Improving Constrained Search Results By Data Melioration

Ido Guy
eBay Research
idoguy@acm.org

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

Slava Novgorodov
eBay Research
snovgorodov@ebay.com

Brit Youngmann
Tel Aviv University
brity@mail.tau.ac.il

***Abstract*—The problem of finding an item-set of maximal aggregated utility that satisfies a set of constraints is at the cornerstone of many search applications. Its classical definition assumes that all the information needed to verify the constraints is explicitly given. However, in real-world databases, the data available on items is often partial. Hence, adequately answering constrained search queries requires the completion of this missing information. A common approach to complete missing data is to employ Machine Learning (ML)-based inference. However, such methods are naturally error-prone. More accurate data can be obtained by asking humans to complete missing information. But, as the number of items in the repository is vast, limiting human effort is crucial. To this end, we introduce the Probabilistic Constrained Search (PCS) problem, which identifies a bounded-size item-set whose data completion is likely to be highly beneficial, as these items are expected to belong to the result set of the constrained search queries in question. We prove PCS to be hard to approximate, and consequently propose a best-effort PTIME heuristic to solve it. We demonstrate the effectiveness and efficiency of our algorithm over real-world datasets and scenarios, showing that our algorithm significantly improves the result sets of constrained search queries, in terms of both utility and constraints satisfaction probability.**

I. Introduction

The selection of a k-size item-set with the maximal aggregated utility that satisfies a set of constraints is a fundamental problem in many search applications, such as e-commerce platforms and search engines. As an example, consider a user searching an e-commerce website for shirts. Rather than simply returning the top-k items matching the user's request (according to the utility scores), the platform often takes into account additional factors. For instance, it might have signed a contract with a particular brand, requiring that the query's result set contains at least one item of this brand. It may also wish to diversify the result set, by including at least two other brands, or shirts having different sleeve lengths [1], [2], [3].

The problem of finding an item-set of maximal aggregated utility that satisfies a set of constraints is often referred to as *Constrained Search* (CS) [2]. Its classical definition in the literature assumes that all the information needed to verify the constraints is explicitly provided. In practice, however, the data available for the items is often partial. For example, in e-commerce platforms, sellers frequently upload goods in batches and tend to focus, in their provided information, only on the most important attributes (e.g., product name and price), with additional information provided by text descriptions and images. Hence, adequately answering constrained search queries requires the completion of missing information.

A common approach to address this problem is the completion of missing data by Machine Learning (ML) algorithms [4], [5]. However, ML is naturally error-prone. Previously-reported results indicate that it is hard to attain 85% precision for a reasonable recall [6]. More accurate data can be obtained by asking humans to complete missing data. Another solution is asking humans to complete all missing data. But, as the number of items in the database is typically huge, limiting human effort is crucial. To this end, we propose a hybrid approach that, given a bound on the number of items for which data completion can be requested, harnesses the data derived by common ML modules (e.g., attribute extraction [7], [5]) to focus on the most "beneficial" items for which manual data completion should be performed.

Specifically, given a set of constrained search queries of interest (e.g., the most frequently asked queries) and a bound k on the number of requests from humans, we consider (in the offline phase) the probabilities derived by ML modules, to identify a k-size item-set I that is expected to contribute the most to these queries (in terms of constraint satisfaction and utility). This is achieved by employing a dedicated algorithm that we experimentally show to be effective, despite the inherent complexity of the corresponding optimization problem. The underlying platform then generates data completion tasks (referred to as data requests) for each item $i \in I$. Each task is assigned to a human(s) and requires to complete all missing data for an item. The system then updates the repository appropriately. Remaining missing values are completed using ML modules. We refer to this phase of partially cleaning the repository as the *data melioration* process. Next, at run time, as before, queries are evaluated over the completed database.

Before presenting our contributions, let us illustrate through a simple example the problem that we address in this work, named the *Probabilistic Constrained Search* (PCS) problem.

Example 1.1: Consider an e-commerce platform selling, among others, women shirts. Part of the database is depicted in Figure 1. It includes information about the shirts' brand names and sleeve lengths. Here, the sellers provided some of the attribute values, and the missing ones were predicted using ML algorithms. Next to each given/predicted value, we also show (in parenthesis) the probabilities of all possible alternative values, as determined by the ML modules. W.l.o.g. assume that the platform selects the value with the highest probability as the inferred attribute value. This may, or may not, match the actual ground truth value (marked in red). We examine two search queries: "women shirts" (q_1), and "women sport shirts" (q_2). Let k, the number of items to be returned, to be 3. The

utility scores of the items w.r.t. the queries are also depicted in Figure 1. Assume that the following constraints are imposed on both queries: During a transition season, the platform requires the queries' results to include items having different sleeve lengths. Also, the platform has signed a contract with VERSACE, requiring all result sets to include at least one VERSACE item. To ensure coverage of the brands, another constraint requires to include items spanning different brands.

The platform's result for q_1 is S_1=$\{i_1, i_7, i_8\}$ and its result for q_2 is S_2=$\{i_4, i_5, i_8\}$ (with utility of 1.8 for both sets). Note, however, that S_1 does not satisfy the first constraint w.r.t. the ground truth, and the result for q_2 could be improved in terms of utility, if the ground truth was known. Completing missing data on item i_7 would improve both results: for q_1 the result becomes S'_1=$\{i_1, i_2, i_7\}$ (with utility of 2.6), and for q_2 the result changes to S'_2=$\{i_4, i_5, i_7\}$ (with utility of 2.7). Moreover, both S'_1 and S'_2 also satisfy the constraints.Completing missing data on item i_2, on the other hand, would improve the result only for q_1, altering it to S'_1. In contrast, completing missing values on items i_8 or i_9 is redundant. Item i_8 has low utility score w.r.t. both queries and item i_9 have alternative items with higher probabilities, which can meet the constraints' requirements (item i_7 is more likely to have the brand VERSACE). Leveraging the probabilities derived by the ML modules, our algorithm chooses a bounded-size item-set that is expected to improve the result quality (combined utility and constraints satisfaction probability) for both queries. For example, assume that the bound of data requests is 3. According to our problem formulation, the platform should complete missing data on items i_2, i_6, and i_7, which would yield here the optimal (w.r.t. the ground truth) solutions for both queries: S'_1 and S'_2.

Combining ML and human work is a common approach in numerous domains (e.g., entity resolution [8]), and much research has been devoted to minimize human efforts [5]. A common usage of a human effort for ML is harnessing domain experts to generate labeled data for supervised learning [5], [4]. Our work complements this previous research by leveraging the probabilities derived by ML algorithms, to identify which data records are best to manually complete. Here as well, we aim to effectively use human effort, by identifying, given a bounded budget, the most beneficial items to be completed for the particular objective of CS. See a detailed comparison to related work in Section VII.

We next outline our main contributions.

Problem Formulation: One can improve the result of a constrained search query q in two manners: increase the overall utility or satisfy the constraints with higher probability (possibly at the cost of utility). Intuitively, an optimal result for q is a k-size item-set that is of *best quality, in terms of its combined utility and constraints satisfaction probability scores*. This set may be different than the one currently returned by the platform (whose answer is determined by the provided/predicted values), which may have a lower quality than such an optimal set. Therefore, we would like to ensure that the information on the items in the optimal set is correct so that it will be possible to include them in q's answer. We, therefore, define the PCS problem for a single query, as the problem of finding a k-size items set of maximal quality, i.e., with the highest, combined, utility score and constraint satisfaction probability. (Section III).

Complexity Analysis: To get some intuition about the hardness of PCS, we also discuss the hardness of the classical Constrained Search (CS) problem - a restricted variant of PCS, where all probabilities are in $\{0, 1\}$. While in the simple setting where all constraints are defined over a single attribute, an optimal PTIME algorithm for CS exists [1], we show PCS to be *NP*-hard, even for this restricted case. For the general case, we show that CS (and therefore also PCS) is hard to approximate to a constant factor in PTIME. We furthermore prove this bound to hold for PCS, even if the solution for the corresponding CS is given (Section III).

Algorithms: Since PCS is hard to approximate, we provide an efficient best-effort algorithm, which we experimentally show to be effective. Two common classes of algorithms for top-k problems are the greedy [9] and interchange [10] algorithms. Our algorithm follows an analogous hybrid line, applying a greedy procedure followed by a local-search one. **Greedy initialization**. This procedure finds an item-set that is likely to satisfy the constraints by iterative processing of the constraints. Each iteration selects new items to satisfy the currently-examined constraint while also considering items selected so far. The main challenge is to ensure that the extended item-set still satisfies previously-examined constraints, while also satisfying the current constraint with the highest possible probability. **Improvement via local search**. This procedure iteratively moves to neighbor solutions by replacing some item(s) with different item(s) having higher utility scores. The main challenge is to devise sufficient conditions on the new item(s) to be added, ensuring improvement. Namely, to assure that if the constraints satisfaction probability decreases, the utility increases sufficiently to compensate for it (Section IV).

Extensions: To handle multiple queries simultaneously, we extend PCS to support two commonly-used aggregation strategies: Average and Least Misery (LM), resp., presenting the AVG-PCS and LM-PCS problems. In AVG-PCS, the goal is to find an item-set that is expected to maximize the average contribution to the queries. In LM-PCS, the goal is to find an item-set that maximizes the minimum contribution for each of the queries. We show how our algorithm can be generalized to these variants(Section V).

Experimental Study: We conducted an experimental study based on real-world datasets from both the e-commerce and people search domains. Our results indicate the effectiveness of each of step of our algorithm, as well as the effectiveness of the whole algorithm compared with multiple alternative baselines. The experiments show that, on average, our algorithm improves the quality of result sets of constrained search queries in 4% in terms of utility, and in 27% in terms of constraints satisfaction probability (Section VI).

A demonstration of our framework was recently presented in [11]. The short paper accompanying the demonstration provides only a brief description of the framework, whereas the present paper provides the theoretical foundations and algorithms underlying the framework.

II. PRELIMINARIES

We present the model of data and constraints underlying our study, used to formally define the PCS problem.

Item	Utility Score	
	"women shirts" (q_1)	"women sport shirts" (q_2)
i_1	0.9	0.001
i_2	0.8	0.001
i_3	0.7	0.001
i_4	0.001	0.9
i_5	0.001	0.9
i_6	0.001	0.9
i_7	0.9	0.9
i_8	0.001	0.001
i_9	0.6	0.8
i_{10}	0.4	0.001

Fig. 1: Example database which include information about items' brand name (Brn), where the values are Gucci (g), Versace (v) and Nike (n), and sleeve lengths (Slv), where the values are long (l) and short (s). The actual values are marked in red. On the bottom are the items' utility scores w.r.t. two search queries.

A. Data Model

A data repository consists of a set $\mathcal{I}=\{i_1,\ldots,i_n\}$ of n items, each associated with a set $\mathcal{A}$ of attributes. Every attribute $a\in\mathcal{A}$ has a domain of values, denoted as V_a. For each item $i\in\mathcal{I}$, attribute values provided by its owner have a probability of 1 (with all other values having the probability of 0). The value probabilities for attributes not specified by the owner are obtained by employing an ML module [7], [4], [5]. Let $Pr_{a,v}[i]$ denote the probability item $i\in\mathcal{I}$ has the value of $v\in V_a$ for the attribute $a\in\mathcal{A}$. We assume that the probabilities of the attribute values, of the same or different items, are independent. Figure 1 depicts information about 10 items. Here the attributes are the items' brand name and sleeve length. The sellers provided some of the values. Missing values were predicted using ML algorithms. Next to each given/predicted value, are also depicted (in the parenthesis) the alternative values, along with their probabilities (as determined by the ML module). W.l.o.g. we assume that the platform chooses the value with the highest probability as the attribute value. This may, or may not match the actual ground truth value (colored red).

Given a search query q, each item $i\in\mathcal{I}$ is associated with a utility score, denoted as $u_q(i)\in[0,1]$, which measures the expected welfare or satisfaction of a consumer w.r.t q. Following [1], we define the overall utility score of an item-set $I\subseteq\mathcal{I}$ w.r.t. a query q, denoted as $u_q(I)$, as the sum of utility scores of each individual item.

B. Count And Coverage Constraints

A constraint is defined over an attribute and a k-size item-set. Here we consider two simple types of constraints, which can capture a wide range of previously-studied constraints [1], [2], [3]: *count* and *coverage* constraints.

Count Constraint. A count constraint defines upper and lower bounds on the number of *set members having a specific value for a given attribute*. Formally, a count constraint defined over an attribute $a\in\mathcal{A}$ and a value $v\in V_a$, requires the item-set to contain at least $floor$ and no more than $ceil$ items having the value of v for a. A count constraint is therefore denoted as a tuple $\langle a,v,floor,ceil\rangle$. Given a k-size item-set I and a count constraint $c=\langle a,v,floor,ceil\rangle$, the probability I satisfies c is denoted as $Pr_c[I]$. One can compute this probability by considering all subsets $I'\subseteq I$ of I of size $floor\leq j\leq ceil$, examining the probability that all items in the subset I' have the value of v for the attribute a, and all items in $I\backslash I'$ do not. Formally: $Pr_c[I] := \sum_{j=floor}^{ceil}\sum_{I'\subseteq I,|I'|=j}\prod_{i\in I'}Pr_{a,v}[i]\prod_{i\notin I'}(1-Pr_{a,v}[i])$.

Coverage Constraint. A coverage constraint defines upper and lower bounds on the number of *set members having different values for a given attribute*. Formally, a coverage constraint defined over an attribute a, requires the item-set to contain at least $floor$ and no more than $ceil$ different values for a. A coverage constraint is therefore denoted as a tuple $\langle a,floor,ceil\rangle$. Given a k-size item-set I and a coverage constraint $c=\langle a,floor,ceil\rangle$, the probability I satisfies c is denoted as $Pr_c[I]$. One can compute this probability using the Poisson Multinomial Distribution (PMD). Given a number $floor\leq j\leq ceil$, (j,k)-PMD expresses the probability that a k-size item-set I will have exactly j different values for a, where the items have different probabilities for having each value $v\in V_a$. In our setting, one needs to quantify the probability that a k-size item-set will have at least $floor$ and no more than $ceil$ different values. We therefore sum the (j,k)-PMD probabilities for all $floor\leq j\leq ceil$, as follows: $Pr_c[I]=\sum_{j=floor}^{ceil}(j,k)\text{-PMD}[I]$.

For both definitions, if no lower (resp., upper) bound is specified, we set $floor$ (resp., $ceil$) to 1 (resp., k). The user may also provide relative bounds (w.r.t the set size) for $floor$ and $ceil$, and the absolute numbers can be derived from them.

Given a k-size item-set I and a set of (count and coverage) constraints C, recall that $Pr_C[I]$ denote the probability I satisfies all constraint in C. Let $\mathcal{P}_C=\{C_1,\ldots,C_l\}$ denote a partition of the constraints in C into l subsets of dependent constraints. Namely, for every $i\in[1,l]$, all constraint in C_i are dependent, however they are independent of all constraints in C_j where $i\neq j$. We note that in the worst case, all constraints in C depend on one another. The probability I satisfies all constraints in given subset $C_j\subseteq C$ is defined, using the chain-rule, as follows: $Pr_{C_j}[I]:=\prod_{i=1}^{|C_j|}Pr[X_{c_i}|\cap_{j=1}^{i}X_{c_j}]$, where X_{c_i} is the event where I satisfies constraint c_i. The probability I satisfies all constraint in C is then: $Pr_C[I]:=\prod_{C_i\in\mathcal{P}_C}Pr_{C_i}[I]$.

This raises the question of finding the partition $\mathcal{P}_C$ for a given set of constraint C. Naturally, all constraints imposed on dependent (or the same) attributes are dependent as well. Namely, this problem reduces to the problem of finding a partition of the attributes into disjoint subsets of dependent attributes. To verify if two categorical attributes (e.g., brand name, sleeves length) are independent, one can use the Chi-Squared test of independence. For numerical attributes (e.g., price), one can use ANOVA test.

In this work we assume that each constraint can be satisfied, and that all constraints can also be satisfied simultaneously. A discussion regarding the related constraint satisfaction problem is given in Section V-B.

C. Constraints Satisfaction Estimation

Given a k-size item-set I and a constraint c, the computation of the exact value of $Pr_c[I]$, as described above, is exponential in k and *ceil*, as both definitions require to examine all subsets of I of size *ceil*. However, if all probabilities were in $\{0,1\}$, computing $Pr_c[I]$ - which in this case is either 0 or 1 - can be done in $O(k)$. We get that for a constraint set C, if all probabilities were boolean, $Pr_C[I]$ can be computed in $O(k{\cdot}|C|)$ (see details in [12]).

To estimate $Pr_C[I]$ in the general case, we use the *possible worlds semantics* [13]. Recall that the items' probabilities are assumed to be independent. A possible world is an assignment of a random value for each item attribute, chosen w.r.t. the probability distribution of its values. In such a possible world, the chosen value has probability 1 whereas all other values have probability 0. To estimate $Pr_C[I]$ we generate m possible worlds. For each such boolean world, we compute the value of $Pr_C[I]$, then output the averaged result. Let $\widehat{Pr_C[I]}$ denote the estimated value of $Pr_C[I]$. For fixed $\epsilon, \delta{>}0$, we can derive m, the number of worlds to be generated, s.t. $|\widehat{Pr_C[I]}-Pr_C[I]|{\leq}\epsilon$ with probability $\geq 1-\delta$. We prove the following ([12]):

Proposition 2.1: For a k-size item-set I, a set of l constraints C, and fixed $\epsilon, \delta{>}0$, we have: $|\widehat{Pr_C[I]} - Pr_C[I]|{\leq}\epsilon$ with probability $\geq(1-\delta)$, where $\widehat{Pr_C[I]}$ is estimated using $O(\frac{1}{\delta\cdot\epsilon^2})$ possible worlds, and it can be computed in $O(l{\cdot}k^2)$.

We assume ϵ and δ are provided as part of the input and all our results take these requirements into account. For simplicity of presentation, we omit explicit discussions of these parameters in our analysis whenever possible.

III. Problem Formulation

We next formally define the PCS problem for a single search query, and discuss its computational complexity.

A. Problem Definition

As mentioned, the data available for the items is often partial, and hence adequately answering constrained search queries requires the completion of missing data. Given a set of constrained search queries of interest (e.g., the most frequently asked queries) and a bound k on the number of items for which data completion can be requested, the PCS problem identifies a k-size item-set that is expected to contribute the most to these queries (in terms of both constraint satisfaction and utility).

For a single search query q, we identify a k-size item set who has the highest potential to improve the result of q. As mentioned, one can improve the result of q by increasing the overall utility or satisfy the constraints with higher probability. Since one may come at the expense of the other, an optimal result for q is an item-set with highest, combined utility score and constraint satisfaction probability. Note that this set may be different than the one currently returned by the platform (whose answer is determined by the provided/predicted attribute values). We define PCS for a single query as follows.

*Definition 3.1 (*PCS*):* Given a number k, a search query q, and a set of constraints C, find a set $I{\subseteq}\mathcal{I}$ s.t: $I{=}argmax_{|I'|=k}u_q(I'){\cdot}Pr_C[I']$.

Similar to [14], our definition uses multiplication to combine the utility and probability scores. Other combinations, e.g., weighted sum, are also possible, and we leave their investigation to future work. Note that PCS is defined using the items' utility scores, which may change when the workers complete missing information. Meantime, these scores are assumed to be the best available approximation to the actual utility scores (which are based on the real attribute values).

System workflow. In the offline phase (i.e., the data melioration phase), we consider a search query q (or a set of queries), a set of constraints C, and a bound k on the number of items for which data completion can be requested. By solving PCS the system identifies a k-size item set I that has the highest potential to improve the result of q, while *considering the items' probabilities (obtained by ML modules)*. The system then generates *data completion tasks* (referred to as *data requests*) for each item $i{\in}I$. Each task is assigned to a human(s) and requires to complete all missing data for an item i (by examining, e.g., its description and image). The tasks can be performed using a dedicated UI, or exported to various external crowd platforms. The system then updates the repository $\mathcal{I}$ appropriately, according to the new data on items in I. Remaining missing values are completed using ML modules. Next, at run time, as before, queries are evaluated over the completed database.

Setting k. In our problem definition, we set the selected item-set size to be k – the number of items in a search query result set. In case the bound on the number of data completion requests, denoted as k', is smaller than k, we generate requests only for the of the top k' items in I with the highest utility scores. In case $k'{>}k$, we repeat the cleaning process $\lceil\frac{k'}{k}\rceil$ times ,as follows. In each iteration, we will: (1) Select (at most) k items to be manually cleaned. (2) Complete their data and update the repository. (3) Run our algorithm again over the *updated repository* to identify the next item-set.

To compute k', the bound on the number of items completion requests, we operate as follows. We assume a cost model where the price per item is fixed. It reflects the average expenditure, per item, for completing all the item's missing attributes with sufficiently high precision (following common practice in e-commerce systems [4], [15]). k' is computed by dividing the budget the system allocates for data melioration by this per-item average cost.

An alternative approach is to ask the sellers (or the data providers) of the items to complete missing data. However, providing information in a structured way is a demanding task and may create friction between the sellers and the platform. Therefore, here as well, it is desirable to identify the items for which it is most beneficial to approach to a seller for data completion, in terms of ability to satisfy consumer demands. Thus, our problem definition stays the same regardless of whether the completion requests are directed to the sellers (in which case the cost is measured by "irritation" units) or to crowd workers (in which case the cost in monetary).

Example 3.2: Recall that the result set of the query q_1 (based on the provided/derived values) is $S_1{=}\{i_1, i_7, i_8\}$, with $u_{q_1}(S_1){=}1.8$ and $Pr_C[S_1]{=}0.55$. According to the PCS problem, the ideal result set whose missing items data should be requested is $S'_1{=}\{i_1, i_2, i_7\}$ with $u_{q_i}(S_1){=}2.6$ and $Pr_C[S'_1]{=}0.44$.

Completing missing data on these items assists the platform to include them in the result set of q_1 (once the database is corrected, the result of q_1 becomes S_1'), improving it in terms of both utility and constraints satisfaction probability (recall that w.r.t the ground truth, S_1 does not satisfy c_1).

To get some intuition about the hardness of PCS and to connect it to previous studied problems, we consider a restricted variant of PCS, referred to as Constrained Search (CS). In CS all probabilities are either 0 or 1. This corresponds to a common scenario where practical systems use ML modules to *predict* missing values. W.l.o.g. assume that the system chooses the value with highest probability as the predicted value (i.e., the probability of the chosen values is 1, and non chosen ones of 0). Let $\hat{\mathcal{I}}$ denote the resulting repository. In CS, the goal is to find a k-size item-set in $\hat{\mathcal{I}}$ that satisfies all constraints in C with probability 1, and maximizes the overall utility.

B. Hardness Results

As mentioned, given a k-size item-set I and a set of constraints C, computing the exact value of $Pr_C[I]$ may be expensive. Our hardness results hold even if $Pr_C[I]$ could be provided by an oracle in $O(1)$. To establish the connection between the PCS and the CS problems, we also provide hardness results for CS.

The authors of [1] have studied a restricted variant of the CS problem, where the goal is to select a k-size item-set that maximized the overall utility, subject to only *count constraints* imposed on a *single attribute*. For this problem, they presented an optimal PTIME algorithm. However, as we show, already in this restricted setting, PCS is NP-hard.

Theorem 3.3: The PCS problem is NP-hard, even with only count constraints imposed on a single attribute.

Practical scenarios, however, may include constraints imposed over multiple attributes. We first provide hardness result for CS, then analyze the more general PCS problem.

Theorem 3.4: The CS problem cannot be approximated to a constant factor in PTIME (unless P=NP), even with only count constraints, and even if all utility scores are equal.

We prove Theorem 3.4 via a novel reduction from the classic Maximum Independent Set (MIS) problem. It follows that there is no PTIME algorithm that can solve general CS instances. Henceforth, commercial systems use heuristic algorithms to find a k-size item-set that satisfies all constraint, yet does not necessarily maximize utility. A prominent example is ElasticSearch [16], used by popular search applications (e.g., eBay, Facebook [16]). This bound naturally holds also for PCS. Moreover, we can show it to hold even if the solution to the analog CS problem is given.

Theorem 3.5: The PCS problem is NP-hard and cannot be approximated to a constant factor in PTIME (unless P=NP), even with only count constraints, and even if the solution to the corresponding CS problem is given.

IV. Algorithms

As mentioned, PCS is hard to approximate in PTIME. We, therefore, provide a heuristic PTIME algorithm, which we experimentally show to be effective. Our algorithm performs two steps. It first builds an initial solution by applying a greedy computation, with the goal of maximizing the constraints satisfaction probability. It then employs a local search to improve the solution in terms of utility. For simplicity of presentation, we assume that the bound on the number of data requests is equal to the number of items in a query result set. We also assume that there is at most one count (resp., coverage) constraint per attribute. Otherwise, we merge all count (resp., coverage) constraints imposed on the same attribute into a single constraint, by taking the maximal $floor$ and the minimal $ceil$ values of these constraints.

Algorithm 1: UpdateConstraint Procedure

input : A coverage constraint c=$\langle a, floor, ceil\rangle$, and a set of items S.
output: A coverage constraint $\widehat{c}$=$\langle a, \widehat{floor}, \widehat{ceil}\rangle$

```
vals ← ∅
foreach i ∈ S do
    v_i ← argmax_{v∈V_a}(Pr_{a,v}[i])
    vals ← vals ⋃ {v_i}
    foreach j ∈ I\S do
        Pr_{a,v_i}[j] ← 0
floor^ ← max(0, floor−|vals|), ceil^ ← max(0, ceil−|vals|)
ĉ = ⟨a, floor^, ceil^⟩
return ĉ
```

A. Greedy-Based Algorithm

Our greedy algorithm employs two procedures: one for handling count constraints and one for coverage constraints. We begin by describing these two procedures (their pseudo-code is given in [12]), then present the full greedy algorithm.

Handling count constraints. Given a count constraint c=$\langle a, v, floor, ceil\rangle$, this procedure selects the top $floor$ items, according to their probabilities of having the value v for the attribute a. Ties are being broken according to utility scores (and arbitrarily for items with the same probability and utility). Then, to ensure no more items having the value v are selected, for every item $i\in\mathcal{I}\backslash S$, we set $Pr_{a,v}[i]$=0.

Handling coverage constraints. Given a coverage constraint c=$\langle a, floor, ceil\rangle$ and the set of items selected so far S, this procedure first updates the bounds of c to take into consideration the items in S. This is done using the procedure UpdateConstraint (Algorithm 1), which works as follow. For every item $i\in S$, let v_i be the value i is most likely to have for the attribute a. To ensure no more items having the value of v_i for a are selected, for every item $j\in\mathcal{I}\backslash S$ we set $Pr_{a,v_i}[j]$=0. Let num denote the number of values for a the items in S have. We update the lower and upper bounds of c by setting $\widehat{floor}$=$max(0, floor-num)$, and $\widehat{ceil}$=$max(0, ceil-num)$. Next, we select the top $\widehat{floor}$ items according to their probabilities of having different values to a, as follows. We sort all items $|V_a|$ times, according to their probabilities of having each value $v\in V_a$ for a. We then select $\widehat{floor}$ items, where each selected item is the top item in a different list, and return these items.

The greedy algorithm iterates over the constraints. While count constraints require to select items having a specific value for a given attribute, coverage constraints require to select items with no particular values. We, thus, first iterate over the count constraints. With the remaining slots, we select items with the highest utility scores that have not been chosen yet. We assume a a random order imposed over the constraints.

Algorithm 2: Greedy Algorithm

input : A set of constraints C, a search query q, and a number k.
output: A k-size set of items.

```
1  S ← ∅
2  foreach count constraint c ∈ C do
3      S_c ← GreedyCountConstraint(c)
4      foreach item i ∈ S_C do
5          if |S| < k then
6              S ← S ⋃{i}
7  foreach coverage constraint c ∈ C do
8      S_c ← GreedyCoverageConstraint(c, S)
9      foreach item i ∈ S_C do
10         if |S| < k then
11             S ← S ⋃{i}
12 while |S| < k do
13     i ← an item in I\S with the highest utility w.r.t. q
14     S ← S ⋃{i}
15 return S
```

Formally, the processing is depicted in Algorithm 2. For every count constraint, we call the GreedyCountConstraint procedure, and add the returned items to S (lines 2−6). We then iterate over all coverage constraints. For every coverage constraint, we call the GreedyCoverageConstraint procedure with the current set S, and add the returned items to S (lines 7−11). Last, if $|S|<k$, we add to S the items with the highest utility scores from $\mathcal{I}\backslash S$ (lines 12−14).

Example 4.1: Let $q=q_1$. Algorithm 2 first process the constraint c_2, adding to S item i_8 (as it has the highest probability of having the brand Versace). Assume that Algorithm 2 considers c_1 before c_3. Next, we process c_1 with $S=\{i_8\}$. Since the number of different brands in S is 1, we update the lower and upper bounds of c_1 to be 1 and 2, resp. We also set the probabilities of all items to have the brand Versace to 0. This procedure returns item i_5, as its brand is Nike with probability of 1. We then process c_3 where $S=\{i_8, i_5\}$. We consider item i_8 as having long sleeves, and item i_5 as having short sleeves. Thus, the lower bound of c_3 is updated to 0, and hence the procedure returns no additional items. Last, sine $|S|=2$, we add item i_1 to S, as it has the highest utility score in $\mathcal{I}\backslash S$, and terminate returning $S=\{i_8, i_5, i_1\}$.

Remark: We note the following limitations of the greedy algorithm. First, in the worst case, no available slots are left before the algorithm has examined all constraints. Second, even if available slots do exist, the final constraints satisfaction probability may be low. Nonetheless, as we show in our experiments, this best-effort algorithm is in practice highly effective for real-life scenarios. Last, as mentioned, the algorithm uses random order over the constraints, and, in case there are available slots, adds items based solely on their utility. We leave for future work optimizing the constraints order and items addition, to increase the satisfaction probability.

B. Local Search Algorithm

The main idea of local search is to move from one solution to a different solution, by applying local changes, until a solution deemed (local) optimal is found or a time bound is elapsed. Such algorithms start from a candidate solution and iteratively move to "neighbor" solutions. To apply local search in our setting, we define a similarity measure between two item-sets, to be used to define the neighborhood relation.

Given a k-size item-set I, and a set of constraint C, let i be an item from I, j is an item from $\mathcal{I}\backslash I$, and let $I'=I\backslash\{i\}\bigcup\{j\}$. We define the similarity between I and I', $sim(I, I')$, as the absolute difference between the probability I and I' satisfy all constraints in C. Formally, $sim(I, I'):=|Pr_C[I]-Pr_C[I']|$. Given a threshold $\theta\in[0, 1]$, we say that I' is a neighbor solution of I if the change in satisfaction probability when moving from I to I' is at most θ, i.e., $sim(I, I')\leq\theta$. A k-size item-set has no more than $k\cdot(n-k-1)$ neighbor solutions, where $n=|\mathcal{I}|$.

Let C be a constraint set, I is a k-size item-set, and I' is a set obtained by replacing one item from I. As noted, computing $Pr_C[I]$ exactly takes exponential time, and hence the same holds for $sim(I, I')$. Let $\widehat{Pr_C[I]}$ (resp. $\widehat{Pr_C[I']}$) denote the estimated constraints satisfaction probability of I (as described in Section II-C). We have showed that for fixed $\epsilon, \delta>0$: $|\widehat{Pr_C[I]} - Pr_C[I]|\leq\epsilon$ with probability $1-\delta$, using m possible worlds. Denote $\widehat{sim(I, I')}= |\widehat{Pr_C[I]}-\widehat{Pr_C[I']}|$.

Lemma 4.2: Given a threshold $\theta\in[0, 1]$, a set of constraints C, and two k-size item-sets I and I', if $\widehat{sim(I, I')}\leq\theta-2\epsilon$ then $sim(I, I')\leq\theta$, with probability $\geq 1-\delta$.

From Lemma 4.2, we can derive which neighbor solution I' of I is superior to I, by examining the utility scores of the items. Formally, let q be a search query, C is a set of constraints, $\epsilon, \delta>0$, and I, I' are two item-sets s.t. $sim(I, I')\leq\theta$ with probability $\geq 1-\delta$. We can show the following:

Theorem 4.3: If $u_q(j)>\frac{u_q(I)\cdot\theta}{1-\theta}$ then: $Pr_c[I']\cdot u_q(I')> Pr_c[I]\cdot u_q(I)$, with probability $\geq 1-\delta$.

The local search algorithm is depicted in Algorithm 3. Given a threshold θ, we set $\epsilon=\frac{\theta}{2}, \delta=1-\epsilon$, and derive the number of possible worlds to be generated (line 1). We then preform t local changes to the solution S. In each iteration, we choose a random item $i\in S$ (line 3), then look for a candidate item $j\notin S$ to replace i (line 4), using Procedure 4, and replace i with j (line 5). Algorithm 4 depicts the procedure for finding the best candidate item, given an item to be replaced i. For every item $j\in\mathcal{I}\backslash S$, let S' to be the solution obtained by replacing i with j (line 4). We examine whether $\widehat{sim}(S, S')\leq\theta$ (with probability $\geq 1-\delta$), according to Lemma 4.2. If so, we examine if the utility of j is high enough to ensure improvement, according to Theorem 4.3 (lines 5−7). Among all items that satisfy these conditions, we select the one with the highest utility (lines 8−9). If no such item exists, we return the original item i.

Example 4.4: Continuing with our running example, let θ=0.1 and t=10. Algorithm 3 starts with the solution returned by Algorithm 2, i.e., $S=\{i_1, i_5, i_8\}$. For i_1 there are no alternative items, as it has the highest utility score w.r.t. q_1. For i_5, the only alternative is item i_7. At this point, also for i_8 there are no alternative items. Therefore, w.h.p., in one of the iterations Algorithm 3 will select item i_5 and will replace it with i_7. As item i_7 also has the highest utility score w.r.t. q_1, it has no alternative items, yet now we can replace i_8. The best alternative for i_8 is item i_2, as for all items that are not in S, it has the highest utility score. Hence, w.h.p., in one of the following iterations, Algorithm 3 will select item i_8 and will replace it with i_2, resulting with the (optimal) set $S=\{i_1, i_2, i_7\}$.

As mentioned, when no improved solution is present in the neighborhood of a solution I, the local search algorithm is stuck at a locally optimal point. A standard optimization to

Algorithm 3: Local Search Algorithm

input : A constraint-set C, a search query q, a threshold θ, number of iteration to executed t, and an item-set S.
output: A k-size item-set.

$\epsilon \leftarrow \frac{\theta}{2}$, $\delta \leftarrow 1-\epsilon$, $m \leftarrow \frac{0.25}{\delta \cdot \epsilon^2}$
foreach *counter* $= 1, ..., t$ **do**
  $i \leftarrow$ an item from S chosen uniformly at random
  $j \leftarrow$ FindBestCandidate(S, i, θ, C, q, m)
  $S \leftarrow S \setminus \{i\} \bigcup \{j\}$
return S

Algorithm 4: FindBestCandidate Procedure

input : An item-set S, an item $i \in I$, a threshold θ, a constraint-set C, a search query q, and the number of random worlds to be generated m.
output: An item from $\mathcal{I}$.

item = i
foreach $j \in \mathcal{I} \setminus S$ **do**
  $S' \leftarrow S \setminus \{i\} \bigcup \{j\}$
  $\widehat{sim} \leftarrow$ EstSim(S, S', C, m)
  if $\widehat{sim} \leq \frac{\theta}{2}$ **then**
    if $u_q(j) > \frac{u_q(S) \cdot \theta}{1-\theta}$ **then**
      if $u_q(j) > u_q(item)$ **then**
        item $\leftarrow j$
return item

avoid this is to extend the neighborhood relation. Namely, we can replace $k' < k$ random items from I with k' random items from $\mathcal{I} \setminus I$ at the same iteration. Here again, we devise sufficient conditions on the set of items to be included, to ensure an improvement (see full details in [12]).

Remark: We note that in case the initial set (obtained by the greedy algorithm) has a low constraint satisfaction probability, the resulting set is likely to have a low constraint satisfaction probability as well. Nonetheless, as we show, in most of the examined scenarios, the constraints satisfaction probabilities of the sets obtained by the greedy algorithm were higher than those of the item-sets that are currently returned by the system. This was the case also for the item-sets obtained by our full two-step algorithm, even though local search may decrease the constraints satisfaction probability.

Time Complexity: The time complexity of the GreedyCountConstrint procedure is dominated by the time it takes to sort the items, which is $O(n \log(n))$, where $n = |\mathcal{I}|$. Similarly, the time complexity of the GreedyCoverageConstrint procedure is dominated by the items it takes to sort the items per value, which is $O(|V_a| \cdot n \log(n))$, where $|V_a|$ is the number of values of the attribute a. Let r denote the maximal number of values for an attribute $a \in \mathcal{A}$. The time complexity of the full greedy algorithm is therefore $O(l \cdot r \cdot n \log(n))$, where l is the number of constraints. Algorithm 3 calls Algorithm 4 t times, which estimates the similarity of two item-sets $(n-k)$ times, where t is the number of iterations. A similarity estimation can be done in $O(m \cdot l \cdot k)$, where $l = |C|$ and $m = O(\frac{1}{\theta^3})$. Therefore, the time complexity of Algorithm 3 is $O(t \cdot (n-k) \cdot l \cdot k \cdot \frac{1}{\theta^3})$.

V. Extensions

We next extend PCS to handle multiple queries. One possible approach to do so is to simply handle each query separately. However, if the number of the overall data requests is bounded, it is desirable to focus on items that would be beneficial to multiple queries. We thus extend our problem definition to handle multiple queries simultaneously. We consider two common aggregation strategies [17]): *Average* and *Least Misery* (LM). Using the average strategy, the goal is to find an item-set that maximizes the average contribution across all queries. Following the LM strategy, the goal is to maximize the minimum contribution for each query. We begin by formally defining the AVG-PCS and LM-PCS problems, then explain how our algorithm can be generalized to handle multiple queries. Last, we discuss how our proposed algorithm can be used to investigate the interactions among the constraints.

A. Multiple Queries

For simplicity of presentation, we assume that the same set of constraints is imposed over all queries, However, all of our definitions and algorithms naturally extended to the general case, where each query may be associated with a different set of constraints (full details are provided in [12]).

We are given a set of m search queries $Q = \{q_1, ..., q_m\}$, and a set of constraints C, imposed over all queries in Q. In AVG-PCS, the goal is to find a k-size item-set that maximizes the average contribution across all queries in Q. Formally, given a number k, find an item-set $I \subseteq \mathcal{I}$ s.t:

$$I = argmax_{|I'| \leq k} \frac{\sum_{q_i \in Q} u_{q_i}(I') \cdot Pr_C[I']}{m} \quad (1)$$

An alternative problem definition follows the Least Misery strategy [17], and maximizes the minimum contribution for each query in Q. Formally, find a set of items $I \subseteq \mathcal{I}$ s.t:

$$I = argmax_{|I'| \leq k} \min_{i \in [1,...,m]} u_{q_i}(I') \cdot Pr_C[I'] \quad (2)$$

Here again, we set the size of the selected item-set in both definitions to be k – the number of items in a query result set. As in the case of a single query, if the number of data requests, k', is $< k$, we take the top k' items having the highest utility scores, and if $k' > k$, we may repeat the process $\frac{k'}{k}$ times.

Example 5.1: Let $Q = \{q_1, q_2\}$. Following the average policy, the selected item-set is: $S_{AVG} = \{i_2, i_6, i_7\}$. Informally, i_7 is relevant for both queries, and items i_2 and i_6 are relevant only for q_1 and q_2, resp. Hence, completing missing data on these items may improve the results of both queries. Following the LM policy, the selected item-set is $S_{LM} = \{i_6, i_7, i_9\}$. Intuitively, as there are fewer items that are relevant for q_2 than for q_1, optimizing the result of q_2 is more challenging. Thus, this set includes more items that are relevant for q_2 (all items in S_{LM}) than items that are relevant for q_1 (only items i_7 and i_9).

We next explain how our algorithm can be extend to support the multiple queries variants. For space limitations, we provide here only a brief overview of the algorithms.

AVG-PCS Given a set of search queries $Q = \{q_1, \ldots, q_m\}$, we compute the average utility score of an item $i \in \mathcal{I}$ w.r.t. all $q \in Q$. Using the average utility scores, we employ the greedy procedure to obtain an initial solution, then use the local-search procedure to improve it, without any further adaptations.

LM-PCS Given a set of search queries Q and a set of constraints C, for every query $q_i \in Q$, we compute the solution obtained by running our algorithm for a single query. Let S_{q_i} denote the solution obtained for the i-th query. We then estimate, for each $q_j \in Q$, the value of $u_{q_j}(S_{q_i}) \cdot Pr_C[S_{q_i}]$ (as described in Section IV-B). Finally, we return the item-set in which its minimum value w.r.t. to all queries in Q is maximal. To speedup processing time, we employ parallelism and compute the solution for all queries and estimate their respective contributions in parallel. Indeed, as we show in

TABLE II: Examined Datasets.

Dataset	n	$\|\mathcal{A}\|$	# of queries	# of constraints
Amazon [19]	431K	12	15	45
CrowdFlower [20]	32K	6	10	31
Victoria's Secret [21]	31K	6	10	30
Home Depot [21]	124K	17	14	39
DM-Authors [22]	10K	10	10	50

our experiments, the overhead for this algorithm is marginal compared with the AVG-PCS algorithm.

B. Investigating Interactions Among The Constraints

Given a set of constraints, the system owner may wish to understand the interactions among the constraints. We note that this problem is orthogonal to PCS and may serve as a pre-processing step that the system owner runs *before running our algorithm*. Nevertheless, we next show how our algorithm can be used for this problem as well. Given a set of constraints C, a search query q, and a new constraint $c \notin C$, one may wish to asses the effect of adding c to C[1]. Namely, to learn if this new constraint may lead to a significant decrease in utility or constraint satisfaction probability. To this end, we run our algorithm twice, once while considering the original set of constraints C, and once while adding c into C. Let $C'=C\cup\{c\}$, and let S_C (resp. $S_{C'}$) denote the solution obtained whit the constraints set C (resp., C'). We examine the difference between S_C and $S_{C'}$ in terms of utility and constraint satisfaction probability. If the difference between $u_q(S_C)$ and $u_q(S_{C'})$ is "significant" (e.g., above a predefined threshold), we may alert the system owner that the constraint c may lead to a significant decrease in utility. Similarly, if the difference between $Pr_C[S_C]$ and $Pr_{C'}[S_{C'}]$ is significant, we may alert the system owner that c may contradict other constraints.

A related problem is the well-studied constraint satisfaction problem [18], whose goal is to verify if there exists a k-size item set that satisfies the constraints. Namely, to verify if there is an assignment to the items' missing attribute values that can satisfy the constraints. Note that this problem is decidable and that a naive approach requires an exhaustive search to examine all attribute-value assignments and for each one to consider all k-size item sets. Here again, note that this problem is orthogonal to the PCS problem. As we mention in Section II-A, in this work, we assume that each constraint can be satisfied and that all constraints can also be satisfied simultaneously.

VI. Experimental Study

Our experimental study conducted to examine the quality and scalability of our algorithm, in multiple practical scenarios.

A. Experimental Setup

We implemented all algorithms in Python 3.7. The experiments were executed on a Linux server with a 2.1GHz CPU, 24 cores, and 96GB memory.

Datasets: We examine 5 datasets spanning two domains: e-commence and people search.

E-commerce We examined 4 publicly-available datasets (listed in Table II), containing search queries and top-k results in several websites. As the obtained results over these datasets demonstrated similar trends, for space limitations, we provide the results only for the Amazon dataset. Amazon is a real-world dataset, containing thousands of products in various categories, including Electronics. The utility function was computed according to search query relevance (using ElasticSearch [16]). The set of attributes includes the products' screen size, brand name, and charger type. The search queries were taken from the BestBuy dataset [23]. For all e-commerce datasets, we extracted the true values of missing attribute values using semi-automatic methods with manual validation (performed by domain experts) over the relevant items. For ML value prediction, we used the NER model of [7], a top-performing neural-network based model that can be used to predict attribute values from text (i.e., products' titles in our setting). We used sets of real-life constraint examples provided by a large e-commerce company (name omitted for privacy request), which includes constraints ensuring specific brand partnerships, gender plurality, and etc. The list of the constrains is private and hence cannot be made fully public. On average, each search query is associated with 3 constraints.

People Search. To demonstrate the applicability of our framework to other domains, we included the DMA dataset, which contains information on researchers who have frequently published in data management-related conferences. The set of attributes includes the researchers' gender, affiliation, continent, research topics, and seniority group. We hid the original values of the authors' gender and continent (to serve as ground truth) and used ML tools to predict their probabilities based on the authors' names. For gender, we used the gender-guesser library [24], and for the continent, we used the algorithm of [25]. For utility scores, we used the researchers' h-index. Here we consider constrained search queries selecting a set of 30 program committee (PC) members.The constraints, obtained by consulting with previous program chairs of major database conferences, include coverage constraints on the authors' continents (ensuring that authors from at least 3 different continents are selected), coverage of different seniority groups (ensuring authors with at least 3 different seniority levels are selected), and coverage constraints on the authors' research topics (ensuring that the number of different research topics is at least 10). The constraints also included count constraints, requiring that the selected author-set will include between $30\%-70\%$ females and $10\%-40\%$ industry affiliates. Namely, the search queries are associated with a set of 5 constraints.

For all datasets, we extracted a smaller subset to be used in the execution of the non-scalable optimal algorithm for the PCS problem variants. These subsets include only the top 50 items/authors with the highest utility scores. We used the Chi-squared test of independence to find subsets of dependent attributes[2], to identify dependent constraints. We identified several correlated attributes in each dataset (e.g., affiliation type and name in the DMA dataset). We treat constraints imposed on correlated attributes as dependent constraints.

Baseline Algorithms: To quantify the usefulness of our proposed two-step algorithm, we assess each of the steps' effectiveness and compare its results with the optimal solution. We thus examine the following baselines:

DO-NOTHING: the algorithm that does not attempt to improve data at all. This no-op algorithm serves as the base no-data-completion case, to which all algorithms are compared.

[1]This procedure can be naturally extended for a set of queries and a set of constraints C', where $C\cap C'=\emptyset$.

[2]All attributes in all datasets are categorical.

CS: this algorithm selects for completion the items returned by the system (i.e., by employing a CS algorithm). The attribute values are assumed to be the ones with the highest probability according to the ML modules. Since CS is itself a hard problem (see Section III-B), for small datasets we compute its optimal solution, whereas for large datasets we use ElasticSearch [16].
GREEDY: the greedy algorithm, as described in Section IV-A.
LS: the local search algorithm described in Section IV-B, which starts with a given item-set and improves it. We consider two variants - one that starts from the CS results (LS(CS)), and one that starts from the GREEDY result (LS(GREEDY)).
ACTIVE: we consider three baselines inspired by Active Learning (see discussion in Section VII). The first variant selects the top-k items with the highest *uncertainty score*. The uncertainty score of an item i is defined as one minus the probability that all of its attribute values are correct. To account for utility, we consider two more variants that, resp., use utility only, and the utility multiplied by the uncertainty score as the ranking metric. Since the results of all three baselines were consistently inferior to the results of other competitors (except DO-NOTHING), we omit them from presentation.
OPT: last, we also provide the optimal solution for the PCS problem variants computed using an exhaustive search.

We report the average results obtained by each baseline over 10 runs. As a default setting, we set the bound on the number of data requests to be k - the size of a query result set. For the LS algorithm, we set the similarity threshold θ to be 0.2, and t, the number of iterations, to $k\times3$.

B. Quality Evaluation

To evaluate the quality of the algorithms we simulate the following data melioration process. We are given a constrained search query q (or a set of queries). Given the item-set returned by an algorithm, we complete all missing data on its items according to the ground truth values (i.e., replacing the predicted values with the ground truth ones). The remaining missing attribute values of other items are unchanged (i.e., are assigned with the value having the highest probability according to the ML modules). We then compute the optimal solution for the CS instance over the resulting database. Next, we examine the quality of the returned set for q in terms of utility and constraint satisfaction (computed w.r.t. the ground truth). The improvement is measured w.r.t. the DO-NOTHING baseline, which shows the system's results for *q before the cleaning was done*. The greater the distance between an algorithm's results from the DO-NOTHING results, the better is the algorithm.

Here we consider the datasets' small versions. We examine two settings: (1) *single-query*, where each search query is considered separately, and report the average result over all queries[3]; (2) *multiple-queries*, where all queries are considered simultaneously. For space limitations, in each experiment we report the results only for one dataset, altering between the Amazon and the DMA datasets. We report that in each case the results over the other dataset demonstrated similar trends. We measure improvement along three dimensions: (1) Overall utility. We report the normalized utility score, computed by dividing the aggregated utility score by the optimal attainable utility score - the aggregated utility of the top k items with the highest utility; (2) Constraint satisfaction probability (computed as explained in Section II-C); (3) Overall utility multiplied by the satisfaction probability (i.e., the objective function of PCS). Figure 2 depicts the improvement of the results achieved over the Amazon dataset.

In all our experiments, the completion of missing data always improves a result-set in terms of constraint satisfaction probability. Note, however, that revealing the real attribute values for certain items may decreases the utility, as some items selected by the underlying CS algorithm to meet the constraints' requirements (due to wrongly inferred attribute values), may now have a different (correct) value that no longer satisfies them. Indeed, this was the case in 16% of the examined queries. Namely, revealing the real attribute values decreased the (normalized) overall utility of 16% of the examined queries by 0.1% on average. In return, their satisfaction probability was increased by 0.3% on average.

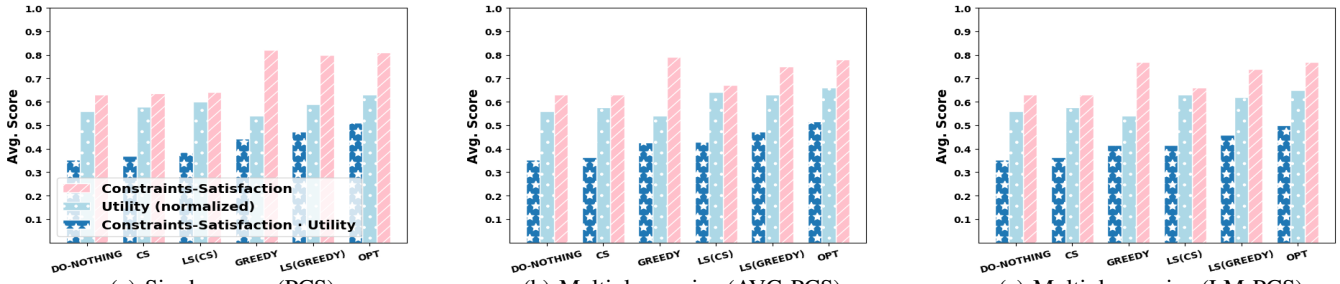
In all cases, the results of DO-NOTHING were inferior to OPT (and to LS(GREEDY)), showing the importance of adequate data completion. This highlights the imperfection of common ML solutions, and that relying solely on data derived by such algorithms may lead to sub-optimal results. Observe that LS(GREEDY) is the second-best competitor after OPT. The LS step significantly improved the results in terms of utility, no matter which result set it had started with. When handling multiple queries simultaneously, the results have also been improved in terms of constraint satisfaction (Figures 2b, 2c). However, the LS step yields the best results when starting from the solution obtained by GREEDY, as this set is more likely to satisfy the constraints than the result set of CS.

We next examine how different parameters affect the results. For space limitations, we present the results only for the single-query setting. We report that the results over the multiple queries settings demonstrated similar trends.

# of data requests: We examine how the number of requests affects performance. The results over Amazon are depicted in Figure 3(a), where the x-axis represents the number of data completion requests and the y-axis represents the probability of constraint satisfaction multiplied by the utility score (i.e., contribution). Naturally, with more data requests, the greater is the improvement. Observe that besides OPT, LS(GREEDY) outperforms all competitors with any number of data requests. The second best competitors here are GREEDY and LS(CS), with only a small difference between them.

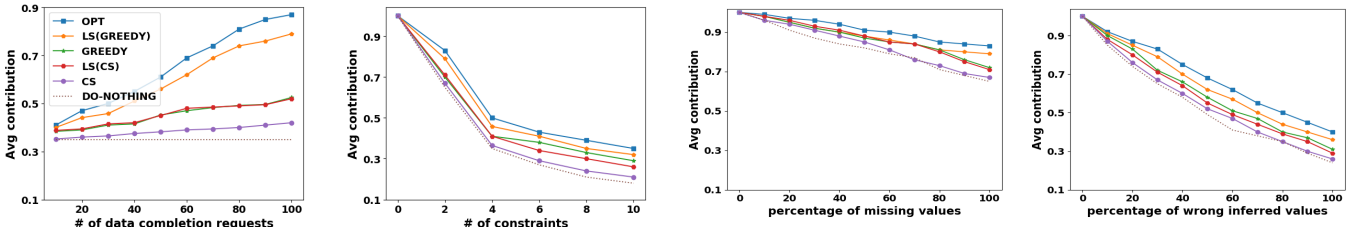
# of constraints: We examine how the number of constraints affects performance. Here we randomly generate constraints as follows. Given an attribute a sampled uniformly at random, we generate (with equal probabilities) either a count or a coverage constraint c. For a count constraint, we randomly selected a value $v\in V_a$, and set the lower (resp., upper) bound of c to be a random number in $[1,k]$ (resp. $[floor,k]$). For a coverage constraint, we set the lower (resp. upper) bound to be a random number in $[1,|V_a|]$ (resp., $[floor,|V_a|]$). The results over Amazon are depicted in Figure 3(b). Naturally, with no constraints at all, there is no need to complete data, as the top k items with the highest utility are the optimal solution. As the number of constraints increases, it becomes harder to find high utility items that also satisfy the constraints. Here again, besides OPT, LS(GREEDY) achieves the best results.

[3]In all cases, the standard deviation was ≤ 0.05.



(a) Single-query (PCS) (b) Multiple-queries (AVG-PCS) (c) Multiple-queries (LM-PCS)

Fig. 2: Improvement in the results in terms of constraints-satisfaction probability and utility.



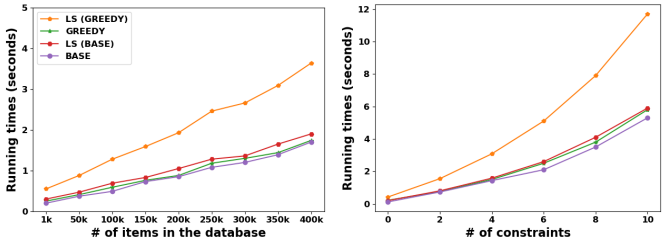
(a) # of data requests. (b) # of constraints. (c) # of missing values. (d) quality of predictions.

Fig. 3: Improvement as a function of various parameters.

Data quality: Here we examine how the percentage of missing values and their prediction quality affect the results. As mentioned, in the DMA dataset, the authors' gender and country names were replaced with values inferred by ML modules. We vary the percentage of missing values by randomly selecting authors whose ground truth values were kept. The results are shown in Figure 3(c). Naturally, with no missing values, the platform outputs the optimal solution, and hence none of the algorithms improve the solution. As the number of missing values increases, the number of wrongly inferred values increases as well. Observe that, here again, LS(Greedy) is consistently the second-best competitor after OPT. Next, we vary the percentage of wrongly inferred values, to examine the effect of the underlying ML module. We report that for both gender and country name predictions, the percentage of wrongly inferred values were 0.78 and 0.67, resp. The results are depicted in Figure 3(d). Observe that where the quality of the ML is poor, all competitors' ability to improve the results is limited. However, we note that there is no point in using ML modules with low accuracy in real-life scenarios.

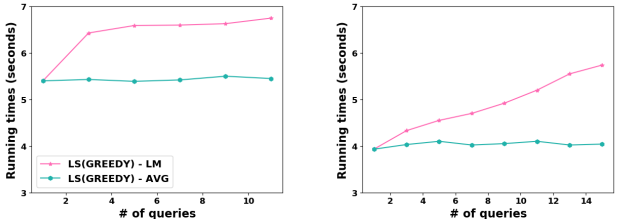
# of queries: We examine how the number of queries affects the results of the AVG-PCS and LM-PCS algorithms. For space limitations, full details are deferred to [12]. Naturally, with more queries considered simultaneously, the contribution for each query becomes smaller. We report that in all cases, LS(GREEDY) is the closest competitor to OPT. Even though the results are almost similar, as expected, the AvG-PCS algorithm achieves a slightly better average contribution than the LM-PCS algorithm. On the other hand, the LM-PCS algorithm ensures that the minimum contribution is slightly higher than the minimum contribution while using the AVG-PCS algorithm. These results indicate that the adaptation of our algorithms for the AVG-PCS and LM-PCS problem variants is adequate for the corresponding optimization problem.

Interactions among the constraints: Last, we used the procedure describe in Section V-B to examine the interactions among the constraints. We observed no particular interactions among the constraints in the Amazon dataset. Not surprisingly, in the DMA dataset, we discovered that the coverage constraint



(a) # of items. (b) # of constraints.

Fig. 4: Running times as a function of various parameters.



(a) DMA. (b) Amazon.

Fig. 5: Running times as a function of # of queries.

imposed on the authors' seniority level decreases the overall utility by nearly 30%. Other constraints had no particular effect on the overall utility or constraint satisfaction probability.

C. Scalability Evaluation

We examine how the running times of our algorithm are affected by various parameters. Here we consider the full datasets, and thus, we do not report the results of OPT. Since the running times of the DO-NOTHING baseline are 0, we omit this baseline from presentation. For space limitations, in the first two experiments, we focus on the single-query setting and the Amamzon dataset, and report that the results over the other settings and datasets demonstrated similar trends.

Repository size: The results are depicted in Figure 4(a). Naturally, with more items, the running times increase for all algorithms. Interestingly, GREEDY behaves similarly to CS (ElasticSearch in this case), both searching for an itemset that is likely to satisfy the constraints. Observe that the

LS step takes longer over the GREEDY set than on the CS one. But, as shown earlier, yields a result of better quality. In all cases, LS(GREEDY) runs in less than 5.5 seconds, which is reasonable for an offline computation, and negligible compared to the time it typically takes to collect missing data. We report that in the DMA dataset, there was a greater increase in running times for all algorithms as the dataset size grows. This stems from the fact that in the DMA search queries, all authors are considered as possible candidates. By contrast, in Amazon, only a subset of the data qualifies for each query (e.g., for the query "laptop", only items having the value of laptop for the attribute device-type are considered).

# of constraints: The results are depicted in Figures 4(b). We observe a near-linear growth in the running times of GREEDY as the number of constraints increases. On the other hand, for LS(GREEDY), we observe a near-exponential growth. This stems from the fact that the estimation of the constraint satisfaction probability this algorithm employs (line 4 of Algorithm 4) is highly affected by the number of constraints. Recall that in both datasets, each query is associated with no more than 5 constraints. Nonetheless, LS(GREEDY) terminates after no more than 12 seconds, even for 10 constraints - a reasonable processing time for an offline task.

# of queries: Next, we examine the effect on running times of our algorithm, adapted to solve the two multiple-queries problem variants, as a function of the number of queries. The results are depicted in Figure 5. The running times of the LS(GREEDY) algorithm adapted to solve AVG-PCS are independent of the number of queries, and it behaves like a single-query PCS algorithm. This stems from the fact that after computing the average utility scores, there is no difference between these algorithms. In contrast, the algorithm adapted to solve LM-PCS demonstrates a near-logarithmic growth in running times. The reason for this is that, due to parallelism, we can process the queries simultaneously. However, this algorithm performs an additional estimation step, for choosing the solution that maximizes the minimum contribution for each query (as described in Section V-A). If the number of considered queries will be higher than the number of available cores, we expect to see a linear growth in running times.

Local search parameters: Last, we examine how the parameters of the local search algorithm affect its performance. For space limitations, we provide an overview of our main findings. According to our experiment, when the similarity threshold increases, we observe a near-logarithmic growth in the result quality; however, this comes at the cost of a near exponential growth in running times. This stems from the fact that with a grater similarity threshold, the size of the neighborhood relation of a solution increases exponentially, and there are a few candidate items that may improve the solution. According to our results, a reasonable value for the similarity threshold is 0.2, as higher threshold values significantly increase running times, while almost not improving result quality. When the number of iterations increases, we observe a near-logarithmic growth in the result quality; here, this comes at the cost of a near-linear growth in the running times. Namely, the number of iterations has a smaller effect on running times compared with the similarity threshold. According to our results, a reasonable value for this parameter is 90 (i.e., approximately $k\times3$ in our setting), as a higher number of iterations significantly increase running times, while almost not improving the result quality.

VII. Related Work

Our work is closely related to a line of work studying different variants of the *non-probabilistic* Constrained Search (CS) problem [2], [26]. For example, as mentioned in Section III-B, the authors of [1] have studied a restricted variant of the CS problem, for which they have devised an optimal PTIME algorithm. While we have established a connection between the general CS and the PCS problems (proving PCS to be harder), we emphasize that our goal is different. Rather than searching for the optimal result set of a constrained query based on *the given information*, we identify those items whose completion may generate better output. Indeed, as our experiments show, the CS results are not the best solution for PCS. A restricted variant of PCS was studied in [3]. However, their model only supports count constraints and assumes equal utility scores. Their proposed algorithms do not generalize to our context, as their hardness results no longer hold in our generalized setting.

Imposing constraints on a result set is known to be important in multiple domains [27], [28]. The two types of constraints that we consider can capture a wide range of constraints studied in multiple applications. For example, coverage constraints have been studied in information retrieval and recommender systems [14], [10], ensuring that a result set covers a wide range of aspects. In algorithmic discrimination, constraints may be imposed on the selected people-set, to increase the representation of disadvantaged populations [1], [26]. In crowdsourcing, much effort was devoted to select an adequate worker-set satisfying coverage requirements [29], [3]. Future work will examine how our results can be applied to improve data quality in such applications.

Another type of constraint, which we intend to study in the future, is similarity-based constraints [27], [9], [10]. Qin et al. [28] formalized the diversified top-k problem. As in our case, they have shown it to be hard to approximate, via a reduction from the maximum independent set problem. We note that what makes this work differently from ours is the details of the reduction itself, which is critical for obtaining our approximation bound. Numerous algorithms for the top-k diversification problem have been proposed [9], [27]. Generally, two main categories of such algorithms are greedy [9] and interchange [10] algorithms. Our algorithm follows an analogous line, applying a greedy procedure followed by a local-search one. However, the different types of constraints that we handle require the design of dedicated greedy and interchange strategies. Interesting future work is to examine how these two complementary lines of work can be integrated to extend our algorithm to support similarity-based constraints.

Data cleansing is a well-studied task. A large body of work focuses on fully-automated cleaning process [30], [31]. To improve results, it is common to combine human knowledge with ML [29]. This was done in numerous domains, including entity resolution [8], and supervised learning [32]. To reduce costs, much research was devoted to minimize the interaction with humans [5], [33]. Our work complements these previous efforts by leveraging the probabilities obtained by ML algorithms, to identify which data records are better to be manually cleaned.

Here as well, we aim to effectively use human effort by limiting the number of data completion requests.

A sub-filed of ML that is closely related to our work is Active Learning, where a learning algorithm iteratively chooses new data points to label (by humans). Similar to our problem, an active learner aims to achieve high performance using a limited number of labeled data points [34]. A common query strategy used to choose which data points to label is *Uncertainty sampling* [35], where an active learner queries the instances about which it is least certain how to label. However, as noted in Section VI, heuristics inspired by this approach performed significantly worse than our algorithm. While in most active learning works instances are selected serially [34], a few frameworks propose to select instances in batches [36], [37]. Most of these approaches use greedy heuristics to ensure that the selected instances are diverse and informative [36], sometimes incorporating diversity constraints [37]. As in our setting, such batch-mode active learning aims to find an optimal (diversity wise), set of instances. However, to the best of our knowledge, none of the existing frameworks account for the general type of constraints studied in our work. A possibly interesting direction for future research may be to devise a batch-mode active learning framework for the PCS problem.

Query evaluation over probabilistic databases has received much attention in recent years [38], [39]. Many uncertain data models (e.g., [39], [13]) adopted the *possible worlds semantics*, where an uncertain relation is viewed as a set of possible instances (worlds). Here as well, we have adopted the possible worlds semantics, assuming that each item is associated with probabilistic attribute values, and used techniques from [13] to estimate the constraint satisfaction probability.

VIII. Conclusion and future work

We presented the PCS problem, which identifies a bounded-size item-set whose data should be completed, to improve the results of constrained search queries. We considered single- and multiple-query settings by supporting common aggregation policies. We provided an approximation bound to PCS, and proposed a heuristic algorithm to solve it. Our experiments demonstrate the advantages of our algorithm in multiple real-life scenarios. An interesting direction for future work is to develop dedicated optimizations for determining the optimal constraints order, to increase the satisfaction probability. Such a technique may use relative weights for the constraints, indicating how important it is to satisfy them.

Acknowledgment: This work has been partially funded by the Israel Science Foundation, the Binational US-Israel Science Foundation, Tel Aviv University Data Science center, and eBay Israel.

References

[1] J. Stoyanovich, K. Yang, and H. Jagadish, "Online set selection with fairness and diversity constraints," in *EDBT*, 2018.

[2] L. E. Celis, D. Straszak, and N. K. Vishnoi, "Ranking with fairness constraints," *arXiv preprint arXiv:1704.06840*, 2017.

[3] T. Wu, L. Chen, P. Hui, C. J. Zhang, and W. Li, "Hear the whole story: Towards the diversity of opinion in crowdsourcing markets," *PVLDB*, 2015.

[4] C. Sun, N. Rampalli, F. Yang, and A. Doan, "Chimera: Large-scale classification using machine learning, rules, and crowdsourcing," *PVLDB*, 2014.

[5] Y. Zheng, J. Wang, G. Li, R. Cheng, and J. Feng, "Qasca: A quality-aware task assignment system for crowdsourcing applications," in *SIGMOD*, 2015.

[6] A. Kannan, I. E. Givoni, R. Agrawal, and A. Fuxman, "Matching unstructured product offers to structured product specifications," in *KDD*, 2011.

[7] Y. Xin, E. Hart, V. Mahajan, and J. Ruvini, "Learning better internal structure of words for sequence labeling," in *EMNLP*, 2018.

[8] J. Wang, T. Kraska, M. J. Franklin, and J. Feng, "Crowder: Crowdsourcing entity resolution," *arXiv preprint arXiv:1208.1927*, 2012.

[9] P. Fraternali, D. Martinenghi, and M. Tagliasacchi, "Top-k bounded diversification," in *SIGMOD*, 2012.

[10] C. Yu, L. Lakshmanan, and S. Amer-Yahia, "It takes variety to make a world: diversification in recommender systems," in *EDBT*, 2009.

[11] I. Guy, T. Milo, S. Novgorodov, and B. Youngmann, "Concierge: improving constrained search results by data melioration," *PVLDB Endowment*, 2020.

[12] "Technical report," 2020.

[13] N. Dalvi and D. Suciu, "Efficient query evaluation on probabilistic databases," *The VLDB Journal*, 2007.

[14] R. Agrawal, S. Gollapudi, A. Halverson, and S. Ieong, "Diversifying search results," in *WSDM*, 2009.

[15] G. Li, J. Wang, Y. Zheng, and M. J. Franklin, "Crowdsourced data management: A survey," *TKDE*, 2016.

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

[17] J. Masthoff, "Group recommender systems: Combining individual models," in *Recommender systems handbook*. Springer, 2011.

[18] V. Kumar, "Algorithms for constraint-satisfaction problems: A survey," *AI magazine*, 1992.

[19] R. He and J. McAuley, "Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering," in *WWW*, 2016.

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

[21] "Kaggle datasets," https://www.kaggle.com/dataset, 2020.

[22] M. Seleznova, B. Omidvar-Tehrani, S. Amer-Yahia, and E. Simon, "Guided exploration of user groups," *PVLDB*, 2020.

[23] E. Dushkin, S. Gershtein, T. Milo, and S. Novgorodov, "Query driven data labeling with experts: Why pay twice?" in *EDBT*, 2019.

[24] "Gender-guesser library," https://pypi.org/project/gender-guesser/, 2020.

[25] J. Lee, H. Kim, M. Ko, D. Choi, J. Choi, and J. Kang, "Name nationality classification with recurrent neural networks." in *IJCAI*, 2017.

[26] K. Yang, V. Gkatzelis, and J. Stoyanovich, "Balanced ranking with diversity constraints," *arXiv preprint arXiv:1906.01747*, 2019.

[27] M. Drosou, H. Jagadish, E. Pitoura, and J. Stoyanovich, "Diversity in big data: A review," *Big data*, 2017.

[28] L. Qin, J. X. Yu, and L. Chang, "Diversifying top-k results," *arXiv preprint arXiv:1208.0076*, 2012.

[29] D. Zhang, H. Xiong, L. Wang, and G. Chen, "Crowdrecruiter: selecting participants for piggyback crowdsensing under probabilistic coverage constraint," in *UbiComp*, 2014.

[30] M. Volkovs, F. Chiang, J. Szlichta, and R. J. Miller, "Continuous data cleaning," in *ICDE*. IEEE, 2014.

[31] M. Yakout, L. Berti-Équille, and A. K. Elmagarmid, "Don't be scared: use scalable automatic repairing with maximal likelihood and bounded changes," in *SIGMOD*, 2013.

[32] A. Ratner, S. H. Bach, H. Ehrenberg, J. Fries, S. Wu, and C. Ré, "Snorkel: Rapid training data creation with weak supervision," *The VLDB Journal*, 2020.

[33] P. Rahman, C. Hebert, and A. Nandi, "Icarus: minimizing human effort in iterative data completion," in *PVLDB Endowment.*, 2018.

[34] B. Settles, "Active learning literature survey," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 2009.

[35] D. D. Lewis and W. A. Gale, "A sequential algorithm for training text classifiers," in *SIGIR*. Springer, 1994.

[36] S. C. Hoi, R. Jin, J. Zhu, and M. R. Lyu, "Batch mode active learning and its application to medical image classification," in *ICML*, 2006.

[37] K. Brinker, "Incorporating diversity in active learning with support vector machines," in *ICML*, 2003.

[38] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang, "Top-k query processing in uncertain databases," in *ICDE*. IEEE, 2007.

[39] A. D. Sarma, O. Benjelloun, A. Halevy, and J. Widom, "Working models for uncertain data," in *ICDE*. IEEE, 2006.