Generating Top-k Packages via Preference Elicitation

Min Xie
Dept. of Computer Science,
Univ. of British Columbia
minxie@cs.ubc.ca

Laks V.S. Lakshmanan
Dept. of Computer Science,
Univ. of British Columbia
laks@cs.ubc.ca

Peter T. Wood
Dept. of CS and Inf. Syst.,
Birkbeck, U. of London
ptw@dcs.bbk.ac.uk

ABSTRACT

There are several applications, such as play lists of songs or movies, and shopping carts, where users are interested in finding top-k packages, consisting of sets of items. In response to this need, there has been a recent flurry of activity around extending classical recommender systems (RS), which are effective at recommending individual items, to recommend packages, or sets of items. The few recent proposals for package RS suffer from one of the following drawbacks: they either rely on hard constraints which may be difficult to be specified exactly by the user or on returning Pareto-optimal packages which are too numerous for the user to sift through. To overcome these limitations, we propose an alternative approach for finding personalized top-k packages for users, by capturing users' preferences over packages using a linear utility function which the system learns. Instead of asking a user to specify this function explicitly, which is unrealistic, we explicitly model the uncertainty in the utility function and propose a preference elicitation-based framework for learning the utility function through feedback provided by the user. We propose several sampling-based methods which, given user feedback, can capture the updated utility function. We develop an efficient algorithm for generating top-k packages using the learned utility function, where the rank ordering respects any of a variety of ranking semantics proposed in the literature. Through extensive experiments on both real and synthetic datasets, we demonstrate the efficiency and effectiveness of the proposed system for finding top-k packages.

1. INTRODUCTION

Recommender systems (RS) have emerged as a popular paradigm for enabling users to find what they might be interested in, complementing search. They exploit feedback provided by users, e.g., in the form of ratings, to build a profile of a user based on users with similar tastes, and use it to make recommendations [1]. However, classical RS are confined to recommending *individual* items. There are several applications where users are interested in packages, i.e., *sets* of items. Examples include play lists of songs on Last.fm and of movies on Netflix, as well as shopping carts of cell phones, accessories, and data plans on Amazon. Since the potential number of packages is exponential in the number of items, finding users with similar tastes for packages is extremely difficult owing to data sparsity. Indeed, current package RS are limited to offering simple interfaces on top of an underlying item RS. E.g., on Amazon, a user has to manually assemble desirable packages by exploring recommended items; on Last.fm and Netflix, a user needs to browse and search packages created by other people. With an exponential number of possible packages, as well as a huge number of user-created packages, both approaches for finding packages of interest quickly become tedious for a user. For a business, this means either lost opportunities to sell packages or poor satisfaction of users overwhelmed by the quantity of user-created content. Clearly, it is desirable to have a system *learn* users' preferences and return packages that are very likely to be of interest to the user.

(a)

item	f_1:cost	f_2:rating
t_1	0.6	0.2
t_2	0.4	0.4
t_3	0.2	0.4

(b)

package	items	package	items
p_1	$\{t_1\}$	p_5	$\{t_2,t_3\}$
p_2	$\{t_2\}$	p_6	$\{t_1,t_3\}$
p_3	$\{t_3\}$	p_7	$\{t_1,t_2,t_3\}$
p_4	$\{t_1,t_2\}$		

Figure 1: Examples of packages of items.

The database community has approached the capturing of preferences from a querying point of view [25]. The idea is to model the utility of an item for a user as a *utility function* over features of the item, using it to return items of highest utility. This notion has been extended to packages or sets of items [20, 29], where the feature values of a package are obtained by aggregating the feature values of the items it contains. For example, Figure 1 shows (a) three items with two features each, and (b) the seven possible non-empty packages. Suppose f_1 represents cost and f_2 represents rating. For a package p, $\mathsf{sum}_1(p)$ defines the total cost of the items in p, i.e., the *cost of the package*, and $\mathsf{avg}_2(p)$ defines the average rating of items in p, i.e., the *quality of the package*. For instance, when purchasing a package of books or CDs from Amazon, a user may want the quality of the package to be as high as possible, and the cost of the package to be as small as possible. Thus, the utility is $U(p) = g(\mathsf{sum}_1(p), \mathsf{avg}_2(p))$, where function g is increasing in $\mathsf{avg}_2(p)$, and decreasing in $\mathsf{sum}_1(p)$. Similar examples can be found when reasoning about the utility of packages on Last.fm and Netflix, where item cost is the price of a song/movie, and item rating can take the form of any

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing info@vldb.org. Articles from this volume were invited to present their results at the 40th International Conference on Very Large Data Bases, September 1st - 5th 2014, Hangzhou, China.
Proceedings of the VLDB Endowment, Vol. 7, No. 14
Copyright 2014 VLDB Endowment 2150-8097/14/10.

combination of average rating, number of listens, number of likes, and number of purchases.

Given this general framework, one way of returning packages to users is to present all *skyline packages* [20, 29], i.e., packages which cannot be *dominated* by another package on every feature. In the above example, a package is a skyline package if there does not exist another package whose cost is lower and quality is higher. However, as shown empirically in [20, 29], the number of skyline packages can be in the hundreds or even thousands for a reasonably-sized dataset, so presenting them all to a user is impractical.

Another way to recommend packages is to define *hard constraints* on some features, and optimize the remaining features in the form of a utility function (e.g., see [27]). For the above example, we could require the total cost of a package to be at most $500, and then find packages with maximum average rating, subject to this cost constraint. Unfortunately, this approach also has the following practical limitations. Firstly, users often only have a rough idea of what they want in a desirable package. E.g., w.r.t. the cost of a package, they may only specify that "smaller is better". Thus, hard constraints on a feature may result either in sub-optimal packages when the budget is set too low, or a huge number of candidate packages when the budget is set too high. Secondly, the importance of each feature to the user is usually unknown. E.g., for some users, monetary budget may not be so important and they can afford to trade a "reasonable" amount of money for higher package quality; other users may be more sensitive to package cost. It is not realistic to expect a user to know, e.g., that they are 0.8 interested in the overall cost, and 0.2 interested in the overall quality of a package!

To address the drawbacks of the above two approaches to package recommendation, we take a *quantitative* approach to rank packages, inspired by recent work on multi-dimensional ranking of items [6, 9, 13]. Specifically, we consider that, when searching for a package of items, each user u has an "implicit" linear utility function U, which captures u's preference or trade-off among different features for choosing a desirable package. E.g., for a user u who attaches equal importance to package cost and quality, the utility function would be $U(p) = -0.5\mathsf{sum}_1(p) + 0.5\mathsf{avg}_2(p)$. However, to account for a user's lack of precise knowledge about their utility, *we assume that the weights of the utility function are hidden.* We follow a *preference elicitation* framework inspired by [6, 9], which *explores* and *exploits* a user's preferences based on feedback received, and learns the hidden weights of the utility function over time.

Preference elicitation (PE) has been studied extensively in the AI community [6, 9]. The general idea of PE is to model users' preferences using a utility function, and then learn the parameters of this utility function through user feedback w.r.t. certain elicitation queries, called *gambling queries.* Though use of gambling queries is grounded in decision theory, to date it has only been applied to applications with *extremely small domains.* Also the form of the gambling query requires that the user be explicitly asked this query by the system through protocols such as user surveys. This limitation makes it unsuitable for deployment in a RS where user feedback needs to be very lightweight, such as "like"s or ratings, or be taken implicitly, e.g., from item click-throughs on web sites.

In this paper, we propose *simple package comparison as the elicitation query.* Users are presented with a list of suggested packages whenever they login to the system. These packages include (i) top packages, w.r.t. the system's current knowledge of the user's utility function, selected according to a chosen ranking semantics as discussed in Section 2.2, and (ii) a set of random packages, which are used to explore the uncertainty in users' preferences. Users' clicks on the suggested packages are logged as implicit signals to the system, showing they are more interested in the clicked package than the other packages (modulo noise in user feedback, discussed in Section 7). Thus the proposed framework can be cleanly integrated into existing applications to capture and update users' preferences, without any disruption to the user (e.g., by presenting multiple gambling queries explicitly).

Specifically, our proposed system assumes the utility function U associated with a user's search is parameterized by a weight vector w. The uncertainty in U is captured by a distribution $\mathbf{P}_w$ over the space of possible weight vectors w. We assume the prior of $\mathbf{P}_w$ is a mixture of Gaussians following [9], which can approximate any arbitrary probability density function. Given $\mathbf{P}_w$, our system can directly leverage it to present the user with a small number of recommended packages, and record the user's feedback on the packages. This feedback can then be leveraged to improve the system's knowledge about U through the posterior of $\mathbf{P}_w$. However, a *major challenge* is that the posterior of $\mathbf{P}_w$, given user feedback, has no closed form solution, as we shall see. To circumvent this, we propose a sampling-based framework which obviates the need for a posterior. Instead, package preferences resulting from user feedback can be translated into constraints on the samples drawn from $\mathbf{P}_w$. However, this raises the question of *how we can obtain samples satisfying the constraints as efficiently as possible.* A related question is whether previously obtained samples can be *maintained* against new user feedback.

We note that following the Bayesian uncertainty-based framework, the posterior distribution of $\mathbf{P}_w$ at any time captures the current optimal representation of a user's preferences over packages, factoring in all observed feedback. We make the following contributions in this paper.

(1) We propose a system which models user preferences using a linear utility function, whose weight vector w is governed by a probability distribution $\mathbf{P}_w$ that captures available information about a user's preference over packages (Section 2). (2) Thereto, we use a non-intrusive Bayesian-based PE framework for eliciting user feedback on recommended packages (Section 2.1). (3) Given user feedback, the posterior of $\mathbf{P}_w$ has no closed form in general, so we employ various constrained sampling strategies to solve this problem, showing how these sampling techniques can leverage the implicit preference feedback (Section 3). (4) We show that an approach based on simple rejection sampling may waste many samples, resulting in poor overall performance, whereas more sophisticated strategies such as importance sampling and MCMC-based sampling make better use of the feedback, and are more efficient (Section 3). (5) Given the utility function U with uncertainty captured by $\mathbf{P}_w$, we discuss how top-k packages can be generated w.r.t. our current knowledge of the user's preferences, following different ranking semantics (Section 4). (6) We address the problem of how to maintain the set of samples generated when new feedback is received, thus further optimizing the number of additional samples needed (Section 3.4). (7) We

demonstrate the efficiency and effectiveness of various sampling methods and ranking algorithms in Section 5.

Related work is covered in Section 6. Finally, we discuss several extensions and conclude in Section 7.

2. PROBLEM SETTING

We assume that we are given a set T of n items, each item being described by a set of m features $\{f_1, \dots, f_m\}$. Each item $t \in T$ can be represented as an m-dimensional vector $\vec{t} = (t_1, \dots, t_m)$, where t_i denotes the value of item t on feature f_i. For simplicity, when no ambiguity arises, we use t to denote both an item and its corresponding feature vector $\vec{t}$. Following [29], without loss of generality, we assume all feature values are non-negative real numbers. In practice, different items can be associated with different feature sets, so some feature values for an item t might be `null`.

As mentioned in the introduction, users' preferences over packages are usually based on *aggregations* over feature values of items in a package. E.g., the sum of the costs of items defines the overall cost of a package, while the average of the ratings of items defines the overall quality of a package. Thus we define an *aggregate feature profile* (or simply *profile*) of a package as follows.

DEFINITION 1. *An* aggregate feature profile *(or* profile*) is defined as $V = (\mathsf{A}_1, \dots, \mathsf{A}_m)$, where each A_i corresponds to feature f_i, $1 \le i \le m$, and is one of the aggregation functions* min, max*,* sum*,* avg *or* `null`*, where* `null` *means that the corresponding feature f_i should be ignored.*

Note that we simplify the presentation by assuming one aggregation per feature, but our algorithms can be easily extended to handle more than one aggregation per feature. Given a package p and a profile V, we define the feature value vector $\vec{p}$ of p w.r.t. V as $\vec{p} = (\mathsf{A}_1(p), \dots, \mathsf{A}_m(p))$, where each $\mathsf{A}_i(p)$ is the aggregate value of items in p w.r.t. feature f_i. Following the usual semantics of aggregate functions, for min, max and sum we have $\mathsf{A}_i(p) = \mathsf{A}_i(\{t_i \mid t \in p \wedge t_i \neq \mathtt{null}\})$, and for avg we have $\mathsf{avg}_i(p) = \mathsf{sum}_i(\{t_i \mid t \in p \wedge t_i \neq \mathtt{null}\})/|\mathsf{p}|$. Similar to the feature value vector of an item, when there is no ambiguity, we simply use p to denote both the package p and its corresponding feature value vector $\vec{p}$. Furthermore, we denote each feature value $\mathsf{A}_i(p)$ of package p by p_i when profile V is clear from the context.

Note that given a fixed item set T, it is trivial to calculate the maximum aggregate value for a feature that can be achieved by any package. E.g., for $\mathsf{avg}_1(p)$, the maximum average value on f_1 that can be achieved by any package is simply the maximum f_1 value of all the items. So we assume in the following that each individual aggregate feature value is normalized to $[0, 1]$ using the maximum possible aggregate value of the corresponding feature.

2.1 Package Utility and Preference Elicitation

Intuitively, the utility of a package p for a user depends on its feature vector and we wish to learn this utility. The space of all mappings between possible aggregate feature values and utility values is uncountable, making this task challenging. Fortunately, most preferences exhibit an internal structure that can be used to express the utility concisely, e.g., an *additive utility function* is commonly assumed in practice [15]. In this work, for a package p and a given profile V, we assume the utility of p can be specified using an additive utility function U, which uses a linear combination of the corresponding (aggregate) feature values in p:

$$U(p) = w_1 p_1 + \cdots + w_m p_m \tag{1}$$

For simplicity, we use w to denote the weight vector $(w_1, \dots, w_m)$. Without loss of generality, we assume each parameter w_i falls in the range $[-1, 1]$, where a positive (negative) w_i means a larger (resp., smaller) value is preferred for the corresponding feature.

A framework based on a utility function essentially defines a total order over all packages, where similar to previous works such as [24], we assume ties in utility score are resolved using a deterministic tie-breaker such as the ID of a package. This differentiates the approach from that of [20, 29] which aim to return all *skyline packages*, the number of which can be prohibitively large, as previously noted.

Despite its intuitive appeal, there are two major challenges in adopting the utility-based framework in practice. First, users are usually not able to specify (or even know) the exact weights w_i of the utility function U. Thus, we must model the uncertainty in U, and elicit user's preferences by means of clicks on recommended packages. Second, unlike [20, 29] which consider packages of fixed size, we allow package size to be flexible in our framework. We believe this is natural. E.g., given a system-defined maximum package size ϕ of say 20, we consider all possible package sizes ranging from 1 to 20. Efficient determination of packages of flexible size that maximize a user's utility under partial knowledge about the utility function from elicited preferences is far more challenging than finding packages of a given fixed size.

One popular way of characterizing uncertainty in U is through Bayesian uncertainty[1] [6, 9], in which for each user, we assume the exact value of the weight vector w is not known, but w can be described by a probability distribution $\mathbf{P}_w$. We assume w follows a mixture of Gaussians: indeed, it has been shown that a mixture of Gaussians can approximate any arbitrary probability density function [3].

While $\mathbf{P}_w$ can be initialized with a system-defined default distribution, in the long run $\mathbf{P}_w$ can be learned by leveraging the feedback provided by the user. In this work, we assume user feedback is in the form of clicks on packages from a set of recommendations presented to them, indicating the clicked packages are more attractive to the user than unclicked packages. This form of feedback is known as *example critiquing* in conjoint analysis [26] and preference elicitation [21].

For a given user u, let the feedback from u, preferring package p_1 to package p_2, be denoted by $p_1 \succ p_2$. This feedback can be leveraged to update the posterior of $\mathbf{P}_w$ through Bayes' rule in Equation 2, where $\mathbf{P}(p_1 \succ p_2 \mid w)$ defines the likelihood of $p_1 \succ p_2$ given w. Note that since each specific w defines a total order over all packages, the value of $\mathbf{P}(p_1 \succ p_2 \mid w)$ is either one or zero. We tentatively assume that every user feedback is consistent, in that the provided preferences correspond to a partial order, and discuss in Section 7 how this assumption can be relaxed.

$$\mathbf{P}_w(w \mid p_1 \succ p_2) = \frac{\mathbf{P}(p_1 \succ p_2 \mid w)\mathbf{P}_w(w)}{\int_w \mathbf{P}(p_1 \succ p_2 \mid w)\mathbf{P}_w(w)\mathrm{d}w} \tag{2}$$

However, Gaussian mixtures are not closed under this kind of update [6], meaning we cannot obtain a closed-form solution for the posterior as presented in the above equation. One popular way to deal with such a situation is to force the posterior to again be a mixture of Gaussians, and thus the posterior can be learned by refitting a Gaussian

[1]The other possibility is strict uncertainty, which requires a set of possible utility functions (down to the weights) to be known, which is more restrictive.

(a)

w	values	prob.
w_1	(0.5,0.1)	0.3
w_2	(0.1,0.5)	0.4
w_3	(0.1,0.1)	0.3

(b)

profile
(sum_1,avg_2)

(c)

utility	p_1	p_2	p_3	p_4	p_5	p_6
w_1	0.35	0.3	0.2	0.575	0.4	0.475
w_2	0.31	0.54	0.52	0.475	0.56	0.455
w_3	0.11	0.14	0.12	0.175	0.16	0.155

(d)

	top-2 packages
w_1	p_4,p_6
w_2	p_5,p_2
w_3	p_4,p_5

Figure 2: Examples of different ranking semantics.

mixture through algorithms such as expectation maximization (EM) [3]. However, the cost of refitting through EM is extremely high, so we take a different approach of representing the posterior by maintaining both the prior distribution and the set of feedback preferences received. The details of our proposal are described in Section 3.1.

2.2 Presenting Packages

While the preference elicitation framework discussed in the previous section can be exploited to update the knowledge of $\mathbf{P}_w$ for any specific user, there is still a remaining question of how to select and present packages to a user in order to receive feedback.

In general, given the uncertainty in the utility function, packages presented to a user serve as a way to *explore* and *exploit* the user's preferences. I.e., our main goal is to exploit our current knowledge about a user's preferences and try to present to them the best packages possible according to the current $\mathbf{P}_w$. On the other hand, we also want to explore the uncertainty in the user's preferences, and present (random) packages to them which might not be considered by our current knowledge about the user's preferences. These packages serve the purpose of correcting bias introduced from the initial distribution of $\mathbf{P}_w$ and combating mistakes and noise from user feedback. In this paper, we follow a simple way of presenting to the user, current best packages along with random packages, so that the current best packages can be used to exploit our current knowledge about the user, and the random ones can be used to explore user preferences.

A challenge in picking the best packages is that there is no universally accepted ranking semantics for packages given the uncertainty in the utility function. Instead of committing to a specific package ranking semantics, we consider alternative semantics studied by different communities, and discuss how they can be integrated into our framework.

The first ranking semantics we consider is based on expectation (EXP), which has been adopted as the most popular semantics for ranking items in preference elicitation papers in the AI community [6, 9]. In the following, by a *package space* P, we mean the set of all possible packages formed using items from T and having size no larger than ϕ (system defined maximum package size).

DEFINITION 2 (EXP). *Given a package space P and probability distribution $\mathbf{P}_w$ over weight vectors w, find the set of top-k packages P_k w.r.t. expected utility value, i.e. $\forall p \in P_k$, $\forall p' \in P \backslash P_k$, $\mathbf{E}_w(w \cdot p) \geq \mathbf{E}_w(w \cdot p')$.*

EXAMPLE 1. *Consider the example in Figure 1. Assuming the maximum package size is 2, the package space P is given by $\{p_1, \ldots, p_6\}$. If the profile under consideration is $(\mathsf{sum}_1, \mathsf{avg}_2)$, then the maximum value for a size-2 package on feature 1 is 1, and the maximum value of a size-2 package on feature 2 is 0.4. We can normalize packages' feature values using these two maximum values. E.g., for package p_1 in Figure 1(b), $\mathsf{sum}_1(p_1) = 0.6$, $\mathsf{avg}_2(p_1) = 0.2$, so the normalized feature value vector for p_1 is $(0.6/1, 0.2/0.4) = (0.6, 0.5)$. To simplify the presentation, we assume in Figure 2(a) that there are only three weight vectors, w_1, w_2 and w_3, under consideration, the probability for which is given in the third column. Given the weight vector information, we can easily calculate the utility of each package under each weight vector, as shown in Figure 2(c). E.g., the utility of package p_1 under w_1 is $0.6 \times 0.5 + 0.5 \times 0.1 = 0.35$. The expected utility value for each package can be calculated accordingly, using the probability of each weight vector. E.g., the expected utility for p_1 is $0.35 \times 0.3 + 0.31 \times 0.4 + 0.11 \times 0.3 = 0.262$. For this example, it is not difficult to verify that p_4 has the largest expected utility, followed by p_5.* □

The second ranking semantics we consider is based on the probability of a package being in the top-σ position under different parameter settings (TKP). This is inspired by recent work on learning to rank in the machine learning community [8]. Let $P_{\succ}(p \mid w) = \{p' \mid p' \in P, w \cdot p' > w \cdot p\}$ denote the set of packages in P which have utility larger than package p, given a fixed w. Let $W_{\succ}$ denote the set of weight vectors w under which a package p is dominated by σ or fewer packages, i.e., $|P_{\succ}(p \mid w)| \leq \sigma$. Since the utility function is convex, we can readily show that $\forall$ w_1, w_2, $w_1 \neq w_2$, if $w_1 \cdot p' > w_1 \cdot p$, and $w_2 \cdot p' > w_2 \cdot p$, then for any $\alpha \in [0, 1]$, $(\alpha w_1 + (1 - \alpha) w_2) \cdot p' > (\alpha w_1 + (1 - \alpha) w_2) \cdot p$. It thus follows that $W_{\neg\succ} := \{w \mid \sigma < |P_{\succ}(p \mid w)|\}$ forms a continuous and convex region, and that $W_{\succ}$ is also continuous. So we define the probability of $p \in P$ being ranked among the top-σ packages as $\mathbf{P}(p \mid \mathbf{P}_w, \sigma) = \int_{w \in W_{\succ}} \mathbf{P}_w(w) \mathrm{d}w$.

DEFINITION 3 (TKP). *Given a package space P and a probability distribution $\mathbf{P}_w$ over weight vectors w, find the top-k packages P_k w.r.t. the probability of being ranked in the top-σ positions, i.e., $\forall p \in P_k$, $\forall p' \in P \backslash P_k$, $\mathbf{P}(p \mid \mathbf{P}_w, \sigma) \geq \mathbf{P}(p' \mid \mathbf{P}_w, \sigma)$.*

EXAMPLE 2. *In Figure 2(d), we show the top-2 package list for each weight vector. We can calculate that the probability of p_5 being in a top-2 package list is $0.4 + 0.3 = 0.7$. Package p_5 has the largest probability of all candidate packages, followed by p_4 for which the probability is 0.6.* □

The third ranking semantics we consider is the most probable ordering (MPO), which has been discussed in recent work on sensitivity analysis of querying top-k items under uncertainty [24]. We note that unlike EXP and TKP which represent the desirability of each individual package independently, adapted to our setting, MPO represents the desirability of the top-k package list P_k as a whole.

For MPO, given a fixed w, let $I(P_k \mid w)$ be an indicator function which denotes whether P_k is the set of top-k packages under w, i.e., $I(P_k \mid w) = 1$ if $\nexists p' \in P \backslash P_k$, $w \cdot p' > w \cdot p$, for any $p \in P_k$; and $I(P_k \mid w) = 0$ otherwise. Let W_{P_k} denote the set of weight vectors w under which $I(P_k \mid w) = 1$. Similar to TKP, we can show W_{P_k} forms a continuous region, so the probability of P_k being the top-k package can be defined as $\mathbf{P}_o(P_k \mid \mathbf{P}_w) = \int_{w \in W_{P_k}} \mathbf{P}_w(w) \mathrm{d}w$.

DEFINITION 4 (MPO). *Given a package space P and a probability distribution $\mathbf{P}_w$ over weight vectors w, find the top-k packages P_k w.r.t. the most probable ordering, i.e., $\forall P'_k \subseteq P$, $|P'_k| = k$, $P'_k \neq P_k$, $\mathbf{P}_o(P_k \mid \mathbf{P}_w) \geq \mathbf{P}_o(P'_k \mid \mathbf{P}_w)$.*

EXAMPLE 3. *In Figure 2(d), we can directly see the probability of each top-2 package list by referring to the probability of the corresponding weight vector. Clearly, the best top-2 package list under MPO is p_5, p_2.* □

In summary, different ranking semantics might lead to different top-2 packages: in our example, the top-2 packages for EXP, TKP, and MPO repectively are p_4, p_5; p_5, p_4; and p_5, p_2. These ranking semantics have been successfully adopted in different communities, and they can all be neatly incorporated into our PE framework (see Section 3).

3. A SAMPLING-BASED FRAMEWORK

To accommodate the preference elicitation framework and various ranking semantics for selecting packages to recommend, we propose to use a sampling-based framework. Unlike the geometric approach proposed in previous work (e.g., see [24]), a sampling-based solution can be easily adapted to handle cases with higher dimensionality, as we show empirically in Section 5. We first discuss simple rejection sampling in Section 3.1. We then consider more sophisticated sampling techniques in Section 3.2. In Section 3.3, we discuss how to optimize the constraint violation checking process for sample generation. Finally in Section 3.4, we discuss how previously generated samples can be reused given newly received user feedback, i.e., we discuss sample maintenance.

3.1 Rejection Sampling

Given the distribution $\mathbf{P}_w$ over w, an intuitive solution for finding the best packages under $\mathbf{P}_w$ is to first sample the weight vectors w according to $\mathbf{P}_w$, and then for every w sampled, try to find the best package under w. The best package results obtained from each sampled w can be aggregated for estimating the final list of best packages. The required aggregation logic depends on the ranking semantics, the details of which will be discussed in Section 4. This approach is intuitive: w's are sampled from $\mathbf{P}_w$, and packages which are ranked higher under those w's that have a higher probability are likely to be given a greater weight.

As discussed in Section 2.1, given current recommendations to the user and the feedback received, we need to constantly refit the distribution $\mathbf{P}_w$ so that it reflects the updated user preferences. However refitting the Gaussian mixture $\mathbf{P}_w$, say using the EM algorithm [6], after every received feedback using Equation (2) can be extremely time consuming. So a naïve way of performing *refit-and-sample* may be inefficient. Thus we consider an alternative approach of maintaining both the prior distribution $\mathbf{P}_w$ and all feedback *without* refitting the Gaussian mixture.

The key idea is that every feedback $p_1 \succ p_2$ rules out weight vectors w under which $p_1 \succ p_2$ is not true. For those w's which do satisfy $p_1 \succ p_2$, the feedback alone does not change their relative order with respect to $\mathbf{P}_w$, i.e., if $\mathbf{P}_w(w_1) > \mathbf{P}_w(w_2)$ for w_1, w_2 which both satisfy $p_1 \succ p_2$, without any further information, we have $\mathbf{P}_w(w_1 \mid p_1 \succ p_2) > \mathbf{P}_w(w_2 \mid p_1 \succ p_2)$, as we show in the following lemma.

Lemma 1. *Given feedback $p_1 \succ p_2$: (1) if $p_1 \succ p_2$ does not hold under weight vector w, $\mathbf{P}_w(w \mid p_1 \succ p_2) = 0$; (2) for two weight vectors w_1, w_2, $w_1 \neq w_2$, if $p_1 \succ p_2$ holds under both w_1, w_2, and $\mathbf{P}_w(w_1) > \mathbf{P}_w(w_2)$, we have $\mathbf{P}_w(w_1 \mid p_1 \succ p_2) > \mathbf{P}_w(w_2 \mid p_1 \succ p_2)$.*

Proof. (1) Consider the likelihood $\mathbf{P}(p_1 \succ p_2 \mid w)$ in Equation (2). If $p_1 \succ p_2$ does not hold under w, clearly the likelihood of $p_1 \succ p_2$ under w is 0, so $\mathbf{P}_w(w \mid p_1 \succ p_2) = 0$.

(2) Consider $\mathbf{P}_w(w_1 \mid p_1 \succ p_2)$ and $\mathbf{P}_w(w_2 \mid p_1 \succ p_2)$. Since $p_1 \succ p_2$ holds under both w_1, w_2, $\mathbf{P}(p_1 \succ p_2 \mid w_1) = \mathbf{P}(p_1 \succ p_2 \mid w_2) = 1$, so $\frac{\mathbf{P}_w(w_1 \mid p_1 \succ p_2)}{\mathbf{P}_w(w_2 \mid p_1 \succ p_2)} = \frac{\mathbf{P}_w(w_1)}{\mathbf{P}_w(w_2)}$. □

So this means that we can use rejection sampling to sample directly from the posterior. I.e., we can sample a random w from the current $\mathbf{P}_w$, and if w violates any user feedback, we reject this sample. Otherwise, the sample is accepted. Clearly the rejection sampling method will only keep samples which conform to the feedback received from the user, and as shown above, the relative order of probabilities of two weight vectors being sampled still conforms to their original relative order following the distribution $\mathbf{P}_w$.

3.2 Feedback-aware Sampling

The simple rejection sampling scheme proposed above may work well when the amount of feedback is small. However, as the amount of feedback grows, the cost of this approach increases, as samples become more likely to be rejected. Thus a better sampling scheme should be "aware" of the feedback received, and try to avoid generating invalid samples, i.e., those that violate any provided feedback constraint.

Recall that user feedback produces a set of pairwise preferences of the form $p_1 \succ p_2$, where p_1 and p_2 are packages. Given a set S_ρ of these preferences, it can be shown that the set of valid weight vectors w, i.e., those that satisfy all feedback preferences, has the following useful property.

Lemma 2. *The set of valid weight vectors which satisfy a set of preferences S_ρ forms a convex set.*

Proof. By definition, for any w_1, w_2 which satisfy S_ρ, $\forall \rho := p_1 \succ p_2 \in S_\rho$, $w_1 \cdot p_1 \geq w_1 \cdot p_2$ and $w_2 \cdot p_1 \geq w_2 \cdot p_2$. Then $\forall \alpha \in [0, 1]$, $\alpha w_1 \cdot p_1 \geq \alpha w_1 \cdot p_2$ and $(1-\alpha) w_2 \cdot p_1 \geq (1-\alpha) w_2 \cdot p_2$. Combining these inequalities shows that any convex combination of w_1 and w_2 also forms a valid w. □

So valid weight vectors form a continuous and convex region. By exploiting this property, we can leverage different sampling methods which can bias samples more towards those which are inside the valid region.

3.2.1 Importance Sampling

The general idea of *importance sampling* is that, instead of sampling from a complex probability distribution (*original distribution*), which in our case is $\mathbf{P}_w(w \mid S_\rho)$, we sample from a different *proposal distribution* $\mathbf{Q}_w$, which is more likely to satisfy the constraints given by the feedback set S_ρ. However, this process will introduce a set of samples which do not follow the original distribution, so we need to correct this *bias* by associating each sample w from $\mathbf{Q}_w$ with an importance weight (or simply weight, when there is no ambiguity) $q(w)$. Next we will discuss how this framework can be employed in solving our problem.

Since valid weight vectors w.r.t. S_ρ form a continuous and convex region, samples which lie close to the "center" of this region are more likely to satisfy S_ρ. However, finding the center of an arbitrary convex polytope is extremely complex and time consuming [5], which negates the motivation for using importance sampling, namely efficiency. Instead, we use a simple geometric decomposition-based approach, which partitions the space into a multi-dimensional grid, and approximates the center of the convex polytope using the centers of the grid cells which overlap with it.

In Figure 3, we show a simple two dimensional example of the above approach. Initially, the entire valid region is divided into a 3×3 grid as depicted in Figure 3(a). Given feedback $\rho := p_1 \succ p_2$, we know any invalid w satisfies the property that $w \cdot p_1 < w \cdot p_2$, or $w \cdot (p_2 - p_1) > 0$, i.e., w is invalid if w is above the line $p_2 - p_1 = 0$. As shown in

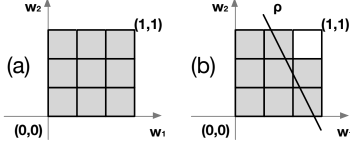

Figure 3: Approximate center of a convex polytope.

Figure 3(b), all w's which are in the top-right grid cell are above the line corresponding to ρ, so we can eliminate this cell from consideration, and the center of the region of valid w's can be approximated by the center of the remaining eight cells. The latter can be calculated by simply taking the average of the eight cell centers.

We note that whether there exists a w in a grid cell which satisfies a constraint ρ can be checked in time linear in the dimensionality of the feature space. Also, finding those cells which violate new feedback can be facilitated by organizing the cells into a hierarchical structure such as a *quad-tree* [12].

Given the center w^* of the valid region, an intuitive choice for the proposal distribution would be a Gaussian $\mathbf{Q}_w \sim \mathcal{N}(w^*, \Sigma)$ with mean w^*, covariance Σ. To correct the bias introduced in samples from $\mathbf{Q}_w$, in importance sampling, we could associate each valid sample w with an *importance weight* $q(w) = \mathbf{P}_w(w)/\mathbf{Q}_w(w)$. Intuitively, this importance weight compensates for the difference between $\mathbf{P}_w$ and $\mathbf{Q}_w$.

The importance weight of each sample $q(w)$ can be easily adapted to different ranking semantics. For EXP, we multiply $q(w)$ by the utility value calculated for each package under consideration for this w. For TKP and MPO, instead of adding one to the corresponding counter of each package w.r.t. the given w, we add $q(w)$.

Performance Analysis: To compare the performance of importance sampling and rejection sampling, we use the classic notion of *Effective Number of Samples* (ENS) [17], which measures the overall efficiency of a sampling algorithm based on a χ^2 distance between the original distribution and the proposal distribution:

$$ENS(\mathbf{P}_w(w \mid S_\rho), \mathbf{Q}_w(w)) = \frac{N}{1 + \chi^2(\mathbf{P}_w(w \mid S_\rho), \mathbf{Q}_w(w))}$$
$$\chi^2(\mathbf{P}_w(w \mid S_\rho), \mathbf{Q}_w(w)) = \int_w \frac{(\mathbf{P}_w(w \mid S_\rho) - \mathbf{Q}_w(w))^2}{\mathbf{Q}_w(w)} \mathrm{d}w \quad (3)$$

where N is the number of samples. Intuitively, the closer the proposal distribution is to the original distribution, the larger the ENS value, meaning more samples out of the N in the sample pool are "effective". ENS reaches its maximum value N when $\mathbf{P}_w(w \mid S_\rho) = \mathbf{Q}_w(w)$, $\forall w$. Note that the rejection sampling of Section 3.1 can be regarded as importance sampling using the prior distribution $\mathbf{P}_w$ as the proposal distribution, except that those samples which do not satisfy ρ will be discarded.

Let $w \vDash \rho$ denote that w satisfies ρ. The following theorem shows that when the same number of samples are generated by importance sampling and rejection sampling, more samples from importance sampling are effective, which is verified by our experiments (Section 5.1).

THEOREM 1. $ENS(\mathbf{P}_w(w \mid S_\rho), \mathbf{Q}_w(w)) \geq ENS(\mathbf{P}_w(w \mid S_\rho), \mathbf{P}_w(w))$

PROOF. Let $W^- = \{w \mid w \nvDash \rho\}$, $W^+ = \{w \mid w \vDash \rho\}$. Because $\mathbf{Q}_w$ is assumed to be a Gaussian whose mean is at the center of the convex region formed by W^+ (according to Lemma 2), samples generated from $\mathbf{Q}_w(w)$ have a higher chance of satisfying user feedback ρ compared to sampling directly from $\mathbf{P}_w$, i.e., for $w \in W^-$, $\mathbf{Q}_w(w) \leq \mathbf{P}_w(w)$. According to Lemma 1, for $w \in W^-$, $\mathbf{P}_w(w \mid S_\rho) = 0$, so we have:

$$\int_{w\in W^-} \frac{(\mathbf{P}_w(w \mid S_\rho) - \mathbf{Q}_w(w))^2}{\mathbf{Q}_w(w)} - \frac{(\mathbf{P}_w(w \mid S_\rho) - \mathbf{P}_w(w))^2}{\mathbf{P}_w(w)} \mathrm{d}w$$
$$= \int_{w\in W^-} \mathbf{Q}_w(w) - \mathbf{P}_w(w) \mathrm{d}w \leq 0 \quad (4)$$

On the other hand, for $w \in W^+$, according to the proposed importance sampling, each sample is corrected with a weight $q(w) = \mathbf{P}_w(w)/\mathbf{Q}_w(w)$, thus the actual probability of generating each $w \in W^+$ is the same as $\mathbf{P}_w(w)$, i.e., $\mathbf{Q}_w(w) = \mathbf{P}_w(w)$, $w \in W^+$. Thus we have:

$$\int_{w\in W^+} \frac{(\mathbf{P}_w(w \mid S_\rho) - \mathbf{Q}_w(w))^2}{\mathbf{Q}_w(w)} - \frac{(\mathbf{P}_w(w \mid S_\rho) - \mathbf{P}_w(w))^2}{\mathbf{P}_w(w)} \mathrm{d}w \quad (5)$$
$$= \int_{w\in W^+} \frac{(\mathbf{P}_w(w \mid S_\rho) - \mathbf{P}_w(w))^2}{\mathbf{P}_w(w)} - \frac{(\mathbf{P}_w(w \mid S_\rho) - \mathbf{P}_w(w))^2}{\mathbf{P}_w(w)} \mathrm{d}w = 0$$

So according to Equation (3), $\chi^2(\mathbf{P}_w(w \mid S_\rho), \mathbf{Q}_w(w)) \leq \chi^2(\mathbf{P}_w(w \mid S_\rho), \mathbf{P}_w(w))$, and $ENS(\mathbf{P}_w(w \mid S_\rho), \mathbf{Q}_w(w)) \geq ENS(\mathbf{P}_w(w \mid S_\rho), \mathbf{P}_w(w))$. □

3.2.2 MCMC-based Sampling

Another popular approach for sampling from complex distributions is the *Markov Chain Monte Carlo* (MCMC) method. This approach generates samples from the distribution by simulating a Markov chain. We construct the Markov chain in such a way that it gives more importance to the regions which are valid, i.e., the stationary distribution of the Markov chain is the same as the posterior $\mathbf{P}_w(w \mid S_\rho)$.

Since valid weight vectors form a single continuous and convex region, we could simply find a first valid weight vector w, and then perform a random walk from w to a new weight vector w'. Note that in order to explore the valid region w.r.t. the current set of preferences S_ρ, it is clearly desirable that each step of the random walk explores only a close region around the current w, as otherwise, w' generated from w may be more likely to be outside the valid region. Thus we define a length threshold l_{max}, and set the transition probability $Q(w' \mid w)$ from w to w' as follows:

$$Q(w' \mid w) = \begin{cases} 1/l_{max} & \text{if } \|w' - w\| \leq l_{max} \\ 0 & \text{otherwise.} \end{cases} \quad (6)$$

One of the most popular MCMC-based sampling algorithms is Metropolis-Hastings (MH). Using MH we can generate samples following the Markov chain defined by $Q(w' \mid w)$. Given a current weight parameter w, we randomly pick a weight parameter w' such that the distance $\|w' - w\|$ is less than or equal to l_{max}. If w' satisfies the preferences in the feedback set S_ρ, w' is accepted as the next sample with a probability α, as defined in Equation (7). If w' is rejected (i.e., with probability $(1 - \alpha)$), we use a copy of w as the next sample.

$$\alpha = \min\{1, \frac{\mathbf{P}_w(w')Q(w \mid w')}{\mathbf{P}_w(w)Q(w' \mid w)}\} \quad (7)$$

Note that $Q(w' \mid w)$ is obviously symmetric in our case, so α can be simply calculated as $\alpha = \min\{1, \frac{\mathbf{P}_w(w')}{\mathbf{P}_w(w)}\}$.

Following recommendations in the MH sampling literature [3], we pick one sample from every δ samples generated, where δ is called the *step length*, rather than including all generated samples in the final sample pool. This avoids generating highly correlated samples.

Performance Analysis: Let $\mathbf{Q}'_w$ be the probability distribution of w obtained by the proposed MH sampler. Obviously $\mathbf{Q}'_w$ converges to the prior distribution $\mathbf{P}_w$ according to Equation (7). So for $w \in W^+$, we have $\chi^2(\mathbf{P}_w(w \mid S_\rho), \mathbf{Q}'_w(w)) = \chi^2(\mathbf{P}_w(w \mid S_\rho), \mathbf{P}_w(w))$.

According to the MH sampler, whenever a new sample w' obtained following $Q(w' \mid w)$ is rejected, we keep a copy of the current w, so the probability of sampling a $w \in W^-$ is 0, which means $\forall w \in W^-$, $\mathbf{Q}'_w(w) = \mathbf{P}_w(w \mid \rho) = 0$. Thus $\forall w \in W^-$, $\chi^2(\mathbf{P}_w(w \mid S_\rho), \mathbf{Q}'_w(w)) \leq \chi^2(\mathbf{P}_w(w \mid S_\rho), \mathbf{P}_w(w))$, yielding the following result.

THEOREM 2. $ENS(\mathbf{P}_w(w \mid S_\rho), \mathbf{Q}_w(w)') \geq ENS(\mathbf{P}_w(w \mid S_\rho), \mathbf{Q}_w(w))$

So the MCMC-based sampler is more effective than importance sampling for our constrained sampling problem, as verified by our experiments (Section 5.1).

3.3 Optimizing Constraint Checking Process

Suppose σ packages $p_1, \ldots, p_\sigma$ are presented to the user. Even if the user clicks on only one package p_1 out of the σ recommended packages, this results in $\sigma-1$ pairwise package preferences: $\rho_1 := p_1 \succ p_2, \ldots, \rho_{\sigma-1} := p_1 \succ p_{\sigma-1}$. Thus, as more feedback is received from the user, the number of package preferences we need to deal with increases quickly.

This raises the following two issues: (1) cycles in preferences, and (2) cost of checking whether a sample w should be rejected. We note that cycles in preferences can be resolved by presenting packages in a cycle to the user, and asking the user to choose the best out of them (which reverses the direction of one edge in the cycle and breaks the cycle). Next we will discuss how a set S_ρ of pairwise package preferences can be organized in order to facilitate efficient checking of whether a sampled weight vector w is valid.

One intuitive solution is to reduce redundant package preferences by exploiting the *transitivity* of the preference relation. It is easy to see that the preference relation $\succ$ over packages is transitive for additive utility functions, i.e., for any packages p_1, p_2, p_3 and any weight vector w, $w \cdot p_1 > w \cdot p_2$ and $w \cdot p_2 > w \cdot p_3$ imply $w \cdot p_1 > w \cdot p_3$. Thus, there is no need to verify satisfaction of $p_1 \succ p_3$ for a sample w whenever w satisfies $p_1 \succ p_2$ and $p_2 \succ p_3$. It follows that the number of preferences we need to check is at most linear in the number of distinct packages (implicitly) appearing in the feedback.

To eliminate redundant preferences received from the user, we can maintain preferences in a *directed acyclic graph* (DAG) G_ρ: an edge (p_i, p_j) represents the preference $p_i \succ p_j$. Then any *transitive reduction* algorithm [2] can be applied to eliminate redundant preferences.

3.4 Sample Maintenance

It is clear from the previous section that depending on the number of feedback preferences received from a given user, the sampling process may actually be quite time consuming. Thus, it is desirable to avoid generating samples from scratch whenever new feedback is received. In other words, it is desirable to *maintain* previously generated samples against new incoming feedback.

Given a probability distribution $\mathbf{P}_w$ of w, a set S_ρ of preferences, and a sample pool S, instead of regenerating all samples, we can simply replace samples which violate the new feedback, and retain samples which do not violate any new feedback. This approach works since the probability of each valid w always follows $\mathbf{P}_w$, regardless of the newly received feedback.

A simple idea for replacing invalid samples in the pool is to scan through all samples in the pool one by one, and check whether each satisfies the new feedback. This simple approach will be effective if many samples might violate the new feedback received. However, the performance will be very poor if hardly any samples from S actually violate the newly received feedback.

Note that, as discussed in the previous section, for feedback $\rho := p_1 \succ p_2$, w violates ρ if $w \cdot (p_2 - p_1) > 0$. Thus finding those $w \in S$ which violate ρ is the same as finding all weight vectors which have a projected value on $p_2 - p_1$ larger than 0. This problem can be solved by leveraging the classic threshold algorithm (TA) framework [13], by iteratively enumerating the largest w from S w.r.t. the query vector $p_2 - p_1$ until the maximum possible score of any unseen w is less than or equal to 0. Obviously this TA-based algorithm is very efficient when not many samples violate the new preference. However, for cases where most samples violate the new feedback, the cost of the TA-based algorithm may be much higher than the naïve algorithm of checking every sample in the pool for possible preference violation.

Algorithm 1: RejectedSampleCheck(S, $\rho = p_1 \succ p_2$)

```
1  Q ← An empty set for rejected sample w's;
2  L^w ← Lists of samples sorted based on feature values;
3  while true do
4      l^w ← Access lists in L^w in round-robin fashion;
5      w ← getNext(l^w);
6      τ ← boundary value vector from L^w;
7      if w · (p2 − p1) > 0 then Add w to Q;
8      if τ · (p2 − p1) ≤ 0 then Break;
9      if C_processed + C_remain ≥ (1 + γ)|S| then
10         Scan and check each remaining w in l^w, Break;
11 return Q;
```

Motivated by this, we propose a hybrid approach shown in Algorithm 1. We organize the samples into m lists $\mathcal{L}^w = l_1^w, \ldots, l_m^w$, where each list l_i^w is a total ordering of items based on the values of the corresponding feature f_i. Given new feedback ρ, we start with the TA-based algorithm, and if the current number $\mathcal{C}_{processed}$ of items processed plus the number $\mathcal{C}_{remain}$ of remaining items in the current list is larger than or equal to $(1+\gamma)$ of the total number of items, we stop the TA process, and instead scan through the remainder of the current list, checking the validity of each sample within this list. Here, γ is a parameter which can be tuned based on the actual performance, with smaller γ leading to performance closer to the simple scanning algorithm, and larger γ leading to performance closer to the pure TA-based algorithm.

4. SEARCH FOR BEST PACKAGES

If we have a top-k package solver which can produce the top-k packages for a given weight vector w, then given a set S of sample weight vectors, we can find the overall top-k packages under different ranking semantics as follows.

For EXP, we need to estimate the expected utility of packages and return the top-k packages w.r.t. the estimates. Given all the top-k package results obtained from the samples $w \in S$, we maintain the sum of the utility values for each package appearing in the results. Then the sample utility mean of each package is simply the utility sum divided by

the number of times the package appears in a result. Note that we only need to consider those packages which appear in the top-k package list w.r.t. at least one sample w.

For TKP, we just maintain a counter for each package which appears in the result set; the k packages appearing most frequently in this set will be the result under TKP.

For MPO, instead of maintaining statistics for each package that appears in the result set, we maintain a counter for each top-k package list. The final top-k package list under MPO is the one with the largest counter value.

Thus a key step in finding the top-k packages for a set S of sample weight vectors is to find the top-k packages for a specific w, which we address next.

Given a set T of items and a fixed w for the utility function, the problem of finding the k best *items* w.r.t. w can be done using any standard top-k query processing technique [13]. However, because we are dealing with subsets of items, the problem becomes challenging since a naïve solution which first enumerates all possible packages, and then uses a top-k query processing algorithm to find the best k packages would be prohibitively expensive. Below, we discuss how classical top-k query processing algorithms can be adapted to finding the top-k *packages*, given a fixed w.

Following recent research on top-k item query processing [13], one intuition is that for a top-k package p, the likelihood of having a high utility item in p is often higher than the likelihood of having a low utility item in p. Thus by accessing items in their descending utility order, we could potentially locate the top-k packages by accessing only a small number of items. To facilitate efficient processing over different weight vectors, we order items based on their utility w.r.t. each individual feature. We denote the resulting set of sorted lists by $\mathcal{L}$.

Given a fixed w and utility function U, Algorithm 2 gives the pseudo-code of the overall algorithm framework Top-k-Pkg. As shown, Top-k-Pkg first sorts the underlying items into different lists, where each list is an ordering of the items in T w.r.t. the desirable order on one specific feature according to the utility function (line 2). E.g., consider $U(p) = 0.5\mathsf{avg}_1(p) - 0.5\mathsf{sum}_2(p)$, where $\mathsf{avg}_1(p)$ and $\mathsf{sum}_2(p)$ are the normalized aggregation values of the package w.r.t. the corresponding feature. The algorithm sorts items into two lists, l_1 and l_2, where in l_1, items are sorted in non-increasing order of feature 1, and in l_2, in non-decreasing order of feature 2[2]. As discussed before, the intuition is that by accessing items with better utility values w.r.t. each individual feature in the utility function early, we can potentially quickly find the top-k packages of items.

After constructing the set of lists $\mathcal{L}$, Top-k-Pkg accesses items from lists in $\mathcal{L}$ in a round-robin fashion (lines 4–5). We assume items reside in memory, so their feature values can be retrieved quickly. After accessing each new item t, we can obtain the new boundary value vector τ in which each feature value equals the corresponding feature value of the last accessed item in each list (line 6). So essentially, the feature vector τ corresponds to the maximum possible utility value for an unaccessed item.

Next, we can expand the existing packages in the queue Q by incorporating the new item t (line 7), a process described in Section 4.2. During package expansion, a current lower bound utility threshold η^{lo} can be obtained by looking at the kth best package so far in queue Q, and an upper bound utility threshold η^{up} of any possible package can be obtained by referring to the maximum utility value an unaccessed item can have, a calculation described in Section 4.1. Obviously, if $\eta^{up} \leq \eta^{lo}$, we can safely return the current top-k packages, as no future packages can have higher utility than the current top-k packages (lines 8–9).

Algorithm 2: Top-k-Pkg(U, T, w, k)

```
1 Q ← A priority queue of packages having one item ∅;
2 L ← Lists of items in T sorted according to util. func. U;
3 while true do
4     l ← Access lists in L in round-robin fashion;
5     t ← getNext(l);
6     τ ← boundary value vector from L;
7     (η^lo, η^up) ← expandPackages(U, Q, t, τ));
8     if η^up ≤ η^lo then break ;
9 return top-k packages in Q;
```

4.1 Upper Bound Estimation for Best Package

Given the accessed item information, one requirement in Top-k-Pkg is to estimate the upper bound value a package can have. In this section, we present an algorithm for estimating this upper bound value.

Given a fixed weight vector w, the utility value $U(p)$ of a specific package p can be calculated as $p \cdot w$, so it depends only on items within the package p. Given the fact that items in each list of $\mathcal{L}$ are ordered in non-increasing utility of the corresponding feature, the maximal marginal utility value of an unseen item is obviously bounded by the imaginary item with feature vector τ.

Given a utility function U, we say that U is *set-monotone* if for any packages p, p' of items, we have $U(p \cup p') \geq U(p)$. E.g., $U(p) = 0.5\mathsf{sum}_1(s) - 0.5\mathsf{min}_2(s)$ is set-monotone. Clearly, if U is set-monotone, the maximum utility of a package p can be achieved by packing as many items with feature vector τ (were they to exist) as possible into p. On the other hand, if U is not set-monotone, e.g., when some aggregate feature values in U are based on avg, we can show that the upper bound value of p in this case is given by packing as many items with feature vector τ into p as possible, as long as the marginal utility gain of this addition is positive.

LEMMA 3. *Given a package p, a utility function U with fixed w, and a sequence of items $t_1, \ldots, t_m$ such that every feature value of t_i is no worse than that of t_{i+1}, then $U(p \cup \{t_1, \ldots, t_i\}) - U(p \cup \{t_1, \ldots, t_{i-1}\}) \geq U(p \cup \{t_1, \ldots, t_{i+1}\}) - U(p \cup \{t_1, \ldots, t_i\})$, $1 < i < m$.*

PROOF. The result follows from that fact that every feature value of t_i is no worse than that of t_{i+1} w.r.t. U. □

Algorithm 3 shows the function upper-exp, which returns an estimate for the upper bound value, where ϕ is the maximum allowed package size.

THEOREM 3. *The upper bound computed by Algorithm 3 is correct.*

PROOF. The result obviously follows for the case where U is set-monotone. When U is not set-monotone, assume to the contrary that there exists a package $p'' \supset p$, such that p'' is returned by upper-exp, $p'' \neq p'$ and $U(p'') > U(p')$. Since τ bounds the maximum value on every feature of an unseen item, $\forall t \in p'' \backslash p$, τ upper bounds t on every feature. Thus according to Lemma 3, we could replace every $t \in p'' \backslash p$

[2]A sorted list can be accessed both forwards and backwards, so there is no need to maintain two separate lists when both non-increasing and non-decreasing orders are required on the *same* feature.

with τ without sacrificing the utility of p''. So the maximum value achieved by padding p with τ is as good as p'', or $U(p') \geq U(p'')$, which contradicts the assumption. $\square$

Algorithm 3: upper-exp(p, U, τ, ϕ)

1 $p' \leftarrow p$;
2 **if** U *is set-monotone* **then**
3   **for** $i \in [1, \phi - |p|]$ **do** $p' \leftarrow p' \cup \{\tau\}$;
4   **return** $U(p')$
5 **else**
6   **for** $i \in [1, \phi - |p|]$ **do**
7   **if** $U(p' \cup \{\tau\}) - U(p') > 0$ **then** $p' \leftarrow p' \cup \{\tau\}$;
8   **else return** $U(p')$;
9   **return** $U(p')$

4.2 Package Expansion

Consider the problem of expanding the set of packages in queue Q on accessing a new item t. A naïve way of expanding packages would be to add t to every possible package in Q as long as the resulting package satisfies the package size budget, inserting the new packages into Q. This strategy, equivalent to enumerating all possible combinations of accessed items, while correct, is highly inefficient.

Given a package p, one intuitive optimization is that if incorporating any unaccessed item cannot improve the value of p, we do not need to consider p in the expansion phase. E.g., let $U(p) = 0.5\mathsf{avg}_1(p) + 0.5\mathsf{min}_2(p)$, with $p = (0.5, 0.5)$ and $\tau = (0.4, 0.4)$. Clearly, any unaccessed item in $\mathcal{L}$ will have a utility worse than or equal to that of τ, so there is no need to consider p for expansion in the future.

To incorporate this optimization, we split the priority queue Q in Top-k-Pkg into two sub-queues Q_+ and Q_-. Queue Q_+ stores packages which can be further expanded (while improving utility), while Q_- stores packages which cannot be further expanded (while improving utility) and so can be pruned from the expansion phase. In Algorithm 4 for the expansion phase, we only need to iterate through packages in Q_+ (lines 2–12), and for each package $p \in Q_+$, we test whether p can be improved by incorporating the new item t (line 3). If true, we generate a new package p', and insert it into the appropriate sub-queue based on whether it can be further improved by an unaccessed item or not (lines 5–8). If false, we can check whether the current p can be further improved by referring to the updated τ, and p will be moved to Q_- if it cannot be improved (lines 9–11).

Algorithm 4: expandPackages(U, Q, t, τ)

1 $(\eta^{lo}, \eta^{up}) \leftarrow$ lower/upper bound value;
2 **foreach** $p \in Q_+$ **do**
3   **if** $U(p \cup \{t\}) > U(p)$ **then**
4         $p' \leftarrow p \cup \{t\}$;
5   **if** $U(p' \cup \{\tau\}) > U(p'))$ **then**
6             $Q_+ \leftarrow Q_+ \cup \{p'\}$;
7             $\eta^{up} \leftarrow \max(\eta^{up}$, upper-exp($p'$, U, τ, ϕ));
8   **else** $Q_- \leftarrow Q_- \cup \{p'\}$;
9   **if** $U(p \cup \{\tau\}) > U(p)$ **then**
10         $\eta^{up} \leftarrow \max(\eta^{up}$, upper-exp($p$, U, τ, ϕ));
11   **else** $Q_+ \leftarrow Q_+ - \{p\}$; $Q_- \leftarrow Q_- \cup \{p\}$;
12 $\eta^{lo} \leftarrow U(Q_-[k])$ or 0 if fewer than k packages in Q_-;
13 **return** (η^{lo}, η^{up})

5. EXPERIMENTAL EVALUATION

In this section, we study the performance of various algorithms proposed in this paper based on one real dataset of NBA statistics and four synthetic datasets. The goals of our experiments are to study: (i) the performance of various sampling techniques w.r.t. our package recommendation problem; (ii) the effectiveness of the proposed pruning process; (iii) the performance of various maintenance algorithms as the system receives new feedback. We implemented all the algorithms in Python, and all experiments were run on a Linux machine with a 4 Core Intel Xeon CPU, OpenSuSE 12.1, and Python 2.7.2.

The NBA dataset is collected from the Basketball Statistics website [3], which contains the career statistics of NBA players until 2009. The dataset has 3705 NBA players and we randomly selected 10 (out of 17) features to be used in our experiments. The synthetic datasets are generated by adapting the benchmark generator in [4]. The *uniform* (UNI) dataset and the *powerlaw* (PWR) dataset are generated by considering each feature independently. For UNI, feature values are sampled from a uniform distribution, while for PWR, feature values are sampled from a power law distribution with $\alpha = 2.5$ and normalized into the range $[0, 1]$. In the *correlated* (COR) synthetic dataset, values from different features are correlated with each other, while in the *anti-correlated* (ANT) synthetic dataset, values from different features are anti-correlated with each other. Each synthetic dataset has 10 features and has 100,000 tuples.

5.1 Comparing Sampling Methods

In Figure 4, we show an example of how different sampling methods generate 100 valid 2-dimensional sample w parameters given 5000 packages and 2 randomly generated preferences. As discussed previously, each preference $\rho := p_1 \succ p_2$ defines a linear hyperplane. A sample w satisfies ρ iff $w \cdot (p_1 - p_2) \geq 0$, or w is above the corresponding hyperplane. In Figure 4 (a), given the set of valid sample w's (black dots) and the set of invalid sample w's (red crosses), we can infer the two linear hyperplanes which correspond to the two given preferences and bound valid sample w's to the bottom. It is clear from the figure that unless these two preferences are way "above" the center of $\mathbf{P}_w$, many sample w's from $\mathbf{P}_w$ will be invalid. Thus using rejection sampling, many samples will be wasted and we need to spend considerable time checking whether each sample w satisfies all preference constraints received.

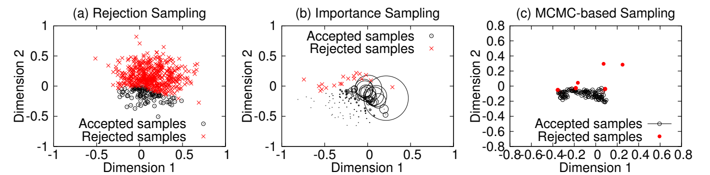

Figure 4: Example of various sampling algorithms.

On the other hand, the two feedback-aware sampling strategies will generate far fewer invalid samples. E.g., in Figure 4 (b), the importance sampling technique samples from a proposal distribution which is more to the center of the valid region, so samples generated are more likely to satisfy all constraints. Notice, each sample w is also associated with a weight, which is captured by the size of the dot/circle in the figure. The higher the probability w has under the original distribution, and the lower the probability w has under the proposal distribution, the larger the weight of w.

[3]http://www.databasebasketball.com

MCMC-based sampling first needs to find one random valid sample w. During this process we leverage the simple rejection sampling, thus these rejected samples (denoted as isolated red crosses in Figure 4 (c)) will not be part of the random walk process in MCMC. Then from this valid sample w, we initiate a random walk from the neighborhood of w, which follows the original distribution of $\mathbf{P}_w$ using a Metropolis-Hastings sampler as discussed Section 3.2.2.

5.2 Constraint Checking

As discussed in Section 3.1, no matter which sampling method we use, an important task is to efficiently check whether a sample satisfies all the feedback constraints received from a user. In Figure 5, we show how pruning strategies discussed in Section 3.3 benefit the overall checking performance by varying the number of features, the number of samples, and the number of Gaussians in the mixture distribution while keeping the other variables fixed at a default value. We set the default value for the number of randomly generated preferences to 10000, the number of packages to 5000, the number of Gaussians to 1, the number of features to 5, and the number of samples to 1000. As can be seen from this figure, when we vary one parameter while fixing other parameters at their default values, the pruning strategy can robustly generate at least a 10% improvement. Results under other different default values are similar.

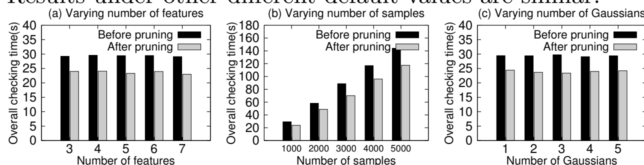

Figure 5: Efficiency of the pruning strategy.

5.3 Overall Time Performance

In this section, we report the overall time performance for package recommendation over different datasets. All time results reported are based on an average of 5 runs.

In Figure 6, we compare overall time performance for generating top-k package recommendations under Rejection Sampling (RS), Importance Sampling (IS), and MCMC-based Sampling (MS). In these experiments, we fixed the ranking semantics to be EXP and varied one of the following two parameters, while fixing the remaining parameters at their default values: (1) Number of valid samples required; (2) Number of features. We also tested varying the ranking semantics, the number of feedback preferences received, and the number of Gaussians in the mixture distribution; the results were very similar, and thus are omitted.

From Figure 6 (a)–(e), with a log-scale for processing time, we observe that the sampling cost for generating valid sample w's mostly outweighs or at least is comparable to the cost for generating top-k packages, as usually the top-k packages can be found by just checking the first few high utility items. Also the rejection sampling cost is usually much higher than that of the other two feedback-aware sampling approaches.

As can be seen from Figure 6 (f)–(j), importance sampling is excluded from high dimensional experiments because finding the center of a high-dimensional polytope is computationally intractable [11]. Even the simple grid-based algorithm discussed in Section 3.2.1 is exponential w.r.t. the dimensionality. When the dimensionality is over 5, the time to find the center will quickly exceed the time for rejection sampling. With dimensionality 6, the algorithm cannot finish within 30 minutes, whereas simple rejection sampling takes only a few seconds. As can be seen from Figure 6 (f)–(j), MCMC-based sampling scales well w.r.t. dimensionality.

5.4 Sample Quality

To measure the quality of different sampling methods, we compare the top-5 package list generated w.r.t. different ranking semantics and different sampling methods. In our experiments, we set the number of samples to 5000,[4] the number of feedback preferences received to 1000, the number of features to 4, and the number of Gaussians in the mixture distribution to 2. Results under different settings are similar and thus excluded. Each package is associated with a unique and random package id.

We have observed that on both the synthetic dataset and the NBA dataset, given enough samples, the top package results from different sampling methods typically tend to become very similar. And the top-k package results under different ranking semantics also tend to be similar to each other. The reason is that although different ranking semantics may potentially result in different top package lists, they can be correlated with each other. E.g., as in our example, if one list of top packages dominates the results given a set of samples, TKP and MPO may tend to give very similar results. This is because picking the same list of top packages guarantees that packages in this list may also appear most frequently among all top packages. EXP may not be affected by this as it is determined by the expected utility of the package, so a package appearing frequently may not necessarily have high expected utility value.

5.5 Sample Maintenance

As discussed in Section 3.4, upon receiving new feedback, a naïve method of scanning through previous samples to determine which samples need to be replaced might be costly if the number of rejected samples is low, whereas a top-k algorithm might help by quickly scanning through the preprocessed sample lists, and determining whether all samples satisfy the constraints. However, this algorithm suffers from a substantial overhead when the number of rejected samples is large. Thus we propose a hybrid method which starts following the top-k based approach, then falls back to the default naïve method if the top-k process cannot stop early.

To assess the actual performance of these three algorithms, we consider in the following experiment a setting where the number of previously generated samples is set to 10000 (results using other values are similar and are omitted), every other parameter is fixed at a default value similar to previous experiments. We randomly generate sets of 1000 feedback preferences, and then according to the number of samples rejected w.r.t. the feedback, we group the maintenance costs into 7 buckets, where each bucket is associated with a label indicating the maximum number of samples rejected (see Figure 7 (a)). Results are placed in the bucket with the smallest qualifying label. Maintenance cost results are averaged for all cases within the same bucket.

According to Figure 7 (a), the top-k based algorithm is a clear winner when the number of rejected samples is small. As the number of rejected samples grows, the performance of the top-k based algorithms deteriorates, especially the non-hybrid method. But the hybrid method introduces only a

[4]Increasing the number of samples beyond 5000 does not change the top package rankings for different datasets.

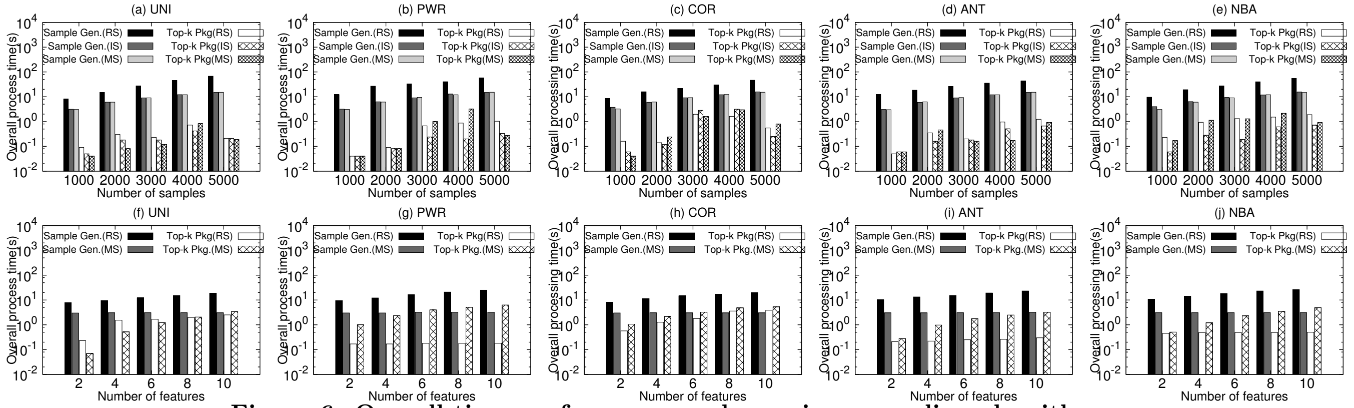

Figure 6: Overall time performance under various sampling algorithms.

small overhead over the naïve algorithm because of the fallback mechanism, and this overhead can be tuned through the parameter γ. In Figure 7 (b), we show how the cost ratio of each of the top-k and hybrid approaches to that of the naïve approach varies with γ. When γ is very small, the average performance of the hybrid method is very similar to the naïve algorithm as the algorithm is forced to check fewer samples. By slightly increasing γ (e.g., to 0.025 as in Figure 7 (b)), the hybrid method can show over 15% improvement compared to the naïve method. When γ increases further, the performance deteriorates as it becomes similar to the non-hybrid method. We note that this property means we could adaptively decrease γ in practice until a reasonable performance gain can be observed.

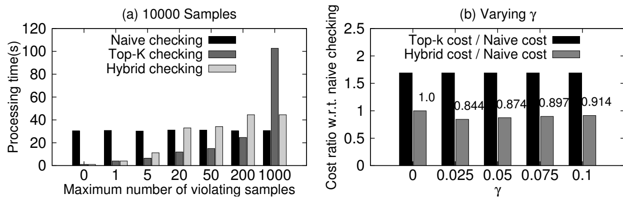

Figure 7: Experiments on sample maintenance.

5.6 Elicitation Effectiveness Study

To study the effectiveness of the proposed preference elicitation framework, we consider in the following experiment a set of 100 randomly generated ground truth utility functions for the NBA dataset which are not known by the package RS. For each utility function U, we use our proposed algorithm to generate 5 recommended packages along with 5 random packages, and we assume users always pick from these recommendations the one which maximizes U.

In Figure 8, we show for MCMC-based sampling and under EXP, only a few clicks/feedbacks are needed for each query before the system converges to a stable top-k package ranking list. Results for different ranking semantics are similar, so are omitted.

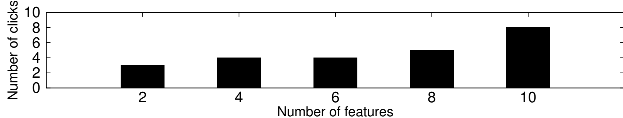

Figure 8: Experiments on elicitation effectiveness.

6. RELATED WORK

Item RS recommend to users personalized lists of single items based on previous transactions of the users in the system. The promise of item RS has been recognized by the database community, resulting in a series of recent research projects such as RecStore [19], FlexRecs [18], aimed at pushing item RS inside the database engine. However, as pointed out in an influential survey [1], item RS suffer from limited flexibility and cannot be leveraged to recommend packages.

There has been much investigation in the database community into handling of preferences over items, e.g., general preference frameworks [16] [10], skyline queries [4, 22], and top-k queries [13]. However only recently have researchers started considering preference handling for sets of items. This calls for exploring a much larger candidate space, and usually has an aggregation-based feature space, which further complicates the underlying problem. Initial work by AI researchers such as [7] usually focuses on the formal aspects of this problem, e.g., expressiveness of the preference language, but the proposed model is often not practical. In [27], we model preferences on packages using hard constraints and a fixed score function, thus turning the problem of finding the top preferred sets into an optimization problem. However, hard constraints may be too rigid for practical purposes, since users are often flexible or unsure when considering budgets and quality, and may be willing to trade cost for better quality results. In [29] and [20], the authors study an alternative approach of finding skyline packages of fixed cardinality. A severe drawback of this approach is that the number of skyline packages can be prohibitive.

In [30], the authors propose an interactive way to elicit preferred items. The paper does not consider preferences for packages, and more importantly, they assume that the weight parameters of the underlying utility function follow a uniform distribution, and do not discuss how user feedback can be leveraged to update the utility function.

In [22] and [14], the authors have a similar setting of inferring preferences given a set of item comparisons. Both papers focus on inferring a most desirable order directly from given comparisons; whereas in our work, we took a Bayesian-based approach by modeling the parameters of the utility function using a distribution. The most desirable order of packages depends on the uncertain utility function following different ranking semantics. Feedback received only affects the posterior of the parameter distribution.

In [23], the authors consider an interactive way of ranking travel packages. However, the user feedback model in [23] is defined in such a way that for each iteration, the user is asked to rank a set of items instead of a set of packages. Also unlike our framework in which the elicitation of preferences is implicit, [23] requires several iterations of explicit preference elicitation before the system shows the user any recommended package.

Finally, in [28], the authors study the problem of constrained sampling. However, the underlying distribution they consider is discrete, whereas we use Gaussian mixtures. Also the technique proposed there depends on the particular form of the constraint, e.g., $\mathsf{sum}(p) \leq 10$, which is quite different from the preference constraints considered here.

7. DISCUSSION AND CONCLUSION

Discussion: A user's online interaction can be noisy. E.g., a user may accidentally click on a package or may change her mind after clicking. A popular method for modeling this problem is to assume that every feedback received has a probability ψ of being "correct" [6]. We can use this noise model in our framework by assuming that every feedback is independent. Then instead of rejecting a sample w whenever it violates some feedback preference, we condition its rejection using the probability that at least one violated feedback preference is correct, i.e., $1-(1-\psi)^x$, where x is the number of feedbacks w violates. This can easily be incorporated into both importance and MCMC-based sampling.

As discussed in [29], users may sometimes specify predicates on the schema of a desired package, e.g., when buying a set of books, at least two should be novels. We can handle such predicates in the top package generation process discussed in Section 4. The idea is that when generating a new candidate package, we evaluate the predicates and retain the package only if it satisfies the specified predicates.

Conclusion: In this paper, we propose a package RS which uses a Bayesian uncertainty-based preference elicitation framework. The core of the proposed system is a linear utility function which captures a user's personalized preferences over packages. Unlike most existing work which assumes that this utility function is given, we learn weight parameters of this function through implicit user feedback in the form of clicks on suggested packages. We propose several constrained sampling-based methods which treat user's feedback as constraints and avoid calculating posterior distributions of the weight parameters explicitly. Each sampling algorithm can be leveraged to efficiently find the top-k packages under different ranking semantics, and a user's feedback over these recommended packages can be used to update the system's knowledge of the corresponding utility function. As future work, we plan to investigate how Algorithm Top-k-Pkg can be further optimized using *domination-based pruning* [29, 20]. The idea is that by pruning away candidate packages which are not promising, we can further reduce the queue Q, which is iteratively searched in the expansion phase. However, pruning strategies also come with an overhead, so a systematic cost-based study of different pruning strategies under our proposed model would be interesting.

8. REFERENCES

[1] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *TKDE*, 17(6):734–749, 2005.

[2] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *J.Comp.*, 1(2):131–137, 1972.

[3] C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[4] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[5] N. Botkin and V. Turova-Botkina. An algorithm for finding the chebyshev center of a convex polyhedron. *Applied Mathematics and Optimization*, 29(2):211–222, 1994.

[6] C. Boutilier. A pomdp formulation of preference elicitation problems. In *AAAI*, pages 239–246, 2002.

[7] G. Brewka, D. Kossmann, and K. Stocker. Representing preferences among sets. In *AAAI*, 2010.

[8] Z. Cao, T. Qin, T.-Y. Liu, M.-F. Tsai, and H. Li. Learning to rank: from pairwise approach to listwise approach. In *ICML*, pages 129–136, 2007.

[9] U. Chajewska, D. Koller, and R. Parr. Making rational decisions using adaptive utility elicitation. In *AAAI*, pages 363–369, 2000.

[10] J. Chomicki. Preference formulas in relational queries. *ACM TODS*, 28(4):427–466, 2003.

[11] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 2000.

[12] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

[13] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[14] B. Jiang, J. Pei, X. Lin, D. W. Cheung, and J. Han. Mining preferences from superior and inferior examples. In *KDD*, pages 390–398, 2008.

[15] R. L. Keeney and H. Raiffa. *Decisions with Multiple Objectives: Decisions with Preferences and Value Tradeoffs*. Cambridge University Press, 2003.

[16] W. Kießling. Foundations of preferences in database systems. In *VLDB*, pages 311–322, 2002.

[17] A. Kong, J. S. Liu, and W. H. Wong. Sequential imputations and bayesian missing data problems. *JASA*, 89(425):278–288, 1994.

[18] G. Koutrika, B. Bercovitz, and H. Garcia-Molina. FlexRecs: expressing and combining flexible recommendations. In *SIGMOD*, pages 745–758, 2009.

[19] J. J. Levandoski, M. Sarwat, M. F. Mokbel, and M. D. Ekstrand. Recstore: an extensible and adaptive framework for online recommender queries inside the database engine. In *EDBT*, pages 86–96, 2012.

[20] C. Li, N. Zhang, N. Hassan, S. Rajasekaran, and G. Das. On skyline groups. In *CIKM*, 2012.

[21] G. Linden, S. Hanks, and N. Lesh. Interactive assessment of user preference models: The automated travel assistant. In *UM*, pages 67–78, 1997.

[22] D. Mindolin and J. Chomicki. Discovering relative importance of skyline attributes. *PVLDB*, 2(1):610–621, 2009.

[23] S. B. Roy, G. Das, S. Amer-Yahia, and C. Yu. Interactive itinerary planning. In *ICDE*, 2011.

[24] M. A. Soliman, I. F. Ilyas, D. Martinenghi, and M. Tagliasacchi. Ranking with uncertain scoring functions: Semantics and sensitivity measures. In *SIGMOD*, pages 805–816, 2011.

[25] K. Stefanidis, G. Koutrika, and E. Pitoura. A survey on representation, composition and application of preferences in database systems. *ACM TODS*, 36(3):19, 2011.

[26] O. Toubia, J. R. Hauser, and D. I. Simester. Polyhedral methods for adaptive choice-based conjoint analysis. *Journal of Marketing Research*, 41(1):116–131, 2004.

[27] M. Xie, L. V. S. Lakshmanan, and P. T. Wood. Breaking out of the box of recommendations: From items to packages. In *RecSys*, pages 151–158. ACM, 2010.

[28] M. Yang, H. Wang, H. Chen, and W.-S. Ku. Querying uncertain data with aggregate constraints. In *SIGMOD*, pages 817–828, 2011.

[29] X. Zhang and J. Chomicki. Preference queries over sets. In *ICDE*, pages 1019–1030, 2011.

[30] F. Zhao, G. Das, K.-L. Tan, and A. K. H. Tung. Call to order: A hierarchical browsing approach to eliciting users preference. In *SIGMOD*, pages 27–38, 2010.