "black shirt", "black adidas shirt", "nike shirt" and "long sleeve shirt".

Example 2.1. The tree T_1, depicted in the middle of the figure, is the optimal solution for the Perfect-Recall variant with $\delta = 0.8$. The categories C_3 and C_4 cover the sets q_2 and q_3, respectively, as they are identical to these sets. The category C_1 covers q_1 as its recall score is 1, and 5 out of the 6 items in C_1 are in q_1, hence the precision is $\frac{5}{6} > \delta$. Note that, we must include f in C_1 since it appears in C_4, and removing f from both categories would result in C_4 no longer covering q_3. Moreover, there is no incentive to place f elsewhere, since the score, when using a binary function, is not penalized for precision errors if the threshold is exceeded.

As for the category C_2, its addition to the tree is optional, since it does not cover any set, despite all its items belonging to the uncovered set, q_4, as we can no longer achieve perfect recall, without the items $\{a,b,f\}$. It is easy to verify that there is no way to cover q_4 by adding a matching category above or below C_1, such that the items $\{a,b,f\}$ would be shared by all categories, without decreasing the precision of other sets to values below the threshold. This logic, where certain subsets of the input are mathematically impossible to cover entirely either on the same tree branch or on separate branches, forcing the solution to give up on some input set in each subset, is formalized by the algorithm in Section 3.

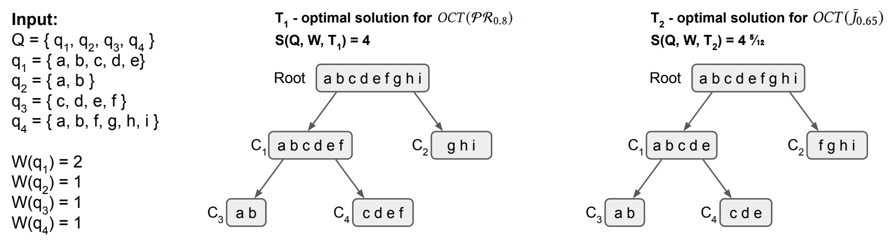


Figure 2: Optimal solutions for two *OCT* variants over the same input, depicted on the left side. The category tree, T_1, is an optimal solution for the Perfect-Recall variant with threshold parameter $\delta = 0.8$, where C_1 covers q_1, C_3 covers q_2, and C_4 covers q_3, with the overall score of $W(q_1) + W(q_2) + W(q_3) = 4$. The rightmost tree, T_2, is the optimal solution for the cutoff Jaccard variant with $\delta = 0.6$, where C_1 covers q_1 with the score of 1, C_2 covers q_4 with the score of $\frac{2}{3}$, C_3 covers q_2 with the score of 1, and C_4 covers q_3 with the score of $\frac{3}{4}$, resulting in the overall score of $W(q_1) \cdot 1 + W(q_2) \cdot 1 + W(q_3) \cdot \frac{3}{4} + W(q_4) \cdot \frac{2}{3} = 4\frac{5}{12}$.

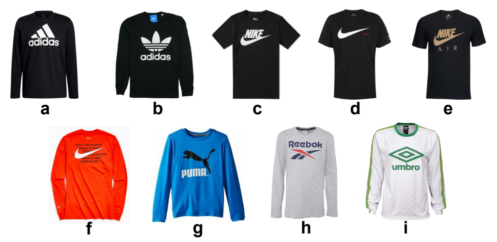


Figure 3: A sample of products from the shirts category.

Example 2.2. We next discuss, T_2, the optimal solution for the cutoff Jaccard variant with $\delta = 0.65$, depicted on the right side of Figure 2. It overlaps with T_1, except for the item f, which is placed in C_2 instead of C_4 and C_1. In this case, compared to the previously examined variant, since Jaccard variants allow for errors in both precision and recall, and since we use a lower threshold, it is now possible to cover all sets, albeit with imperfect scores. Indeed, every non-root category in T_2 covers an input set, as explained in the figure. Moreover, q_1 is the heaviest set, hence it is not surprising that the optimal tree covers it with a perfect score, at the expense of errors in the covers of less significant input sets. We note that, in practice, the same category often covers multiple sets. For instance, if we decrease the threshold from 0.65 to 0.4, then C_1 would also cover q_2, as its precision w.r.t. q_2 is exactly 0.4.

2.3 Practical Applicability

We now explain how our approach captures important practical considerations and contributes to complementary tasks. These considerations were also empirically verified by taxonomists during an extensive user study (the detailed results appear in Section 5.4).

Customization. Our solution is customizable to allow taxonomists to adjust the categorization. First, taxonomists may raise the weights of underrepresented candidate categories. Second, our algorithms schematically apply for numerous similarity functions, parameterized to provide granular control of how precisely each input set is matched. Finally, our algorithms also support a more general model with varying upper bounds on the number of same-level categories each item may belong to, as explained in Section 3.

Continual conservative updates. Our approach is suited for both generation and maintenance of category trees, which are often tackled separately [35, 36]. An important concern is ensuring that the new tree would not be radically different, to maintain consistency. One solution consists of adding the categories of the existing tree as additional input sets, adjusting their weights and thresholds to modulate the extent to which the current categorization is preserved. A complementary solution is running the algorithms separately on selected subtrees, where changes are desirable. The effectiveness of this approach is verified in our user study.

Labeling. The problem of meaningfully naming categories has been studied in various settings [9] and is outside the scope of this work. Nevertheless, we demonstrate that our solution produces semantically cohesive categories, which naturally lend themselves to succinct labeling. This stems from preprocessing the input such that each set corresponds to an explicit property or a coherent query. The accuracy requirements ensure that the categories are sufficiently similar to these sets, preserving cohesiveness. Lastly, we mark each category with the sets it matches, and their labels (a shared property or a search query) naturally hint at a name (if a category matches multiple sets, the precision ensures a large overlap, indicating a similar label). We demonstrate this cohesiveness and naming compatibility via quantitative metrics and a user study.

Navigation. Taxonomists may aim for trees with various structural properties to ensure ease of navigation. To this end, our algorithms produce a tree consisting of the minimal number of categories necessary to achieve its score. Taxonomists are then free to add intermediate categories to aid navigation (our model allows introducing intermediate nodes without affecting the score).

2.4 Hardness Bounds

To conclude this section, we present hardness bounds, derived in [13], where the $\tilde{\Theta}(\cdot)$ notation hides negligible factors.

Theorem 2.3. *The cutoff and threshold Jaccard and F_1 variants are NP-hard to approximate below a $\tilde{\Omega}(\sqrt{n})$ factor ($n = |Q|$), whereas the Perfect-Recall variant is NP-hard to approximate below a $\tilde{\Theta}(n)$ factor, even when input sets may intersect by at most one item. Moreover, the Exact variant is NP-hard, even for input sets of size 3. All bounds also hold for the unweighted case with a single uniform threshold.*

We also proved similar bounds in [13] for a much more general and relaxed model, and, therefore, cannot hope to provide practical worst-case guarantees, even by a reasonable relaxation.

3 MIS-BASED ALGORITHM

Despite the inapproximability bounds presented in Section 2, we devise two *OCT* algorithms, that empirically produce trees of scores well above these bounds. In this section, we describe the Category

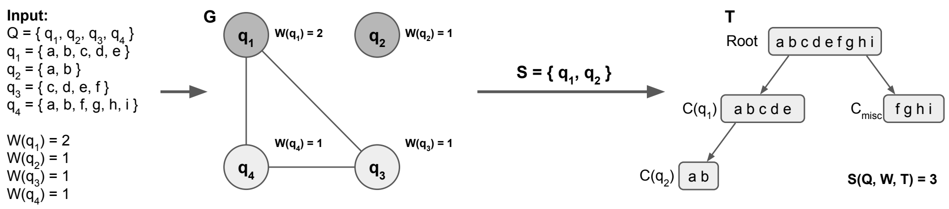


Figure 4: Execution of *CTCR* for the Exact variant, $OCT(\mathcal{PR}_1)$, over the input from Figure 2. The conflict graph, G, is depicted next to the input, where the shaded vertices form the optimal solution, S, for the *MIS* problem over G. The tree, T, which covers S (due to the optimality of S, T is also optimal), is depicted on the right side. Note that, it differs from the trees of Figure 2 as it is constructed w.r.t. the Exact variant ($\delta = 1$).

Tree Conflict Resolver (*CTCR*) algorithm, based on the novel approach of leveraging algorithms for the Maximum Independent Set problem (*MIS*). It takes the following approach: we identify pairs and triplets of input sets, referred to as *conflicts*, such that, for each conflict, it is mathematically impossible for any tree to cover all its sets simultaneously. We then leverage *MIS* algorithms to compute a conflict-free subset of input sets, which we aim to cover entirely. While *MIS* is also hard to approximate, there are known *MIS* algorithms shown to perform well in practice. Moreover, in all examined datasets, the derived *MIS* instances are sparse, which allows for improved performance and theoretical guarantees.

CTCR applies schematically to all considered similarity functions. Moreover, for the Perfect-Recall variant, it reduces to a simpler form, and, for the special case of the Exact variant, we derive an even simpler algorithm, that comes with tight approximation guarantees, w.r.t. the number of conflicts. We, thus, present the *CTCR* algorithm gradually. We first describe the simplest algorithm for the Exact variant, followed by its extension for the Perfect-Recall variant, and, finally, present the algorithm in its most general form, which applies to the Jaccard and F_1 variants. The algorithm is depicted at high-level in Algorithm 1, where we also mark which steps are unnecessary for the Perfect-Recall and Exact variants.

Algorithm 1: *CTCR*

```
1  $Rankings \leftarrow$ sort $Q$ ;                                  // only for $\delta < 1$
2  $CovT \leftarrow$ set pairs that can be covered together
3  $CovS \leftarrow$ set pairs that can be covered separately
4  $2Con \leftarrow$ 2Conflicts$(Q, CovT, CovS)$
5  $CovT \leftarrow CovT \setminus CovS$ ;                          // only for $\delta < 1$
6  $3Con \leftarrow$ 3Conflicts$(CovT)$ ;                            // only for $\delta < 1$
7  $Con \leftarrow 2Con \bigcup 3Con$ ;                             // only for $\delta < 1$
8  $G \leftarrow$ ConflictHypergraph$(Q, Con)$ ;                   // only for $\delta < 1$
9  $G \leftarrow$ ConflictGraph$(Q, 2Con)$ ;                        // only for $\delta = 1$
10 $S \leftarrow$ SolveMIS$(G)$
11 $T \leftarrow$ empty category tree
12 foreach $q \in S$ do
13     $T$.AddCategory$(C(q))$
14 foreach $q \in S$ do
15     $T$.AssignParentCategory$(C(q), S, CovT, Rankings)$
16 foreach $q \in S$ do
17     $C(q)$.items $\leftarrow$ ItemsCoveredTogether$(q, S, T)$
   // only for $\delta < 1$
18 foreach $q \in S$ do
19     $C(q)$.items $\leftarrow C(q)$.items $\bigcup$ DescendantsItems$(C(q), T)$
20 AssignItems$(S, T)$ ;              // Call Algorithm 2; not for $\mathcal{PR}_\delta$
   // not for $\mathcal{PR}_\delta$
21 foreach $C \in$ NonLeafCategories$(T)$ do
22     while $|$Children$(C)| > 2$ and HasInteresectingChildSets$(C)$ do
23         AddIntermediateCategories$(C)$ ;
24 $T$.RemoveNoncoveredItems$(S)$ ;                              // only for $\delta < 1$
25 $T$.RemoveNoncoveringCategories$(S)$ ;                        // only for $\delta < 1$
26 $T$.AddCategoryWithUnassignedItems$(Q)$
27 return $T$
```

3.1 Exact Variant

The *CTCR* algorithm for the Exact variant consists of three stages. In the first stage (lines 2 – 4 in Algorithm 1), we identify all 2-*conflicts*, where a 2-conflict is a pair of input sets that cannot be covered simultaneously by any tree. It follows that the optimal tree score is upper-bounded by the weight of the maximum-weight conflict-free subset of input sets. For the Exact variant, this bound is tight, and computing this subset is equivalent to the *MIS* problem. To that end, in the second stage (lines 9 – 10), we cast the problem as an *MIS* instance. While, in the general case, *MIS* is as inapproximable as *OCT*, the problem can, nevertheless, in practice, be reasonably approximated and often solved exactly [7, 20, 22]. Let S denote the produced independent set (*IS*), we then, in the third stage (lines 11 – 17), construct a tree T, where, besides the root, there is, for each $q \in S$, a category $C(q)$, identical to q. Then, we assign all unused items to a separate category (line 26). Note that the lines of Algorithm 1 that have not been mentioned so far are not relevant for the Exact Variant and will be discussed in the sequel.

We show that T is a valid solution covering S and that the performance ratio of *CTCR* equals the performance ratio of the *MIS* algorithm. By using the *MIS* algorithm in [7], we derive the following guarantee, whose tightness follows from the hardness proofs in [13]. We note that, in our experiments, *CTCR*, using the *MIS* algorithm from [22], solved all instances optimally and efficiently.

Theorem 3.1. *For each $q \in Q$, let $C2(q)$ denote the number of 2-conflicts containing q, and let $C2(Q, W) = \frac{\sum_{q \in Q} W(q) \cdot C2(q)}{\sum_{q \in q} W(q)}$ denote the weighted average number of conflicts per input set. Then, CTCR for the Exact variant has a tight performance ratio of $O(C2(Q, W))$.*

Algorithm 2: *AssignItems*

```
1  $Dups \leftarrow$ UnassignedItems$(S, T)$ ;
2  $UC \leftarrow$ Uncovered$(S, T) \cap$ CanBeCovered$(S, T, Dups)$ ;
3  while $UC \neq \emptyset$ do
4      $\hat{q} \leftarrow \arg\max_{q \in UC}$ Gain$(q)$ ;
5      $k \leftarrow$ CoverGap$(\hat{q})$ ;
6      $I \leftarrow$ TopKItemsByBranchGain$(k, \hat{q}, T)$ ;
7      AssignToBranch$(I, C(\hat{q}))$ ;
8      $Dups \leftarrow$ UnassignedDuplicateItems$(S, T)$ ;
9      $UC \leftarrow$ Uncovered$(S, T) \cap$ CanBeCovered$(S, T, Dups)$ ;
10 foreach $i \in Dups$ do
11     $\hat{C} \leftarrow \arg\max_{C \in \text{Categories}(T)}$ MarginalGain$(i, C)$ ;
12     AssignItemToCategory$(i, \hat{C})$ ;
```

We next discuss each stage of the algorithm in more detail.

Identifying 2-conflicts (lines 2-4). We say that input sets are *covered together* if they are covered by categories (or one category) on the same branch, whereas sets are *covered separately* if they are covered on different branches. Two sets that can be covered neither together nor separately are said to form a 2-*conflict*. Intuitively, for any *OCT* variant, the more items two sets have in common, the less likely it becomes for them to be covered separately, as the shared items must be partitioned. Conversely, the fewer items two sets have in common, the less likely it becomes for the sets to be covered together, as the higher-placed category must also contain the irrelevant items of the lower category. Therefore, two sets form a 2-conflict, when they are neither sufficiently similar nor sufficiently dissimilar. In the Exact variant, two sets can be covered separately when they are disjoint and can be covered together when one contains the other. Therefore, two sets form a 2-conflict if and only if they are neither disjoint nor one contains the other.

Reduction to MIS (lines 9-10). Once we have identified all 2-conflicts, we construct an *MIS* instance, G, called the *conflict graph*. The vertices of G are Q (vertex weights are the set weights), and the edges are the 2-conflicts: $\{(q_i, q_j) \mid q_i \cap q_j \neq \emptyset,\ q_i \not\subseteq q_j,\ q_j \not\subseteq q_i\}$. Over G we run the exact *MIS* algorithm in [22] that has been shown to solve the problem optimally and efficiently on massive instances.

Tree construction (lines 11-17). Given a conflict-free set of input sets, S, produced over G, the tree T contains, for each $q \in S$, a category, $C(q)$, identical to q. If a set, $q \in S$, is not contained in any other set in S, then the parent of $C(q)$ is the root. Otherwise, the parent of $C(q)$ is $C(q')$, where q' is the smallest set that contains q. This is well defined, since if two distinct sets of the same size had both contained q, then they would form a 2-conflict.

It follows that any two categories on the same branch pertain to two input sets where one contains the other. Moreover, every item appears in at most one branch, otherwise, it would appear in two categories for sets that form a 2-conflict. Thus, T is a valid tree that covers S, the entire *IS*. Conversely, let S' denote the set of input sets covered by a given tree T', then S' is an *IS* in the conflict graph, since, by definition, it contains no conflicts. Hence, the performance ratio of *CTCR* equals that of the *MIS* algorithm.

Finally, as the last step of the algorithm (line 26), we add, under the root, a new category, denoted by C_{misc}, with all unassigned items. In the case of the Exact variant, the unassigned items are all the items that do not appear in any set in S. This step does not affect the score, hence from a mathematical perspective, we can always generically assign all such elements to the same category. However, a more practical solution, reported in our user study (Section 5), is reemploying the algorithm with reduced thresholds for uncovered queries, as explained at the end of this section.

The operation of *CTCR* for the Exact variant is demonstrated in Figure 4 over the input provided in Figure 2.

3.2 Perfect-Recall Variant

The algorithm presented in the previous subsection applies to the restricted case where $\delta = 1$. We now explain what is needed in order to extend it to apply for $OCT(PR_\delta)$ instances, where $\delta < 1$.

For $\delta < 1$, it is insufficient to resolve all 2-conflicts to ensure that every two categories on the same branch cover sets that can indeed be covered together. To overcome this, we, in addition to 2-conflicts, also identify 3-*conflicts* (lines 5 – 6), which are triplets of sets such that no tree can cover a triplet simultaneously. Consequently, the conflict graph becomes the *conflict hypergraph* (lines 7 – 8), as it contains hyperedges of sizes 2 (regular graph edges - vertex sets of size 2) and 3 (vertex sets of size 3). Over the conflict hypergraph we employ the partitioning-based *MIS* algorithm in [15], suited for sparse hypergraphs (line 10).

In principle, one could also consider conflicts of a higher order, however, these cannot be listed exhaustively in sub-exponential time. Moreover, not considering conflicts beyond triplets is not an arbitrary stopping point, rather resolving 3-conflicts ensures that every two categories on the same branch cover sets that must be covered together, as was the case, by definition, in the Exact variant. To that end, before identifying the conflicts, we sort the sets, to help determine in advance all ancestor-descendant relations. Knowing which categories will be placed on the same branch, and in what order, allows to identify more conflicts w.r.t. the designated tree.

Lastly, the algorithm concludes with a procedure that condenses the tree (lines 24-25). This is unnecessary in the Exact variant, where all non-root categories cover an input set, and no items are redundant since all precision scores are optimal.

We next explain in more detail each of the stages of the algorithm that differ from the Exact variant version.

Sorting the input sets (line 1). The first step is to sort the sets from largest to smallest, and as a secondary criterion to sort by weight, from lightest to heaviest, breaking ties arbitrarily. We denote by $rank(q) \in [n]$ the ranking of the set q in this sorted order (the ranking of the largest set is 1). When checking whether two sets can be covered together, we may restrict ourselves to the case where the sets are covered by two separate categories, such that the higher category corresponds to the set of the lower ranking.

To optimize the score, providing a separate category for each set is preferable. To see this, consider a category C that aims to cover two sets, q_1 and q_2, where $rank(q_1) < rank(q_2)$. One can only increase the score by adding to C a child category, C', containing the same items as C, except those only relevant for q_1. This provides a second, improved opportunity to cover q_2. For this reason, among same-size sets, we assign a higher ranking to the heavier ones.

Identifying 2-conflicts (lines 2-4). In the case of the Perfect-Recall variant, two sets form a conflict when they are not disjoint (thus, cannot be covered separately), and also the size of the lower-ranking set is below a δ-fraction of the size of the union of the two sets (thus, they cannot be covered together, as the precision of the higher-placed category will be too low).

Identifying 3-conflicts (lines 5-6). We want every two categories on the same branch to correspond to sets that can be covered together, as is the case, by definition, for $\delta = 1$. However, we first need to determine which sets will be covered together, as some pairs can be covered both together and separately. In the Perfect-Recall variant, this only happens when two sets are disjoint and one is much larger than the other, such that it can be covered above it with sufficient precision. Our solution is to cover together only sets that *must* be covered together (i.e. can *only* be covered together), as computed in line 5. Correspondingly, for any q_1, q_2 and q_3, where $\{q_1, q_2\}$ and $\{q_2, q_3\}$ must be covered together and also q_2 is not the set of the lowest ranking of the three: if it is **not** the case that $\{q_1, q_3\}$ must also be covered together, then we, in line 6, add this triplet as a 3-conflict (unless $\{q_1, q_3\}$ is already a 2-conflict).

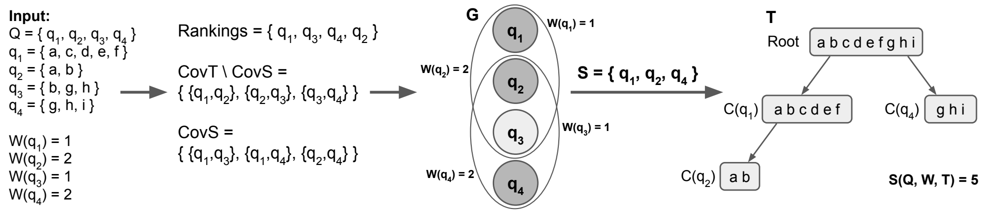


Figure 5: Example of an execution of *CTCR* for the Perfect-Recall variant, $OCT(\mathcal{PR}_{0.61})$. The input sets and their weights are depicted on the left side. Next to the input, from top to bottom, the figure shows the ranking of the input sets (Rankings), the pairs of input sets that must be covered together ($CovT \setminus CovS$), and the input sets that can be covered separately ($CovS$). Then, one column to the right depicts the conflict hypergraph, G, where the two ovals outline the two hyperedges and the shaded vertices form the optimal solution, S, for the *MIS* problem over G. The tree, T, which covers S, is depicted on the right side.

The following example illustrates why such a triplet is declared as a conflict, even when q_1 and q_3 can be covered together.

Example 3.2. Consider the sets $q_1 = \{a, c, d, e, f\}$, $q_2 = \{a, b\}$, $q_3 = \{b, g, h\}$ in the Perfect-Recall variant with $\delta = 0.61$ (also depicted in Figure 5). Each of the two pairs $\{q_1, q_2\}$ and $\{q_2, q_3\}$ are intersecting and must be covered together, whereas q_1 and q_3 can be covered both together (the higher-placed category, $C(q_1)$, would contain $q_1 \cup q_3$, with precision $5/8 > \delta$) and separately. If the conflict-free set of input sets, S, extracted from the conflict hypergraph, contains only q_1 and q_3, without q_2, then these would be covered separately, as they are disjoint. However, if all three sets appear in S, then all three must be covered together, and the merging of two otherwise separate branches may introduce conflicts.

Thus, to anticipate all conflicts in advance, we use the stronger condition above, which also improves the pairwise similarities within branches. Finally, we explain why we have included the condition that q_2 must not be the lowest-ranking set of the three. If this were the case, then its category would be the ancestor of the categories for the other two sets, and there is no contradiction when two descendants of the same category are covered separately.

Tree construction (lines 11-19). As in the Exact variant, given a conflict-free set of input sets, S, extracted from the conflict hypergraph (line 10), the tree T contains, for each $q \in S$, a category, $C(q)$ (lines 11-13). However, since, for $\delta < 1$, a covered set does not necessarily entirely contain a set covered below it, two steps in the tree construction are generalized. First, $C(q)$ is assigned (lines 14-15) the parent $C(q')$, where q' is the highest-ranking set of rank below $rank(q)$ that also satisfies the more general condition that it must be covered on the same branch as q, instead of the analogous condition for the Exact variant where q' must contain q. If no set satisfies this condition, then the parent is the root. Second, some items in sets covered below q may not be in q. Nevertheless, to produce a valid tree we must include them in $C(q)$ (lines 18-19), along with all the items of q (lines 16-17).

It follows that, for $\delta < 1$, a non-leaf category, $C(q)$, may not cover q, due to its descendants containing too many items irrelevant for q. While we ensured that $C(Q)$ can cover q when placed above each of its descendants individually, since we did not account for higher-order conflicts, the aggregate precision error may be too high. Nevertheless, we demonstrate empirically, in Section 5, that for the vast majority of the input sets, this is not the case. Moreover, our formulation of 3-conflicts is beneficial in this regard, increasing the similarities of the corresponding sets within each branch.

Condensing the tree (lines 24-25). In the final stage, we remove redundant items and categories, which is unnecessary for the Exact variant, since all categories covered the sets with perfect scores. First, we remove all items that only appear in uncovered sets (line 24, and later reassigned in line 26). Second, we remove (line 25) any non-covering category (if a set is covered by multiple categories, we retain the one with the highest precision). These operations may only increase the score, by improving precision.

The operation of *CTCR* for the $OCT(\mathcal{PR}_{0.61})$ variant is demonstrated in Figure 5. Note that there are only 3-conflicts. The reason why $\{q_1, q_2, q_3\}$ is a 3-conflict was explained above in Example 3.2, and an analogous explanation also applies to the other 3-conflict, $\{q_2, q_3, q_4\}$. These two conflicts correspond to the two hyperedges of size 3 in the hypergraph G. In this example, as all non-root categories cover an input set, there is no need to condense the tree. It is easy to verify that the final solution is optimal, as only one set of the lowest weight is not covered, which is the minimum possible uncovered weight when at least one conflict exists. Also note that, due to $C(q_1)$ containing the item b, the precision of the cover of q_1 is less than 1, which was impossible in the Exact variant.

3.3 General Algorithm

We are now ready to present the *CTCR* algorithm in its most general form. To enable a generic approach, the algorithm handles any threshold function as its cutoff counterpart, but never improves scores beyond the threshold at the expense of uncovering input sets. The algorithm differs from its simplified Perfect-Recall version by two additional consecutive stages, and the specifics of computing 2-conflicts are different for each variant. Both extensions are a consequence of allowing recall errors. The first new stage (line 20 in Algorithm 1, which calls Algorithm 2) generalizes the assignment of items to categories, as now items may belong to separately covered sets and if so, must be partitioned. Correspondingly, we, in the second new stage (lines 21-23 in Algorithm 1), add intermediate categories, that recombine some of the partitioned item sets.

Next, we describe the modifications in more detail.

Identifying 2-conflicts (lines 1-4). We provide here the specific computations for identifying conflicts in Jaccard variants. The formulas for F_1 variants are derived analogously.

To check whether q_1 and q_2 can be covered separately, let $I = q_1 \cap q_2$, and let x_1 and x_2 denote the maximum number of items from I we can exclude from their covering categories, respectively. A simple computation yields for $i \in \{1, 2\}$, that $x_i = \min\{\lfloor |q_i| \cdot$

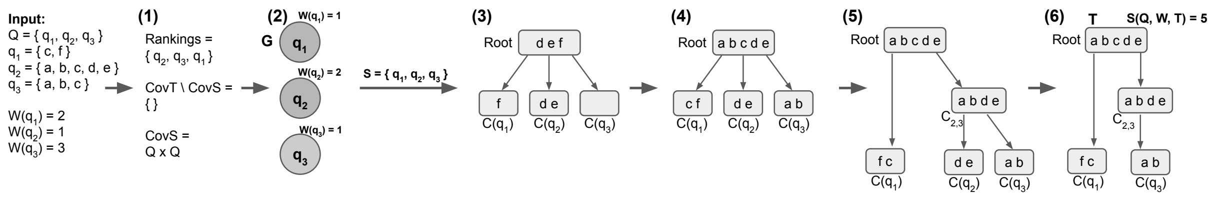


Figure 6: Example of an execution of *CTCR* for the threshold Jaccard variant with $\delta = 0.6$, $OCT(\hat{J}_{0.6})$. The input is depicted on the left side, and next to it 6 consecutive stages of the algorithm are depicted from left to right, marked (1) - (6). The first stage indicates that there are no conflicts, hence the hypergraph in stage (2) has no hyperedges. In stage (3) only non-duplicates are assigned, whereas in stage (4) the duplicates are also assigned. Stage (5) adds an intermediate category $C_{2,3}$, and in stage (6) the tree is condensed by removing the non-covering category.

$(1-\delta)\rfloor, |I|\}$. Thus, sets can be covered separately when $(|I| - x_1) + (|I| - x_2) \le |I|$, simplifying into $|I| \le x_1 + x_2$.

When checking whether these two sets, assuming $rank(q_1) < rank(q_2)$, can be covered together, since the category for q_1 also contains the items of the category for q_2, we need to make sure that, in the cover of q_2, we use as few items as possible from $q_2 \setminus q_1$. Specifically, the minimum number of items outside of I, that must remain in the cover of q_2, is $y_2 = \max\{0, \lceil \delta \cdot |q_2| - |I| \rceil\}$. Thus, the two sets can be covered together when $y_2 \le |q_1| \frac{1-\delta}{\delta}$.

Item assignment (line 20 in Algorithm 1, which calls Algorithm 2). We first, as is the case in the simplified algorithms, assign to each category, $C(q)$, all the items in q that only appear in sets that are covered together (lines 16-17 in Algorithm 1), and the items assigned to its descendants (lines 18-19 in Algorithm 1).

Unlike the Perfect-Recall variant, as it is possible to cover separately intersecting sets, some items in the conflict-free sets, we call *duplicates*, may be unassigned. To assign duplicates, we use an iterative greedy procedure, prioritizing covering sets of higher weight that are also closer to being covered, as captured by the following metrics. For each uncovered $q \in S$, its *cover gap* is the number of duplicates from q that, if added to $C(q)$, will suffice to reach the threshold. Correspondingly, the *gain factor* of q is the ratio of its weight to its cover gap.

In each iteration, we select the set $\hat{q}$ with the highest gain factor of the sets that can be covered by the remaining duplicates (line 4 in Algorithm 2). We then select a set, I, of duplicates from $\hat{q}$ of the size of its cover gap (lines 5-6)), as follows. For each relevant duplicate, we compute the sum of the gain factors of the sets that contain it on each branch that $C(\hat{q})$ is placed on (non-leaf categories are placed on several different branches, each ending in a different leaf), matching it with the branch where this sum is maximized. We then select the top duplicates, in terms of total gain, and assign each duplicate to the lowest relevant category on its matched branch (line 7). The set of unassigned duplicates is updated before each iteration (lines 1 and 8), as is the set of uncovered sets in S, which can still be covered by the remaining duplicates (lines 2 and 9).

Finally, once we cannot cover more sets, we assign the remaining duplicates iteratively, choosing the assignment with the highest marginal gain to the cutoff score (lines $10-12$).

Adding intermediate categories (lines 21-23 in Algorithm 1). When recall errors are allowed, intersecting sets may be covered separately. However, if two categories share a parent, we can add a category that recombines the partitioned shared items. Specifically, for each category with more than two children, we add as a new child an intermediate parent to any pair of child categories, that correspond to intersecting sets, containing their union. Each new intermediate category is considered to correspond to the union of the sets its child categories correspond to (and can itself, in a future iteration, be subject to a new intermediate parent). We iteratively add a parent for the two sets that share the largest fraction of items out of the smaller set, until there are either two child categories left, or no two categories correspond to sets that intersect.

Extensions. To avoid a convoluted presentation, we have omitted from the above description various extensions implemented in *CTCR*, which we briefly outline below.

First, to handle input sets with varying thresholds, note that when computing conflicts, pairs are examined separately and can be evaluated with different thresholds. Similarly, the item assignment targets sets separately and may each time use a different threshold.

CTCR also supports varying bounds on the number of same-level categories each item may be assigned to. Namely, when testing whether two sets can be covered separately, we allow in the computation for both categories to include any item whose bound exceeds 1. When assigning the items, each item is duplicated according to its bound. Moreover, if many item bounds exceed 1, we also accordingly test for conflicts of a higher order.

The operation of *CTCR* for the threshold Jaccard variant with $\delta = 0.6$, $OCT(\hat{J}_{0.6})$ is demonstrated in Figure 6, where the 6 consecutive stages of the algorithm are marked (1) - (6). The first stage concludes that all 6 pairs of input sets can be covered separately, hence there are no conflicts. Consequently, the conflict hypergraph G in stage (2) contains no hyperedges, and the optimal solution S is the entire vertex set. The tree in stage (3), thus, has 3 categories on separate branches that correspond to the 3 input sets, and the items $\{f, d, e\}$ that only appear in sets covered on the same branch are assigned. In stage (4) the duplicates are also assigned. A simple computation shows that q_1 has the highest gain factor of $2/1 = 2$ (as its weight is 2 and the addition of only one item is sufficient for $C(q_1)$ to cover it). The only relevant duplicate item is c. Then, q_3 has the next highest gain factor of $3/2$, and we must add to it the only two relevant remaining duplicates $\{a, b\}$. At this point only q_2 is not covered. However, in stage (5) we add the intermediate category $C_{2,3}$ that is the parent of C_{q_2} and C_{q_3} (this is the chosen pair since $q_2 \subseteq q_3$), and now covers q_2. The score of the tree is now optimal, however, it still has one non-covering category, C_{q_2}, that is removed when the tree is condensed in the final stage.

4 CLUSTERING-BASED ALGORITHM

While we have demonstrated empirically, in Section 5, the effectiveness of *CTCR*, nevertheless, as its performance is dependent on

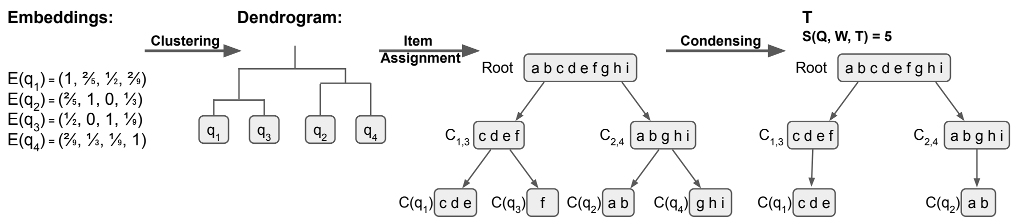


Figure 7: Execution of *CCT* for the threshold Jaccard variant with $\delta = 0.6$, $OCT(\hat{J}_{0.6})$, over the input from Figure 2. The embeddings are depicted on the left side. The dendrogram extracted from the agglomerative clustering is depicted to the right of the embeddings. To its right, we have the tree, T, based on this dendrogram, along with the item assignment. The tree is optimal as it covers Q entirely. The final condensed tree with two non-covering categories removed is depicted on the right side.

the underlying *MIS* algorithm, it may be desirable to provide an alternative algorithm, which avoids resolving conflicts directly. To this end, we present in this section the Clustering-Based Category Tree (*CCT*) algorithm. Unlike existing clustering-based categorization methods, which cluster the items directly (see Section 6), *CCT* clusters the input sets. It does so to derive the tree structure, and not to determine the item assignments. Concretely, we first employ a *hierarchical clustering* algorithm over Q, to derive the structure of the tree. This tree would have a leaf category for each set in Q, and the finer is the finest cluster two sets are assigned to, the lower their lowest common ancestor is placed in the tree. The items are then partitioned across the leaf categories using the same iterative greedy item assignment procedure as in the *CTCR* algorithm.

Unlike *CTCR*, instead of resolving conflicts in advance, *CCT* aims to do so implicitly. Specifically, for a conflict (q_i, q_j), if we first assign items to cover q_i, then q_j can no longer be covered, and the greedy assignment algorithm will prioritize other sets over q_j, such that *CCT* avoids wasting items on covering q_j.

CCT applies schematically to all *OCT* variants, as we explain momentarily. The algorithm is depicted at high-level in Algorithm 3. Moreover, its operation, over the input from Figure 2, for the threshold Jaccard variant with $\delta = 0.6$, is demonstrated in Figure 7.

Next, we describe each stage of the algorithm in more detail.

Input sets embedding (line 1 in Algorithm 3). Applying a clustering algorithm requires computing pairwise distances. The main novelty of our clustering approach is, instead of relying on pairwise similarities, to take into account the entire "global context". Namely, for Jaccard and F_1 variants, we embed every set $q \in Q$ as a vector $E(q) \in \{[0,1]\}^n$ in Euclidean space, where the i-th entry is $E(q)_i = \mathcal{S}(q, q_i)$, the similarity of q and the i-th input set, w.r.t. some ordering of the sets, as evaluated by similarity function.

The left side of Figure 7 depicts the embeddings of the input sets from Figure 2. As mentioned, the i-th entry in the vector $E(q_j)$, is the Jaccard similarity of q_i and q_j.

Deriving a clustering-based tree structure (lines 2-3). In the second step of *CCT*, we run an agglomerative clustering algorithm (line 2), which continually merges subsets of vectors (starting from singletons, and ending in a single set containing all the vectors), according to the euclidean distance (we have also examined other metrics, with inferior results) merging each time the two closest subsets. The distance of two subsets is the average of all the pairwise distances of the Cartesian product of the sets.

The execution of an agglomerative algorithm is represented using a dendrogram, a diagram shaped as a binary rooted tree depicting the order of the merges, with the earliest merges presented

Algorithm 3: *CCT*

```
1 $E \leftarrow$ ResolveEmbeddings($Q$)
2 $Dendrogram \leftarrow$ AgglomerativeClustering($E$)
3 $T \leftarrow$ ConstructTreeFromDendrogram($Dendrogram$)
4 AssignItems($Q, T$) ;                                    // Call Algorithm 2
5 $T$.RemoveNoncoveredItems($Q$)
6 $T$.RemoveNoncoveringCategories($Q$)
7 $T$.AddCategoryWithUnassignedItems($Q$)
8 return $T$
```

at the bottom. This dendrogram is then used as the template for the tree structure (line 3), determining completely all the categories.

Item assignment (line 4 in Algorithm 3, which calls Algorithm 2) and Condensing the tree (lines 5-7). Once the tree structure is determined based on the dendrogram, we assign the items to the categories using the same procedure, depicted in Algorithm 2, as in the *CTCR* algorithm (in the case of *CCT*, the items will only be assigned to leaf-categories, as the dedicated category for each input set is a leaf category), and we also condense the tree exactly as in *CTCR*. This is demonstrated in 7, following similar reasoning as the example in Figure 7.

For the Perfect-Recall variant, we use different embeddings, as the similarity function is based separately on precision and recall. Concretely, to combine both measures, we use their average, such that the i-th entry of the embedding of q is $E(q)_i = (r(q, q_i) + p(q, q_i))/2$ (note that $r(q, q_i) = p(q_i, q)$).

5 IMPLEMENTATION AND EXPERIMENTS

We open this section by explaining how we derived and preprocessed the input sets, in particular, the result sets to search queries, to best exploit our algorithms. We then describe, in the experimental setup, the real-life datasets (both private and public) we used for the experiments, the algorithms we compared, and the quantitative evaluation methods, followed by the evaluation results. Lastly, we describe the setup, methodology and findings of a user study, focusing on qualitative metrics and the scope of the manual intervention required for implementing our human-in-the-loop approach.

5.1 Data Preparation

As discussed in the introduction, we advocate using result sets to frequent search queries to facilitate a data-driven approach for deriving candidate categories. Thus, our analysis of the score achieved by each algorithm uses real-world datasets of raw search queries, as described below. We examined four public datasets and four private datasets provided by XYZ. We next explain how we preprocessed these raw search queries into the inputs used for our experiments. Concretely, we describe how we (1) removed noisy/incoherent queries, (2) cleaned query result sets, (3) assigned weights that reflect the relative importance of covering each query, and (4) heuristically merged queries to improve solution quality and scalability.

When devising our preprocessing scheme, we relied on objective arguments for improving the quality or efficiency of the solution, according to the experience of the taxonomists. In the description of the user study (Section 5.4), we also report conclusions of ablation tests, that highlight the preprocessing effectiveness.

Cleaning the query set. To omit incoherent, outdated, or rare queries not indicative of user demand, we take the following approach. First, as infrequent queries are less reflective of demand, we consider only queries that were submitted at least X (the exact number is confidential as consumer data can be derived from rare queries) times a day, consecutively over the last 90 days (XYZ reconstructs the tree every 90 days). However, in our user study we demonstrate that platforms can capitalize on short-lived trends, by applying the algorithms over data skewed towards more recent periods. Second, to remove nonsensical queries we omit queries whose result sets consist of items from more than 10 different branches in the existing company tree (fewer than 1% of the queries). This captures the logic that queries whose result sets are scattered across many distant categories are not indicative of one unifying category.

Computing result sets. The result sets are computed via the platform's search engine. When search engines evaluate queries, they provide a relevance score in $[0, 1]$ for each returned item. To reduce noise, we remove items whose score is below a certain relevance threshold. Taxonomists reported that including too long of a tail in the final result sets reduced the categorization coherence.

Experimentation with various values led to selecting a 0.8 relevance score for Jaccard and F_1 variants, and 0.9 for the Perfect-Recall and Exact variants. These thresholds ensured almost in all cases that each item in the repository appears in at least two sets.

Assigning weights. The weight of each query was set to the average number of times it was searched per day, over the examined period. In the public datasets, we assigned a uniform weight of 1 as all queries are distinct with no frequency data.

Merging similar queries. Lastly, we incorporated an optimization that improved the running time and the quality. Namely, we merged every two very similar result sets into a single set whose weight is the combined weight of the original sets. This reduced the number of queries by more than half in all XYZ datasets. Moreover, the scores were either the same or slightly improved, when evaluated over the original queries. Experiments by the taxonomists led to merging queries whose similarity, as evaluated by the similarity function, lies in $[\delta + \frac{3}{4}(1 - \delta), 1]$.

5.2 Experimental Setup

We implemented our algorithms [2] using Python and ran the experiments on a server with 128GB RAM and 32 cores. We compared our two algorithms to three baselines. These include the existing tree, which represents the approach taken by e-commerce platforms, as well as a modern clustering approach from the literature.

Input categories. We ran our experiments with different combinations of sources for deriving the input candidate categories. The three main sources were: (1) categories of the existing tree, (2) sets derived by taxonomists for each shared property explicitly recorded in the products database (e.g., size for TV screens), and (3) result sets of search queries. Unsurprisingly, the scores and the relative ranking of the algorithms were robust w.r.t. all considered input sources, and we thus show in the score-based evaluation representative results over the fully automated approach of using only result sets for frequent queries. Note, however, that the evaluation pertaining to the score function is not indicative of the quality of the ground truth input itself. Hence, the contribution of incorporating these queries is evaluated separately in the user study.

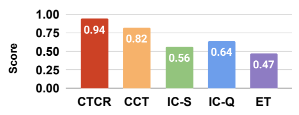
Datasets. The four private datasets, provided by XYZ, are named A, B, C, and D, and contain, respectively, 450, 1.2K, 3K, and 20K queries and their result sets. Note that these are the dataset sizes after the preprocessing. For example, D, the largest dataset, contained originally 100K queries prior to merging the queries.The numbers of distinct items, in these datasets, are 28K, 94K, 340K, and 1.2M, respectively. The first three datasets (A, B, C) are all taken from the Fashion domain, while D is extracted from the Electronics domain. These datasets are based on search queries submitted in the first quarter of 2020 and contain only English queries from the US site.

We also examined publicly-available datasets of various e-commerce platforms: CrowdFlower [3], HomeDepot [4] and Victoria's Secret [5], containing search queries and top-k results. We also examined the dataset of queries taken from BestBuy [12], and evaluated these queries over the Electronics subtree of Amazon [17]. Concretely, we indexed Amazon's products and computed the result sets using Elasticsearch [6]. When we report in the sequel our evaluation results over this dataset, we refer to it as E.

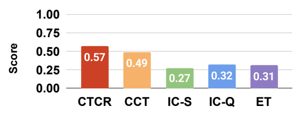
As the obtained results over all datasets demonstrated very similar trends, for space limitations, we provide representative results only for some private and public datasets. Most shown results are over the XYZ datasets, which are also the largest and most detailed datasets, containing query frequency information.

Algorithms. We compared our two algorithms, *CTCR* (Section 3) and *CCT* (Section 4), to the following three algorithms.

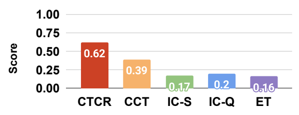
- **IC-S** - An adaptation to our context of an e-commerce-specific algorithm [18]. It takes a conventional approach in related settings (see discussion in Section 6) of embedding product titles, and employing a hierarchical clustering algorithm over the embeddings. It differs from *CCT* in two key aspects. First, IC-S clusters the items directly, unlike *CCT*, which clusters the input sets and then performs the item assignment separately. Second, IC-S does not take the input sets into account, rather it relies on extracting semantic information from item metadata. IC-S differs from the implementation of [18], by an optimization (verified to improve the scores), that uses title embeddings produced by a domain-specific model, trained by XYZ, instead of word2vec, and by replacing k-means with hierarchical clustering as in *CCT*, since [18] add only a single lower layer to an existing tree.
- **IC-Q** - Another algorithm that clusters the items directly. However, while IC-S clusters the items based on semantic information, IC-Q performs the clustering based on the input sets each item appears in. Thus, IC-Q can be viewed as a hybrid approach that combines *CCT* with IC-S. Concretely, IC-Q represents each item as a vector, where the i-th entry is 1 if the item appears in the



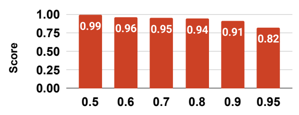
(a) Jaccard, $\delta = 0.8$ [C].



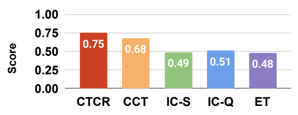
(b) PR, $\delta = 0.8$ [C].



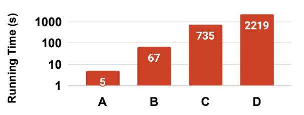
(c) Exact variant [C].



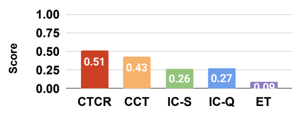
(d) CTCR, Jaccard, various δ [C].



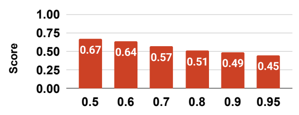
(e) Train/Test; Jaccard, $\delta = 0.8$ [D].



(f) Scalability [Various].



(g) PR, $\delta = 0.8$ [E].



(h) CTCR, PR, various δ [E].

Figure 8: Experimental results. The studied dataset is mentioned in the square brackets.

i-th input set and 0 otherwise. It then employs an agglomerative clustering algorithm over these vectors, to create the tree.

- **ET** - The existing company tree (created manually).

Evaluation Methods. We performed four complementary types of experiments to evaluate our algorithms. First, we compared the scores of the generated trees by all five algorithms, over inputs taken from all five datasets, for each of the six variants: Jaccard (both threshold and cutoff), F_1 (both threshold and cutoff), Perfect-Recall, and Exact variants, using a wide range of threshold parameters (except for the Exact variant, where $\delta = 1$). Specifically, we considered threshold values in the range $[0.5, 1]$, in incremental steps of 0.01. For the Perfect-Recall variant, where, as explained in Section 2, lower precision may be required in practice, we examined results for threshold values in the more extensive range of $[0.1, 1]$.

Second, to evaluate the robustness of the algorithms, we adopted a prevalent evaluation method in machine learning, where we randomly partition the largest dataset, D, into two query sets of equal cardinality: a training set and a test set. The tree was constructed over the training set and evaluated w.r.t. the test set.

Third, we evaluated the scalability of *CTCR*, the best-performing algorithm, demonstrating its efficient parallel implementation.

Finally, we also conducted a user-study with XYZ taxonomists.

5.3 Score-based Evaluation Results

Score comparisons. To normalize the scores into the range $[0, 1]$ we divided the actual score by the sum of the weights of all the sets in the input, which is a loose upper bound on the maximum possible score. As the optimal score cannot even be approximated (Section 2), over some inputs, a score of 0.001 may be optimal, whereas, over other inputs, a score of 0.5 may be trivial. The score, thus, offers an objective comparison only between different algorithms over the same input and problem variant.

In all examined variants and inputs, *CTCR* outperformed all its competitors, with *CCT* being the second-best algorithm. Moreover, the score gap between *CTCR* and *CCT* is typically significant (roughly 10% on average). This trend is demonstrated in Figures 8a, 8b, 8c and 8e. Specifically, figures 8a-8c depict representative results over the C dataset for the threshold Jaccard, Perfect-Recall, and Exact variants, respectively, whereas results over the E dataset for the Perfect-Recall variant are depicted in Figure 8e.

Importantly, the scores of *CTCR* for the Exact variant, including the score in Figure 8c, are all optimal. By using the *MIS* algorithm from [22], we were able to solve all Exact *OCT* instances optimally. Consequently, the scores for the Exact variant exceeded the scores of the Perfect-Recall variant even for much lower threshold values in the range of $[0.7, 1)$. Thus, an important insight is that due to the improved performance of *CTCR* for the Exact variant, it is worth employing this specialized version, even when some similarity error can be tolerated, particularly in the Perfect-Recall variant.

In general, the score of *CTCR* never dropped below 0.5, which demonstrates that in practice, the problem can often be reasonably approximated (for a score of 0.5, the approximation ratio is in the range $[1, 2]$), despite the worst-case theoretical bounds. Due to space constraints, we omitted results for the F_1 variants and the cutoff Jaccard variant, which demonstrated similar trends. Moreover, the ranking of the algorithms, in terms of the score, is roughly the same as in the figures, across all examined datasets.

Figures 8g-8h demonstrate the scores achieved by *CTCR* across a wide range of threshold values for the threshold Jaccard variant over the C dataset and the Perfect-Recall variant over the E dataset, respectively. As expected, lowering the threshold consistently allows covering more input sets and achieving higher scores.

Train/Test evaluation. The evaluation results where we split the data into train/test sets are depicted in Figure 8e. We randomly partitioned the input 50 times, taking the average score over the test set. The scores are predictably lower than the analogous scores in Figure 8a, however, *CTCR* still achieved the best performance.

Scalability. We performed the scalability tests over the four XYZ datasets, whose sizes range from 450 queries (28K items) to 20,000 queries (1.2M items). Note that *CTCR* is highly parallelizable. First, it computes all the 2-conflicts (as described in Section 3) in parallel. It also computes in parallel the cover scores for each category, at any given point in the item assignment phase. Figure 8f depicts the running times of *CTCR*. For the smallest dataset it takes 5 seconds to generate the results, while for the largest dataset it takes about 37 minutes - a reasonable time for an offline process. Manual construction by taxonomists typically takes weeks.

5.4 User Study

As mentioned in the introduction, our solutions have been evaluated by XYZ taxonomists, as a tool that allows reducing manual work, and reflect user demand as indicated by large rapidly-changing datasets. Specifically, this user study was performed over two months by three experienced in-house full-time taxonomists, employed by XYZ. All three taxonomists are experts in the Electronics and Fashion domains. The user study was performed over the XYZ datasets to maximally leverage their expertise, as they work regularly on categorizing this exact data.

Manual effort. Before detailing the evaluation, we note that the main reported advantage of our approach was time-saving, as

Table 1: The contribution of covering each subset of the input sets (result sets to queries in D vs. existing categories), in the threshold Jaccard variant with $\delta = 0.8$, to the final score of the *CTCR* tree, per the ratio of the total weights of each subset.

Queries/Existing	% of Score from Queries	% of Score from Existing
90%/10%	93.14%	6.86%
70%/30%	68.22%	31.78%
50%/50%	48.18%	51.82%
30%/70%	29.55%	70.45%
10%/90%	7.13%	92.87%

almost all necessary manual interventions are also required in the fully-manual approach. The scalability results above show that the execution time of our solution is negligible compared to manual construction. Moreover, taxonomists already maintain the tree by correcting item assignments and assigning new items (automatically by using, e.g., [10]). They also constantly monitor the tree to assess how well the latest query trends are addressed. Therefore, the tuning process is almost entirely subsumed by the taxonomists' daily manual work. Moreover, as the findings below show, it also simplifies complementary manual tasks, and provides various improvements in the categorization itself.

In the first part of the study, all algorithms were compared w.r.t. qualitative metrics. The second part of the study was dedicated to an extensive evaluation of the best-performing algorithm, *CTCR*.

Comparing all algorithms. The taxonomists were asked to compare the outputs of the above four automated algorithms, based on the following criteria. (1) Categorization structure: which structure makes the most sense and helps identify more problematic categorization decisions in the existing company tree. (2) Item assignment: which assignment makes the most sense. (3) Overall assessment: which tree is closest to a finished product and presumed to save taxonomists the most time. In terms of all the above criteria, the impressions of the taxonomists were highly correlated with the actual tree scores, and *CTCR* was viewed as by far the leading algorithm, followed by *CCT*.

Evaluation the *CTCR*-based human-in-the-loop workflow. In the second part of the study the taxonomists examined the real-world usage of *CTCR* over 6 weeks. As our solution takes the human-in-the-loop approach by design, we first report the findings, in terms of the complementary manual tasks discussed in Section 2.3: (1) ensuring conservative updates, (2) ensuring navigability, cohesiveness and labeling categories, and (3) tuning the solution. We conclude with examples of categorization errors and improvements.

Ensuring conservative updates. To test whether the tree can reflect user queries without radically changing the categorization, taxonomists added as input sets the existing tree categories, weighted uniformly. Table 1 shows that the weight ratio between query result sets and existing categories (modulated by adjusting the query weights) is roughly translated into the same ratio between the contribution of covering each of the two subsets to the total score (computed over the D dataset and the existing categories, for the threshold Jaccard variant with $\delta = 0.8$). This indicates that modulating the weights is sufficient to exert control over the extent to which the existing tree may change. This process of setting weights to control the extent of the changes was reported as significantly quicker than manual tuning (hours vs. days).

Ensuring cohesiveness, navigation and correct labeling. All taxonomists reported that assigning labels to categories of the CTCR-based tree was straightforward, as each category was marked by the search query or existing category label of the input sets it matches. After two taxonomists had assigned category names and added high-level and intermediate categories to reach a comparable navigation structure to the existing tree, the third taxonomist reported that in terms of the assessment of navigation and conceptual cohesiveness of categories, as seen in user studies in [11, 35], there were no noticeable quality differences. This equivalence in cohesiveness is confirmed by computing (over the same trees as Table 1) the average pairwise tf-idf similarity within each category, w.r.t. to the product titles. When uniformly averaging across all categories, the scores were 0.52 for the *CTCR*-based tree and 0.49 for the Existing tree. When weighting the average by category sizes, the both scores were 0.45.

Fine-tuning the preprocessing and algorithm parameters. After extensive experimentation, the favored reported setting was using the Jaccard threshold variant with $\delta = 0.8$. However, as *CTCR* can be reemployed independently over selected subtrees, for some subtrees with diverse categories where users are presented with an additional filtering interface the Perfect-Recall variant with $\delta = 0.6$ was preferred (for reasons discussed in Section 2.2).

Note, however, that the solution is generally robust to small changes in the threshold parameter. As seen in Figure 8d, the change to the score is relatively minor when using any threshold in the range $[0.6 - 0.9]$. Hence, tuning the threshold parameter was reported to be straightforward.

Ablation tests indicated that all preprocessing steps were significant, as removing any step added noise in the form of many small errors (assessed by taxonomists).

Identifying and correcting errors. In spite of the preprocessing steps, some misclassifications made by the search engine still propagated into the input sets. While to a large extent this is mitigated by our preprocessing procedure, it cannot be resolved entirely These rare occurrences do not pose novel challenges, but are rather addressed by dedicated tools used for manually-constructed categories as well. For example, a Nike blazer is a popular shoe, unrelated to the blazer jacket. However, since Nike blazers were also included in the result set for the "blazers" query, the "Nike Blazer" category was created under "Blazers". Taxonomists routinely search for such suboptimal assignments via a tool that detects high pairwise distances between embeddings of items within a category. In the above example, the problem detected in the "Blazers" category led to manually repositioning the "Nike Blazer" category. When only a few items are erroneously assigned to a new category, the automatic tool [10] reassigns these to an existing category.

Another issue relates to important categories being underrepresented in the search queries. For example, as enough time passes after the World Cup, related merchandise is searched less frequently. Some of these categories are of collectible items and were deemed necessary by taxonomists, that easily detected their absence when presented with the list of non-covered categories from the existing tree. This was resolved by reemploying the algorithm over the relevant subtree after increasing the weights of the corresponding input sets and lowering the threshold for queries where it was not essential to include all relevant items in the category.

Similar automatic solutions are also applied for underrepresented items that appear in very few queries. In general, a query containing many items that do not appear in other queries has a higher chance of being covered. However, if no query containing a rare item is covered, it is initially absent from any covering category. These are automatically detected and assigned to existing categories. The only exception is when many non-covered items appear in the same query. This indicates a need for a separate category, and hence the threshold of the query is reduced when reemploying the algorithm.

All taxonomists reported that reemploying *CTCR* several times is sufficient to derive a tree with the desired categorization improvements, such that the number of (mostly automatic) daily fixes is not higher than what is needed for manually constructed categories.

Categorization improvement. In terms of the contribution of *CTCR* to the low-level categorization, a reported advantage was identifying, via search queries, errors in matching user demand. One such insight is the example of memory cards provided in the introduction. Another example relates to the tragic death of the basketball player, Kobe Bryant, which led to a significant spike during February 2020 in the demand for memorabilia related to him. *CTCR* identified that there should have been a dedicated "Kobe" subtree to better serve the many users that sought these products.

To conclude, these evaluations indicate that *CTCR* qualitatively outperforms other automated algorithms, allows modulating ratios between covering various input sources, and offers improvements in the categorization while saving effort and time.

6 RELATED WORK

The construction and maintenance of hierarchical categorizations have been the focus of research in e-commerce, document management, and question answering [11, 16, 32, 35], with many effective automated solutions proposed in recent years [19, 28, 30, 36]. The vast majority of these works, however, categorize terms and concepts based on semantic and lexical relations, often derived from a given corpus of documents [36]. Concretely, in *term taxonomies* [23], each node represents a term or a phrase, and directed edges connect more general concepts to related more specific terms, whereas in *topic taxonomies* [19], each node is associated with a set of terms that characterize the topic. Common solutions include using lexical patterns to extract hyponymy ("is-a") relations [25], crowd-sourcing [31], training a classifier over term relations in a supervised manner [24], and using a hierarchical clustering algorithm to cluster word representations based on similarity [36], which is also a common method for document classification [26].

Other than the different domains of applicability and input types, there are several important distinctions in our setting. A first fundamental distinction is the different combinatorial settings in many of these works, e.g., categorizations often having a more general DAG-based structure [14, 30], or term taxonomies assigning only a single entity per node [34]. Second, as explained in the introduction, in the context of an e-commerce product tree, there are important considerations beyond the lexical and semantic relations that determine which potential category is a sub-type of another and which items match it. Instead, in our setting, the dominant sets are application-dependent and can be determined by domain experts, derived from result sets of query logs, or any relevant means. Our modeling of a computational problem w.r.t. an input of candidate categories is a novel approach, tailored for e-commerce deployment. Namely, taxonomists can manually and algorithmically curate the sources and contents of input sets, and adjust weight and threshold parameters for each set. Correspondingly, the algorithmic problem we model is not comparable to other works, and we are not aware of any categorization research that provides a formal model and studies the approximation complexity of the corresponding algorithms. Interestingly, while some works compute analogous similarity scores of the solution to a labeled ground-truth [11, 26, 27], this metric is only used for evaluation and does not guide the construction. In general, the utility of a taxonomy is to a large extent subjective, and many works evaluate the solution via a user study [33, 36], which is also part of our experimental evaluation.

Closest to ours is a work that also targeted e-commerce [18]. It improved the low-level categorization by adding one additional layer below existing leaf categories. The distinction to our work is captured by the discussion above, as they also used the common approach in topic taxonomies of embedding products based on titles and clustering the embeddings. Nevertheless, to compare approaches, we have adapted this algorithm to our setting (baseline IC-S in Section 5) and demonstrated experimentally that our algorithms produce trees with better scores and qualitative evaluations. A related line of research studies the classification of a new item to a category of a given tree [10, 21, 29], an objective complementary to ours. We also note the work of [37], which generates a taxonomy of products as part of an algorithm for personalized recommendations. They learn the taxonomy from product descriptions and evaluate it based on the improvement to the recommendation model.

Lastly, recall that we have proposed two algorithms. The *CTCR* algorithm of Section 3 leverages reductions to the Maximum Independent Set problem, which, to our knowledge, is a novel method. The *CCT* algorithm of Section 4 clusters the sets to derive the tree structure. Compared to existing clustering solutions [18, 32, 36] it incorporates higher-order item relations into the input. Namely, it clusters item sets, instead of the items. Moreover, to accentuate the "global perspective", the embeddings used by *CCT* allow comparing input set pairs not only in terms of their direct similarity but also in terms of their similarities to all other sets.

7 CONCLUSION

In this work, we have put forth a formal model for capturing the problem of constructing a category tree, tailored specifically to the e-commerce setting, and provided two algorithms that on real-world data achieve high scores, far above worst-case bounds. In particular, our best-performing algorithm, *CTCR*, which is based on a novel method of harnessing solvers and algorithms for the Maximum Independent Set problem to resolve categorization conflicts, optimally solved all instances of one problem variant. A user study among taxonomists also indicated the qualitative advantages of *CTCR* over solutions from related settings, and verified the efficacy of leveraging query logs to capture user demand and trends.

An interesting direction for future research is exploring whether in other categorization contexts one can also derive an extensive set of candidate categories, and adapt our model and algorithms to the specification of the application at hand.

REFERENCES

[1] https://export.ebay.com/en/start-sell/selling-basics/seller-fees/fees-optional-listing-upgrades/.

[2] Ctcr implementation. https://github.com/shayg1/CategoryTrees.

[3] Crowdflower search relevance. https://data.world/crowdflower/ecommerce-search-relevance, 2015.

[4] Home depot product search relevance. https://www.kaggle.com/c/home-depot-product-search-relevance/data, 2016.

[5] Innerwear data from victoria's secret and others. https://www.kaggle.com/PromptCloudHQ/innerwear-data-from-victorias-secret-and-others, 2017.

[6] Elasticsearch. https://www.elastic.co/elasticsearch, 2020.

[7] Geir Agnarsson, Magnús M Halldórsson, and Elena Losievskaja. Sdp-based algorithms for maximum independent set problems on hypergraphs. *Theoretical Computer Science*, 470:1–9, 2013.

[8] Uri Avron, Shay Gershtein, Ido Guy, Tova Milo, and Slava Novgorodov. ConCaT: Construction of Category Trees from Search Queries in E-Commerce. In *ICDE*, 2021.

[9] Slobodan Beliga, Ana Meštrović, and Sanda Martinčić-Ipšić. An overview of graph-based keyword extraction methods and approaches. *JIOS*, 39(1):1–20, 2015.

[10] Ali Cevahir and Koji Murakami. Large-scale multi-class and hierarchical product categorization for an e-commerce giant. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, pages 525–535, 2016.

[11] Shui-Lung Chuang and Lee-Feng Chien. A practical web-based approach to generating topic hierarchy for text segments. In *CIKM*, page 127–136, 2004.

[12] Eyal Dushkin, Shay Gershtein, Tova Milo, and Slava Novgorodov. Query driven data labeling with experts: Why pay twice? In *EDBT*, 2019.

[13] Shay Gershtein, Uri Avron, Ido Guy, Tova Milo, and Slava Novgorodov. On the hardness of category tree construction. In *ICDT*, pages 4:1–4:17, 2022.

[14] Amit Gupta, Rémi Lebret, Hamza Harkous, and Karl Aberer. Taxonomy induction using hypernym subsequences. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 1329–1338, 2017.

[15] Magnús M Halldórsson and Elena Losievskaja. Independent sets in bounded-degree hypergraphs. *Discrete applied mathematics*, 157(8):1773–1786, 2009.

[16] Idan Hasson, Slava Novgorodov, Gilad Fuchs, and Yoni Acriche. Category recognition in e-commerce using sequence-to-sequence hierarchical classification. In *WSDM*, 2021.

[17] Ruining He and Julian McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *Proc. of WWW*, pages 507–517, 2016.

[18] Y. Hsieh, S. Wu, L. Chen, and P. Yang. Constructing hierarchical product categories for e-commerce by word embedding and clustering. In *IRI*, 2017.

[19] Jiaxin Huang, Yiqing Xie, Yu Meng, Yunyi Zhang, and Jiawei Han. Corel: Seed-guided topical taxonomy construction by concept learning and relation transferring. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1928–1936, 2020.

[20] Hua Jiang, Chu-Min Li, and Felip Manya. An exact algorithm for the maximum weight clique problem in large graphs. In *AAAI*, pages 830–838, 2017.

[21] Zornitsa Kozareva. Everyone likes shopping! multi-class product categorization for e-commerce. In *NAACL*, pages 1329–1333, 2015.

[22] Sebastian Lamm, Christian Schulz, Darren Strash, Robert Williger, and Huashuo Zhang. Exactly solving the maximum weight independent set problem on large real-world graphs. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 144–158. SIAM, 2019.

[23] Xueqing Liu, Yangqiu Song, Shixia Liu, and Haixun Wang. Automatic taxonomy construction from keywords. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1433–1441, 2012.

[24] Maximilian Nickel and Douwe Kiela. Poincar\'e embeddings for learning hierarchical representations. *arXiv preprint arXiv:1705.08039*, 2017.

[25] Alexander Panchenko, Stefano Faralli, Eugen Ruppert, Steffen Remus, Hubert Naets, Cédrick Fairon, Simone Paolo Ponzetto, and Chris Biemann. Taxi at semeval-2016 task 13: a taxonomy induction method based on lexico-syntactic patterns, substrings and focused crawling. In *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016)*, pages 1320–1327, 2016.

[26] Kunal Punera, Suju Rajan, and Joydeep Ghosh. Automatically learning document taxonomies for hierarchical classification. In *Proc. of WWW*, 2005.

[27] Cécile Robin, James O'Neill, and Paul Buitelaar. Automatic taxonomy generation: A use-case in the legal domain. In *Language and Technology Conference*, pages 318–328. Springer, 2017.

[28] Jingbo Shang, Xinyang Zhang, Liyuan Liu, Sha Li, and Jiawei Han. Nettaxo: Automated topic taxonomy construction from text-rich network. In *Proceedings of The Web Conference 2020*, page 1908–1919, 2020.

[29] Dan Shen, Jean-David Ruvini, and Badrul Sarwar. Large-scale item categorization for e-commerce. pages 595–604, 10 2012.

[30] Jiaming Shen, Zhihong Shen, Chenyan Xiong, Chi Wang, Kuansan Wang, and Jiawei Han. Taxoexpan: self-supervised taxonomy expansion with position-enhanced graph neural network. In *Proceedings of The Web Conference 2020*, pages 486–497, 2020.

[31] Yuyin Sun, Adish Singla, Dieter Fox, and Andreas Krause. Building hierarchies of concepts via crowdsourcing, 2015.

[32] Lei Tang, Jianping Zhang, and Huan Liu. Acclimatizing taxonomic semantics for hierarchical content classification. volume 2006, pages 384–393, 01 2006.

[33] Chi Wang, Marina Danilevsky, Nihit Desai, Yinan Zhang, Phuong Nguyen, Thrivikrama Taula, and Jiawei Han. A phrase mining framework for recursive construction of a topical hierarchy. In *KDD*, pages 437–445, 2013.

[34] Yue Yu, Yinghao Li, Jiaming Shen, Hao Feng, Jimeng Sun, and Chao Zhang. Steam: Self-supervised taxonomy expansion with mini-paths. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1026–1035, 2020.

[35] Quan Yuan, Gao Cong, Aixin Sun, Chin-Yew Lin, and Nadia Magnenat Thalmann. Category hierarchy maintenance: a data-driven approach. In *SIGIR*, 2012.

[36] Chao Zhang, Fangbo Tao, Xiusi Chen, Jiaming Shen, Meng Jiang, Brian Sadler, Michelle Vanni, and Jiawei Han. Taxogen: Unsupervised topic taxonomy construction by adaptive term embedding and clustering. In *KDD*, page 2701–2709, 2018.

[37] Yuchen Zhang, Amr Ahmed, Vanja Josifovski, and Alexander Smola. Taxonomy discovery for personalized recommendation. In *WSDM*, pages 243–252, 2014.