Finding Best Tuple via Error-prone User Interaction

Qixu Chen, Raymond Chi-Wing Wong
The Hong Kong University of Science and Technology
qchenax@connect.ust.hk, raywong@cse.ust.hk

***Abstract*—In the literature of the database community, there are a lot of studies about finding a utility function from a user (representing the user's preference), via interaction with the user by asking a number of questions each requiring him/her to compare 2 points for choosing a more preferred point, in order to find the best tuple in the database containing a lot of tuples. In the real world, the user may make mistakes (carelessly), which means that s/he may answer some of the questions *wrongly*. Unfortunately, existing interaction algorithms may find the *undesirable* point based on the *wrongly learnt* utility function because they assume that all answers from the user are 100% correct. In particular, even if the user answers only 1 wrong answer, the output of the existing algorithms may be far away from the users' real need. Motivated by this, in this paper, we propose a new problem of finding the most interesting point via interaction which is robust to possible mistakes made by a user. Besides, we propose (1) an algorithm that asks an asymptotically optimal number of questions when the dataset contains 2 dimensions and (2) two algorithms with provable performance guarantee when the dataset contains d dimensions where $d \geq 2$. Experiments on real and synthetic datasets show that our algorithms outperform the existing ones with a higher accuracy with only a small number of questions asked.**

I. Introduction

A database system may contain millions of points (or tuples). In order to help the user to find his/her interesting point, we need queries to obtain a representative set of points consisting of his/her interesting point. Such queries can be considered as *multi-criteria decision making* problems and they can be applied in various domains, including house buying, car purchase and job search. For example, in a car purchasing database where each car is described by some attributes, Alice wants to find a car with a low gasoline consumption and a high speed, and is as cheap as possible. Here, gasoline consumption, speed and price are some attributes that Alice would consider when buying a car.

There are two popular types of traditional multi-criteria decision making queries, namely *top-k* and *skyline* [6]. The top-k query measures the *utility* of tuples based on a *utility function* provided by the user. A high utility indicates that the corresponding tuple is more preferred. The query returns k tuples with the highest utility in the dataset. Top-k requires the knowledge of the user's exact utility function. On the other hand, the skyline query does not need this information and uses a *"dominant"* concept. Specifically, a tuple p is said to *dominate* a tuple q if p is not worse than q in any attribute and is better than q in at least one attribute. Intuitively, p is better than q w.r.t. all *monotonic* utility functions. The skyline query returns the set of tuples that are not dominated by any tuple in the dataset. Unfortunately, the size of the returned set is uncontrollable and could be as large as the database size.

Motivated by this, a novel interactive framework [28, 37, 40] was proposed to overcome the disadvantages of both the top-k query (requiring a given utility function) and the skyline query (returning an output with an uncontrollable size). Intuitively, it asks the user a number of *rounds* of simple questions and returns the most interesting tuple to the user. The interactive system not only does not require the user to provide an exact utility function, but also can control the size of the returned set (i.e., the only one tuple returned). Therefore, it does not have the limitation of both top-k and skyline queries. A widely applied form of question [28, 37, 40] is to display 2 points in each round, and the user is asked to select the preferred point. Consider the car purchasing scenario. The interactive framework simulates a sales assistant that asks Alice to indicate her preference among several pairs of cars, and make recommendations based on the answers of Alice.

However, in real world, when answering questions, people make occasional mistakes due to various reasons. For example, in a simple task of selecting the correct switch among two given switches that are dissimilar in shape, a human knowing which switch is correct can still select the wrong one with probability 0.1% [21]. For operations that require some care, the error probability can range from 1% to 3%, which may even increase under high mental stress. Simple calculation shows that if the system interacts with the user for 10 rounds (i.e., asks the user a sequence of 10 questions), a user has 10%-30% chance to make at least one mistake. For example, Alice may indicate her preference to a cheap car with high gas consumption among some similar cars due to a careless mistake, even though she intends to buy a low-consumption car. Note that in the real world, a meticulous sales assistant will notice the inconsistency and check with Alice.

It is worth mentioning that making a "small" mistake can lead to unforgettable and unchangeable consequences. Let us give two real cases. The first case is about the selection of the tertiary school, one of the critical milestones of one's life. In 2020, an 18-year-old student in Guangdong province in Mainland China who obtained a top-tier score from the National College Entrance Examination in China (also called gaokao) was admitted to a low-ranked college with a similar name to his target university due to his mistake of choosing a wrong school in the tertiary school selection system [38]. The second case is about a huge financial loss due to a well-known "fat finger error" (an error made by the operator in the trading system when making a wrong deal in the trading system by

mis-clicking or pressing a wrong key). In 2018, Samsung Securities made a wrong transaction worth 100 billion dollars due to a fat finger error, which could incur a loss of 428 million dollars, 12.17% of the company's market capitalization [1].

Motivated by this, in this paper, we propose a new problem called the *interactive best point retrieval problem* considering error-prone user input, which is more realistic. Roughly speaking, our problem is to find the best point in a dataset D for a user with an *unknown* utility vector u, in the scenario that the user is "imperfect" and makes a random error with probability at most θ for each question requiring a user to select one preferred point among 2 points displayed, where θ is called an *error rate* and is a user parameter. In other words, the user chooses the preferred point with probability at least $1 - \theta$ and chooses the other point with probability at most θ. Note that θ can be obtained from some channels like user behavior studies [21]. In our user study, θ is found to be 4.5%.

Although most (if not all) existing interaction algorithms [28, 37, 40] do not consider user errors, unfortunately, their adaptions may find the *undesirable* point based on the *wrongly learnt* utility function since they assume that the user never makes mistake. In our experiment, these algorithms return incorrect results. For example, on a dataset with 1M points, the accuracies of all closely related adapted algorithms *UtilityApprox* [28], *UH-Simplex* [40] and *HD-PI* [37] are at most 74% only.

Furthermore, all adapted versions of existing interactive algorithms considering user errors [19, 31] for this problem do not perform well. In our experiment, the accuracies of the adapted versions of *Active-Ranking* [19] and *Preference-Learning* [31] on a dataset with size 1M are at most 67% only, which is not acceptable.

Contributions. We summarize our contributions as follows. Firstly, we are the first to propose the best point retrieval problem considering random interaction errors during user interaction under interactive multi-criteria decision making problems. Secondly, we prove a lower bound on the expected number of questions needed to determine the best point with a desired confidence threshold. Thirdly, we propose (1) an algorithm with an asymptotically optimal number of questions asked when the dataset contains two dimensions and (2) two solutions with provable guarantee in terms of both the number of questions asked and the confidence on finding the best point when the dataset contains d dimensions where $d \geq 2$. Fourthly, we conducted comprehensive experiments to demonstrate the superiority of the proposed methods. The results show that our algorithms maintain a high accuracy (e.g., nearly to 100% in most experiments) in finding the best point using only a small number of questions, but existing approaches either ask too many questions (e.g., twice as many as ours), or are much inaccurate (e.g., more than 10% less accuracy than ours).

Organizations. The rest of this paper is organized as follows: Section II gives our problem definition. Section III shows the related work. We introduce the algorithm for the dataset containing two dimensions in Section IV and two algorithms for the dataset containing at least two dimensions in Section V. In Section VI, we present the experimental results. Section VII concludes the paper.

II. Problem Definition

In this section, we provide a formal definition to our problem. We first introduce some basic terminologies. Then, we formally define the random user error setting. Next, We study the lower bound of the number of questions to return the best point with a desired confidence. The input of our problem is a dataset D in a d-dimensional space. Note that each tuple in D could be described by more than d attributes, but the user is interested in exactly d of them.

Terminologies. In this paper, we use the word "tuple" and "point" interchangeably. We denote the i-th dimensional value of a point $p \in D$ by $p[i]$ where $i \in [1, d]$. Without loss of generality, the value of each dimension is normalized in range $[0, 1]$ and for each $i \in [1, d]$, there exist at least one point $p \in D$ such that $p[i] = 1$. We assume that a larger value in each dimension is more preferable to the user. If a smaller value is preferred for an attribute (e.g., price), we can modify the dimension by subtracting each value from 1 so that it satisfies the above assumption. Consider the 2-dimensional example in Table I. We have a database $D = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$ and we are interested in attribute X_1 and X_2.

As widely applied in [26, 28, 31, 37, 39, 40], the user's preference is modeled as an unknown *linear utility function*. Specifically, we model the utility function $f : \mathbb{R}^d_+ \to \mathbb{R}_+$ as a linear function $f(p) = u \cdot p$, where u is a non-negative d-dimensional real vector and $u[i]$ measures the importance of the i-th attribute. We call $f(p)$ the *utility* of p w.r.t. f and u is called the *utility vector*. In the rest of this paper, we also refer f by its utility vector u. We are interested in finding the point with the highest utility, which is the point $p_h = \arg\max_{p \in D} u \cdot p$. Note that scaling u does not change the rank of points in D and thus, does not change the best point. Therefore, without loss of generality, we assume that $\sum_{i=1}^{d} u[i] = 1$. Although we target at retrieving the best point, the algorithms developed in this paper can be easily adapted to return the top-k points w.r.t. u. We include the details in [9] due to space constraint.

Handling Random Errors. The system interacts with a user for several *rounds*, until a stopping condition is satisfied. The user is asked 1 question in each round and we use the term "question" and "round" interchangeably in the rest of this paper. We adopt a popular strategy of asking questions in the literature [19, 22, 24, 31] that in each round, the system displays a pair of points, namely p_i and p_j, to the user, and the user returns the preferred point between these 2 points. We primarily focus on pairwise comparison in this paper, but the algorithms we developed can be adapted to a variety of question types (e.g., choosing one out of multiple points). The details can be found in [9]. Instead of assuming the user always makes correct choices, the user makes a random error with probability at most θ in each round. Specifically, let p^* denote the point with a higher utility in two points p_i and p_j, the user

p	X_1	X_2	$f(p)(u=(0.3,0.7))$
p_1	0.2	1	0.76
p_2	0.5	0.9	0.78
p_3	0.8	0.7	0.73
p_4	0.9	0.6	0.69
p_5	1	0.2	0.44
p_6	0.6	0.8	0.74
p_7	0.8	0.5	0.59

TABLE I: Dataset and utility

D	the input dataset
d	the dimensionality of D
N	size of raw dataset
n	size of $conv(D)$
u	the utility vector
θ	the upper bound of user error rate
δ	the failure probability
k	parameter of the checking subroutine
p_i	tuple in the dataset
P_i	best partition of point p_i
$h_{i,j}$	the hyperplane related to p_i and p_j

TABLE II: Commonly used symbols

$h_{i,j}$	$h^+_{i,j}$	$h^-_{i,j}$	intersects
$h_{1,2}$	$\{P_1\}$	$\{P_2, P_3\}$	$\{P_4, P_5\}$
$h_{1,4}$	$\{P_1, P_3, P_5\}$	$\{P_4\}$	$\{P_2\}$
$h_{2,4}$	$\{P_2, P_3, P_5\}$	$\{P_4\}$	$\{P_1\}$
$h_{2,5}$	$\{P_1, P_2, P_4\}$	$\{P_3, P_5\}$	$\{\}$
$h_{3,5}$	$\{P_3\}$	$\{P_1, P_2, P_4, P_5\}$	$\{\}$

TABLE III: Table L

$h_{i,j}$	$h^+_{i,j}$	$h^-_{i,j}$	intersects
$h_{1,2}$	$\{P_1\}$	$\{P_2\}$	$\{P_4\}$
$h_{1,4}$	$\{P_1\}$	$\{P_4\}$	$\{P_2\}$
$h_{2,4}$	$\{P_2\}$	$\{P_4\}$	$\{P_1\}$

TABLE IV: Table L after selecting $h_{2,5}$

chooses p^* with probability at least $1-\theta$, and, with probability at most θ, s/he selects the other point. Typically, θ should not be too large since it is a careless mistake. Thus, it is natural to assume that θ is smaller than 0.5 as supported by existing statistics about the error rate described in Section I. If θ is greater than 0.5, there is no hope that we could find the best point for this user (because the user already gives more than half of his/her answers wrongly.)

We summarize the frequently used symbols in Table II.

Lower Bound. We are interested in the following problem: Given an input size n, and given that the user makes an error with probability at most θ in each round, how many questions do we expect to ask to obtain the best point? We present the following theorem about the lower bound.

Theorem 1. *For any dimensionality d, given an error rate θ and a confidence parameter δ, there exists a dataset of n points such that any pairwise comparison-based algorithm needs to ask $\Omega(\frac{\theta^2(1-\theta)}{(1-2\theta)^2}(\log n)\log(\frac{\log n}{\delta}))$ rounds on expectation to determine the best point with confidence at least $1-\delta$.*

Proof sketch. We first show that in order to find the best point, $\Omega(\log n)$ questions must be asked and the expected number of errors we must handle is $\mu = \Omega(\theta \log n)$. We then prove that to let the total failure probability be less than δ, at least $k = \Omega(\frac{\theta(1-\theta)}{(1-2\theta)^2}\log(\frac{\log n}{\delta}))$ additional questions must be asked to handle each error. The total number of rounds is therefore $\Omega(\log n + \mu k) = \Omega(\frac{\theta^2(1-\theta)}{(1-2\theta)^2}(\log n)\log(\frac{\log n}{\delta}))$. The complete proofs of theorems and lemmas presented in this paper can be found in [9]. □

III. Related Work

Besides the traditional top-*k* and skyline queries mentioned in Section I, various types of multi-criteria decision making queries were proposed. [3, 35] propose the similarity query which looks for tuples that are similar to a given query tuple, where the similarity is measured by a given distance function. However, the query tuple and the distance function must be provided in advance [4], which may be an unrealistic assumption in practice. On the other hand, the regret-minimizing related queries [10, 29, 30, 39] returns a set of tuples that minimizes the *regret level* of the user, where the regret level measures how regretful the user will be if s/he only examines the returned points instead of the entire dataset. Although regret minimizing related queries do not require the presence of user's utility function, it is hard to achieve both a small regret level and a small size of returned set. When a small regret level is fixed, the output size is usually large [10, 40]. Some recent studies [11, 26, 27] aim to combine top-*k* and skyline and return personalized results by computing points that are not dominated in a specific region. But, these studies still require some knowledge on user's utility function. If the user cannot provide a good estimation on the utility function, the performance of these algorithms will degenerate.

To overcome the limitations of the above queries, some existing studies [4, 19, 28, 31, 34, 37, 40] proposed to involve user interactions. The form of interaction varies. A form of interaction that is widely adopted in [19, 28, 31, 37, 40] is to ask the user to select the favorite point among a set of displayed points. [28] proposed the interactive regret minimizing query, which aims to lower the regret ratio while keeping the output size small. [40] follows the study on regret minimization and proposed two algorithms, namely *UH-Simplex* and *UH-Random*, that only displays real tuples inside the dataset. [37] proposed interactive algorithms, *HD-PI* and *RH*, that target at searching for one of the top-*k* point in the database. However, all of these algorithms assume that the user never makes mistakes and completely prune some points from further consideration, making it not applicable to adapt them to handle user errors.

Besides asking a user to select one point from a set of points, there are studies focusing on other types of interactions. [41] developed algorithm *Sort-Simplex* which asks the user to give a ranking on the displayed points. [25] asks the user to partition points into *superior* and *inferior* groups to learn his/her preference. [5] proposed the interactive similarity query that learns the distance function and the query tuple via user interaction. However, it requires a user to assign *relevance scores* to hundreds of tuples to locate the query tuple. In a user's perspective, these interactions are too demanding and may affect their willingness to interact with the system. Besides, user errors are not considered in these studies.

The robustness issue is also studied in the Machine Learning (ML) and Information Retrieval (IR) literature [17, 19, 20, 31, 33]. But, one major difference between their work and ours is that most (if not all) of them focus on a given *static* dataset without user interaction involved, but in our case, the data is created *dynamically* during user interaction. It is worth mentioning that for many algorithms in the ML/IR field, the data required to return an accurate result can be more than 10^3 [17, 20], which means that if we directly adapt them to our problem, the user needs to answer thousands of questions. [31] proposed *Preference-Learning* to learn user's preference that

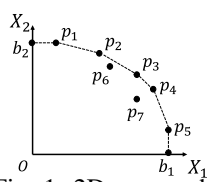


Fig. 1: 2D convex hull

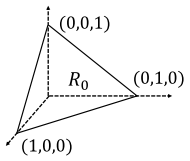


Fig. 2: Utility space in 3D

copes user errors by introducing a slack variant in a linear SVM. [19] proposed *Active-Ranking* and resolves possible conflict using the majority vote. Although these algorithms can be *adapted* to find the best point considering user errors, they are not efficient enough and tend to ask many more questions.

Compared with the existing studies, our work has the following advantages. Firstly, we do not require the user to provide an exact utility function or query tuple, as in the top-*k* query or the similarity query. Secondly, we reduce the user's effort by asking fewer questions and only asking the user to select one point (i.e., the most desirable point) in each question, while some existing algorithms ask too many questions (e.g., [19, 31]) and some other algorithms (e.g., [4, 25, 41]) ask too difficult questions. Finally, we allow the user to make unavoidable errors in interaction, which discards the unrealistic error-free assumption made in some existing studies. Even under the user error setting, we guarantee the retrieval of the best point with a desired confidence level.

IV. Two-dimensional Algorithm

In this section we focus on the case where $d = 2$ and present the *2-dimensional Robust Interactive* (*2RI*) algorithm. We first introduce some important concepts in Section IV-A. Then, in Section IV-B, we show a useful checking scheme that will be used in later algorithms. Finally, we show the details of *2RI* in Section IV-C. The number of rounds used by *2RI* is asymptotically equal to the lower bound in Section II.

A. Preliminaries

In geometry, the *convex hull* of a dataset D, denoted by $CH(D)$, is the smallest convex set containing all points in D [16]. A point $p \in D$ is a *vertex* of $CH(D)$ if $p \notin CH(D/\{p\})$. We use $conv(D)$ to denote the set of *vertices* of $CH(D)$. Let b_i denote the point with the i-th coordinate being 1 and all other coordinates being 0. Also, denote $B = \{b_i | 1 \leq i \leq d\}$ and denote the origin as O. Consider the set $D \cup B \cup \{O\}$. In the remaining sections, when we say $conv(D)$, we mean the set of points that are both in D and in $conv(D \cup B \cup \{O\})$. We assume that there are n points in $conv(D)$, namely $p_1, p_2, \ldots, p_n$, in a clockwise order. Consider the dataset D in Table I and its corresponding $conv(D)$ visualized in Figure 1. In this example, $conv(D) = \{p_1, p_2, p_3, p_4, p_5\}$. In our experiment, $conv(D)$ is found using an existing algorithm called *Quickhull* [2].

One important conclusion is that the best point must be in $conv(D)$ [40], and thus, we need only look at points in $conv(D)$. This is because for any $p \notin conv(D)$ and any utility vector u, there must exist a point $p^* \in conv(D)$ s.t. $u \cdot p \leq u \cdot p^*$. This conclusion can also be applied to any dimensions $d \geq 2$. In the rest of this paper, unless explicitly stated, we will assume that the input to our algorithm is $conv(D)$ and we use n to denote the size of $conv(D)$.

Algorithm 1 Check(p_i, p_j, k)

```
select_p_i ← 0, select_p_j ← 0
while select_p_i < ⌈k/2⌉ and select_p_j < ⌈k/2⌉ do
    display pair (p_i, p_j) to the user
    if p_i is chosen by the user then
        select_p_i ← select_p_i + 1
    else
        select_p_j ← select_p_j + 1
return p_i if (select_p_i ≥ ⌈k/2⌉) and p_j otherwise.
```

B. Checking Subroutine

Before introducing *2RI*, we first introduce a checking subroutine (called *Check*$(\cdot, \cdot, \cdot)$), which will be applied later in our algorithms. Intuitively, since the preference indicated by the user between two points, p_i and p_j, is incorrect with probability at most θ, we need to devise a way of "checking" to increase the confidence level of the results obtained. The details of the subroutine, *Check*$(\cdot, \cdot, \cdot)$, are shown in Algorithm 1. In short, what it does is to check the relation between two points, namely p_i and p_j, for at most k times, where k is a user parameter, and return whether p_i is more preferred to p_j based on the majority vote. k is set to an odd number to break ties. In case that we want to avoid asking repetitive questions involving same points, we could use common techniques in the field of questionnaire reliability (formally called *intra-rater reliability* [18, 32]). One example is that we could re-scale the two points in each question by multiplying with a random number between 0.95 and 1 so that these 2 new points are shown to the user and look different from the 2 original points, resulting in a question which looks different from the original question. Another example is asking correlated questions which look differently using statistical methods.

Corollary 1. *Given a user error rate θ and a desired failure probability α, the checking subroutine fails with probability at most α by setting $k = \Theta(\frac{\theta(1-\theta)}{(1-2\theta)^2} \log(\frac{1}{\alpha}))$.*

Proof sketch. The checking subroutine fails when fewer than half of the k answers are correct. We can approximate the number of correct answers in all k answers by a Gaussian distribution with mean $\mu = (1-\theta)k$ and variance $\sigma^2 = \theta(1-\theta)k$. Let z_α denote the α-quantile of $N(0,1)$ (i.e., $P(X \leq z_\alpha) = \alpha$), by solving $\frac{\frac{k}{2}-\mu}{\sigma} \leq z_\alpha$, we obtain $k \geq \frac{4\theta(1-\theta)}{(1-2\theta)^2} z_\alpha^2$. Since $z_\alpha = \Theta(\sqrt{\log(\frac{1}{\alpha})})$, $k = \Theta(\frac{4\theta(1-\theta)}{(1-2\theta)^2} \log(\frac{1}{\alpha}))$. □

A naive solution of the problem is then to apply this checking subroutine for every question asked to ensure their correctness. However, this solution incurs too many unnecessary questions. We seek ways of using this subroutine as few as possible and only invoke this subroutine when necessary.

C. 2RI

We are now ready to present our 2-d algorithm *2RI*. The input is a list of points $\{p_1, \ldots, p_n\}$, sorted in a clockwise order. For the ease of illustration, we label the points from 1 to n. Intuitively, algorithm *2RI* performs a search on this sorted list by initializing a search range on this sorted list, denoted by $[L, U]$, where L and U are initialized to 1 and n, updating variables L and U to shrink/expand the search range based on the user's interaction and returning the point in the search range as an output when it contains only one element.

Before we describe algorithm *2RI*, we need to describe 3 concepts. The first concept is an observation on the increasing-and-decreasing trend of the utility values over the sorted list.

Observation 1. *Given the utility vector u and n points $p_1, p_2, \ldots, p_n$ in $conv(D)$ ordered in a clockwise direction. Let p_h be the point with the highest utility score, that is, $p_h = \arg\max_{p \in conv(D)} u \cdot p$. Then $\forall 1 \leq i < j \leq h, u \cdot p_i < u \cdot p_j$. Besides, $\forall h \leq i < j \leq n, u \cdot p_i > u \cdot p_j$.*

The second concept is the size of a search range. Given a range $R = [L, U]$, we define the *size* of the range R, denoted as m, to be $m = U - L + 1$.

The third concept is related to the details of an operation to be used in algorithm *2RI*. This operation is the procedure of how we could determine whether the desired point p_h (which is unknown and is to be found) is in the search range $[L', U']$ where $h \in [1, n]$, $L' \in [1, n]$ and $U' \in [1, n]$. Without loss of generality, we show how to decide if the left boundary of the search range is correct, that is, if $L' \leq h$. The case of the right boundary is then symmetric. To test the left boundary, we can simply check if $p_{L'}$ is preferred to $p_{L'-1}$ (If $p_{L'-1}$ does not exist, then the left boundary is trivially correct). According to Observation 1, if $p_{L'}$ is preferred, then we can conclude that $L' \leq h$. Otherwise, the left boundary is wrong and we must adjust L'. In conclusion, checking the left (right) boundary requires to call the checking subroutine *Check*$(p_{L'}, p_{L'-1}, k)$ (*Check*$(p_{U'}, p_{U'+1}, k)$.).

We are ready to describe algorithm *2RI*. Initially, the search range, denoted by $[L, U]$, is initialized such that variable L is set to 1 and variable U is set to n. In the following, variables L and U are updated during the execution of the algorithm which needs interaction from the user. Specifically, we perform the following iterative process until $U = L$ (i.e., there is only one element in the search range). When $U = L$, we return p_L as the output point of this algorithm.

Specifically, the interactive process involves a number of iterations. Each iteration has the following two steps.

- **Step 1 (Search Range Shrinking):** We initialize variables L' and U' to be L and U, respectively. We initialize variable m to be the size of the initial search range (i.e., $U - L + 1$). Due to the increasing-decreasing trend of the utility value as shown in Observation 1, we could perform a binary search on range $R' = [L', U']$ by updating variables L' and U' until the size of the updated search range $R' = [L', U']$ is at most $\lceil 2m\theta \rceil$. The reason why we choose this maximum size as $\lceil 2m\theta \rceil$ can be found in Lemma 1. Here, each step involved in a binary search corresponds to the operation of asking the user to select a more preferred point among the two displayed points where these two points are p_r and p_{r+1} where $r = \lfloor \frac{U'+L'}{2} \rfloor$. Depending on the user's answer, by Observation 1, the search range could be shrunk accordingly. This technique is similar to [28, 40] and details could be found therein. It is worth mentioning that we did not call any checking routine described before in this step.
- **Step 2 (Search Range Verification and Correction):** It performs the checking subroutine to verify if the desired point p_h is still inside range $[L', U']$ (i.e., R'). If yes, the algorithm proceeds to the next iteration with the confirmed range R' by updating variable L to be L' and variable U to be U'. However, if one of L' and U' is not correct, we need to perform some additional checking subroutine and update variables L' and U' accordingly. Note that either the left boundary L' or the right boundary U' is wrong, but not both. Without loss of generality, we assume that L' is wrong (i.e., $L' > h$). In the following, we want to update the search range such that both the left boundary and the right boundary of the updated search range are smaller than L'. At the same time, we also try to keep the size of the updated search range at most $\lceil 2m\theta \rceil$ w.h.p.. There are two cases where the first is a general case and the second is a boundary case.
  - Case (1) (i.e., $L' - \lceil 2m\theta \rceil > L$) In this case, the algorithm performs the checking subroutine again to decide whether p_h is inside range $[L' - \lceil 2m\theta \rceil, L' - 1]$. If this is the case, it updates variable L' to be $L' - \lceil 2m\theta \rceil$ and variable U' to be $L' - 1$, which essentially "shifts" the search range to this new range (i.e., $[L' - \lceil 2m\theta \rceil, L' - 1]$). Note that this new range also has size at most $\lceil 2m\theta \rceil$ and is exactly just on the left-hand-side of the original search range over the sorted list. But, if p_h is still not in this range (i.e., h is smaller than $L' - \lceil 2m\theta \rceil$), the algorithm updates variable L' to be L (i.e., the initial content of L just at the beginning of the iteration) and variable U' to be $L' - \lceil 2m\theta \rceil - 1$. Note that this new search range is on the left-hand-side of the original search range over the sorted list. In this case, it is possible that the size of this new search range could be greater than $\lceil 2m\theta \rceil$ but the chance is low (which could be explained by Lemma 1).
  - Case (2) (i.e., $L' - \lceil 2m\theta \rceil \leq L$) In this case, the algorithm directly sets variable L' to be L and variable U' to $L' - 1$, without performing any checking subroutine. Note that this new range also has size at most $\lceil 2m\theta \rceil$ and is exactly just on the left-hand-side of the original search range over the sorted list.

  The case where U' is wrong (i.e., $U' < h$) can also be addressed symmetrically. Finally, it sets variable L to be L', variable U to be U' and enters the next iteration.

Consider the running example as shown in Figure 1, where the input is $\{p_1, p_2, p_3, p_4, p_5\}$. Assume that $\lceil 2m\theta \rceil = 2$, $L = 1$ and $U = 5$, and by performing a binary search, we obtain $L' = 4$ and $U' = 5$. Clearly, the user made some error because

p_h ($= p_2$ in this case) is not in the current search interval. By checking pair (p_3, p_4), the algorithm finds out that p_h is on the left of L' (i.e., $h < 4$), it then checks p_1, p_2 and since p_2 is more preferred to p_1, the algorithm shifts the search interval to $[2, 3]$, and enters a new iteration.

Lemma 1 shows an important property of the distance between the best point and the search range, which explains where the term $\lceil 2m\theta \rceil$ in our algorithm comes from. Before presenting Lemma 1, we first introduce an important concept called $Dist(\cdot, \cdot)$ that will be used later in this lemma. Given a point p and a range $R = [L, U]$ where L (resp. U) is the left (resp. right) boundary of the range, we define the distance between the point and the range, denoted by $Dist(p, R)$, to be 0 if $p \in [L, U]$, and $\min(|p - L|, |p - U|)$ if $p \notin [L, U]$.

Lemma 1. *We are given the range size $m = U - L + 1$ and the user error rate upper bound θ. Let R' be the range obtained by the binary search just after Step 1 of 2RI and p_h be the real best point. Then, $P(Dist(p_h, R') \geq \lceil 2m\theta \rceil) \leq \frac{1}{m\theta}$.* □

Proof sketch. We first prove that $E[Dist(p_h, R')] \leq m\theta$. Then, let D_i denote the increase of $Dist(p_h, R')$ "caused" by answering the i-th question (i.e., $Dist(p_h, R') = \sum_i D_i$) and $\sigma^2_{D_i}$ the variance of D_i. Since D_is are independent and $\sigma^2_{D_i} = \frac{m}{2^{i+1}}\theta(1 - \theta)$, using Chebyshev ineqality yields the lemma. □

Based on Lemma 1, Theorem 2 presents the main results on *2RI*. Corollary 2 shows that *2RI* is asymptotically optimal.

Theorem 2. *Given an input size n, an error rate θ and a failure probability δ, 2RI finds the best point with probability at least $1 - \delta$ using $O(\frac{1}{\log \frac{1}{2\theta}}(\log n) \log \frac{\log n}{\delta})$ rounds on expectation.*

Proof sketch. We first prove that, the expected number of iterations is $O(\log_{\frac{1}{2\theta}} n)$. Since each iteration contains $O(\log \frac{1}{2\theta} + k)$ rounds, the total number of rounds required is $O(\log n + k \log_{\frac{1}{2\theta}} n)$ with total success probability at least $1 - O(P_k \log_{\frac{1}{2\theta}} n)$, where P_k is the probability that a checking subroutine using k questions fails. Given P_k, the corresponding k can be found using Corollary 1. Setting $\delta = O(P_k \log_{\frac{1}{2\theta}} n)$ yields the theorem. □

Corollary 2. *2RI is asymptotically optimal.*

Proof. Since θ is fixed, the time complexity of 2RI is asymptotically equal to the lower bound in Theorem 1. □

V. Multi-dimensional Algorithm

Although the previous algorithm works well when $d = 2$, it cannot be adapted to a higher dimensional situation due to the change of nature of the convex hull in high dimensions. A possible way of adapting it to a high-dimensional space is to follow the idea presented in [28] and estimate the utility value of each dimension one by one using artificial points. This adaption, however, will be ineffective in a high dimension space since it cannot determine if the estimation on the utility value of each dimension is accurate enough, resulting in asking unnecessary questions. In this section, we present two algorithms, namely *Verify-Point* and *Verify-Space*, that can be applied to databases with the dimensionality $d \geq 2$. Both algorithms enjoy provable theoretical guarantee in terms of the success probability of best point retrieval and the number of questions required (also called the *round complexity*), and differ in a sense that *Verify-Space* is less dependent on the distribution of data. Both algorithms follow a 2-phase framework, and their first phase are similar. Therefore, we arrange the following sections as follows. We first introduce some preliminaries and the general algorithm framework in Section V-A, and then introduce the first phase, called *Conjecture Phase*, of the two algorithms in Section V-B. We then describe the second phase, called *Verification Phase*, of *Verify-Point* and *Verify-Space* in Section V-C1 and V-C2 respectively.

A. Preliminaries

Recall that in a d-dimensional database, the utility vector u is a d-dimensional non-negative real vector and $\sum_{i=1}^{d} u[i] = 1$. Therefore, the collection of all possible utility vectors, called the *utility space* [16] and denoted as R_0, is a $(d - 1)$-dimensional polytope. For example, as shown in Figure 2, in a 3-dimensional dataset, the utility space is a triangle with vertices $(1, 0, 0), (0, 1, 0)$ and $(0, 0, 1)$.

For any two points, namely p_i and p_j, in $conv(D)$, we can construct a hyperplane $h_{i,j}$ that passes through the origin with normal $p_i - p_j$. $h_{i,j}$ intersects the utility space and divides it into two $halfspaces$ [16]. The halfspace above (resp. below) $h_{i,j}$ is denoted as $h^+_{i,j}$ (resp. $h^-_{i,j}$), and contains all the utility vectors that ranks p_i higher (resp. lower) than p_j, or equivalently, $u \cdot p_i > u \cdot p_j$ (resp. $u \cdot p_i < u \cdot p_j$) where u is the utility vector. When the user chooses from the two displayed points, namely p_i and p_j, s/he will indicate in which halfspace her/his utility vector lies. For the ease of illustration, we use $s_{i,j}$ to denote the halfspace chosen by the user that is bounded by $h_{i,j}$. Note that based on the user's preference between p_i and p_j, $s_{i,j}$ may be either $h^+_{i,j}$ or $h^-_{i,j}$. We also denote the counterpart of $s_{i,j}$ as $s^-_{i,j}$.

For a convex polytope P, we denote the set of its vertices by V_P. The *utility range* R [40], which is defined to be the convex region that contains the true utility vector u, can be determined as follows. Initially, R is the entire utility space (i.e., R_0). When a new halfspace (e.g., $s_{i,j}$) is provided by the user's answer to the comparison between p_i and p_j, it means that $u \in s_{i,j}$ so we update R to $R \cap s_{i,j}$. Since R can be represented by the intersection of halfspaces, it is a convex polytope. Many existing studies [19, 37, 40] applied this framework and strategically choose the pair of points to reduce the number of questions asked. However, since in our problem setting, each halfspace has probability at most θ to be wrong, the utility range found by these algorithms may deviate from the real utility vector and their performance will degenerate when a user makes mistakes. To alleviate the effect of user errors, we present two algorithms that have higher error-tolerance.

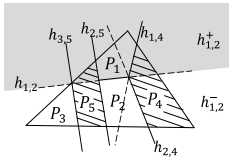


Fig. 3: Illustration on the concept of partitions

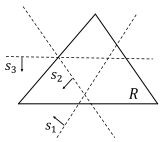


Fig. 4: An example on *Verify-Space*

The general framework of the algorithms works as follows. It runs for several *iterations* and each iteration consists of two phases, namely Phase 1, called *Conjecture Phase*, and Phase 2, called *Verification Phase*. The input to Conjecture Phase is a utility range R' indicating a convex region where the utility vector u lies, and a set C' storing some possible best points. The goal of Conjecture Phase is to "pretend" that there is no user error and interact with the user for several rounds until some *best point candidate* $p_c \in C'$ is found. During the execution of Conjecture Phase, we also maintain a set S storing all halfspaces indicated by the user. Later, in Verification Phase, we will selectively check the correctness of some halfspaces in S so that if p_c is not the *real* best point, we will still have a chance to remedy the mistake. If all the decisions made in Conjecture Phase and Verification Phase are correct, then we can conclude that p_c is the best point with high confidence. However, if some questions in Conjecture Phase are answered incorrectly, we may "waste" some rounds and focus on the wrong region of the utility space that does not contain the utility vector u. As a consequence, more than one point *can* still be the best point and at least one additional iteration is needed. In this case, with the shrunken size of the set of points under consideration, the algorithm returns to Conjecture Phase. It alternates between Conjecture Phase and Verification Phase until only one point remains under consideration. This point is returned as the final answer.

B. Conjecture Phase

Given a utility range R' indicating a convex region where the utility vector u lies, and a set C' containing some possible best points, the purpose of Conjecture Phase is to locate the user's best point candidate with the least number of rounds, without considering possible interaction errors. Note that since the only information required by Verification Phase from Conjecture Phase is the user's answer on each question, or in other words, the set S containing all halfspaces indicated by the user, the design of Conjecture Phase is quite flexible. In general, it could be any comparison-based interactive algorithms considering no user errors. Here, we adapt from the algorithm *HD-PI* [37] since it uses the least number of questions empirically to find the best point candidate.

At the beginning of Conjecture Phase, S is initialized to an empty set. In each round of Conjecture Phase, the algorithm selects a pair of points, namely p_i and p_j, and asks the user to choose the preferred point. The user's answer indicates the halfspace $s_{i,j}$ where the utility vector lies, so the algorithm updates R' to $R' \cap s_{i,j}$ and inserts $s_{i,j}$ into S. It is worth mentioning that when the user makes an error, u will not lie in $s_{i,j}$. However, in Conjecture Phase, the algorithm "pretend" that the user has no error, and the potential error will be handled later by Verification Phase. After Conjecture Phase ends, the set S is passed to Verification Phase.

The round complexity of Conjecture Phase varies from different implementations. Here, we denote the complexity of Conjecture Phase as $O(conj)$. For example, when we adopt *UH-Simplex* [40] in Conjecture Phase, the round complexity is $O(\deg_{\max} \sqrt[d]{n})$ where $\deg_{\max}$ is the maximum number of neighboring vertices for a vertex in $conv(D)$. When we adopt *HD-PI* [37] in Conjecture Phase, $O(conj)$ varies from $O(\log n')$ to $O(n')$ depending on the data distribution, where n' is the size of set C'. In our experiment, we adopt *HD-PI* in Conjecture Phase since it has the lowest round complexity though it has a linear complexity in the worst case.

C. Verification Phase

When there is no user error, p_c found in Conjecture Phase would be the true best point. However, since the user may make mistakes, the result from Conjecture Phase may no longer be the true best point. Therefore, Verification Phase is applied to make sure that the true best point is not pruned for further consideration with high probability. In the following 2 sections, we introduce Verification Phases of *Verify-Point* and *Verify-Space* respectively, where the pruning process in *Verify-Point* is performed mainly based on *points* in the dataset but that in *Verify-Space* is mainly based on the search space being considered.

1) Verification Phase of Verify-Point: During Conjecture Phase, the set of halfspaces selected by the user is stored in set S. Observe that for each halfspace $s \in S$, s contains the utility vector u with probability at least $1-\theta$ and its counterpart s^- contains u with probability at most θ (since a user makes a mistake with probability at most θ). Based on this observation, it would be more efficient to *verify* the correctness of those halfspaces that are expected to help eliminate many points from further consideration. Therefore, we develop the algorithm called *Verify-Point*.

Before we present *Verify-Point*, we first introduce some preliminaries and data structures that will be used in this algorithm. Recall that the input of *Verify-Point* is $conv(D)$. For each point $p_i \in conv(D)$, we define its *best partition* P_i, or *partition* for short, as $P_i = \{u | p_i = \arg\max_{p \in D}\ u \cdot p\}$, which corresponds to the region in the utility space where p_i is the max-utility point. In another perspective, p_i being the max-utility point means that it ranks higher than any other point. Therefore, P_i is also equal to the intersection of a set of halfspaces and the utility space R_0, i.e., $(\bigcap_{p_j \in conv(D)/\{p_i\}} h^+_{i,j}) \cap R_0$, which is a $(d-1)$-dimensional convex polytope. Given a partition P and a hyperplane $h_{i,j}$, there are 3 cases: (1) P is in $h^+_{i,j}$, (2) P is in $h^-_{i,j}$ and (3) P intersects $h_{i,j}$. For example, as shown in Figure 3, (1) P_1 is in $h^+_{1,2}$, (2) P_2 and P_3 are in $h^-_{1,2}$, and (3) both P_4 and P_5 (marked with shaded regions) intersect $h^+_{1,2}$. Suppose that the true utility vector is verified to lie in halfspace $s_{i,j}$ and this could be done in Verification Phase (to be described later) w.h.p.. If a partition P_i is disjoint from this *verified* halfspace $s_{i,j}$, we can safely prune p_i from further consideration because

for any $u \in R$, there is always some point p_j that ranks higher than p_i. To find the relation between P and the hyperplane $h_{i,j}$, it is sufficient to check P's vertices with $h_{i,j}$ in $O(|V_P|)$ time, where V_P is the set of vertices of a convex polytope P.

In Verification Phase of *Verify-Point*, given a set S of halfspaces returned from Conjecture Phase, we want to find the "best" halfspace that is expected to help eliminate the largest fraction of partitions and their corresponding points from further consideration, so that we can minimize the number of questions. To efficiently find this halfspace, we maintain a table L, where each row records the relation between a hyperplane $h_{i,j}$ and a set X of all partitions that are intersected or contained by the utility range R. Specifically, L consists of 3 columns, which are named (1) $h^+_{i,j}$, which stores a set of all partitions in X that are entirely in $h^+_{i,j}$, (2) $h^-_{i,j}$, which stores a set of all partitions in X that are entirely in $h^-_{i,j}$, and (3) intersects, which stores a set of all partitions in X that intersect with $h_{i,j}$. An example of L corresponding to Figure 3 is shown in Table III. By maintaining table L, we can efficiently find the best halfspace using a concept called $Num(\cdot)$. Specifically, given a halfspace s, we define $Num(s)$ to be the number of partitions that lie entirely outside s. For example, from Table III, we could compute $Num(h^+_{2,5}) = 2$ since there are two partitions that are completely outside halfspace $h^+_{2,5}$ (because P_3 and P_5 are in $h^-_{2,5}$). As described before, we want to find the best halfspace that is expected to prune the *largest* number of partitions. Thus, the best halfspace is defined to be the halfspace in S with the greatest value of $Num(\cdot)$ (i.e., $\arg\max_{s \in S} Num(s)$).

We are now ready to introduce *Verify-Point* which involves two steps. The first step is the initialization step. Initially, we set the utility range R to be the entire utility space R_0 and the set C containing all possible best points to be $conv(D)$. We also set L to record the relation between the set of all hyperplanes $\{h_{i,j} | p_i, p_j \in conv(D)\}$ and the set of all partitions $\{P_i | p_i \in conv(D)\}$. The second step is the iterative step which involves a number of iterations where at each iteration, Conjecture Phase is first performed and Verification is then performed until some stopping conditions are satisfied. Thus, Conjecture Phase and Verification Phase is performed in an interleaving way for this iterative step.

- **Conjecture Phase:** We create variable R' and variable C', denoting the current content of the utility range R and the current content of set C, respectively, just before Conjecture Phase is performed. The intuition of why we maintain these copies can be understood as follows. In Conjecture Phase, R' and C' is *updated* based on "conjectures" (which could be regarded as the information which has a lower confidence) according to the steps described in Section V-B (which includes how to update S, initialized to an empty set each time we re-perform Conjecture Phase). It is worth mentioning that variable R, variable C and variable L does *not* change in Conjecture Phase. But, they will be updated in the next Verification Phase based on "verified" information, and thus, these data structures are correct with high confidence.
- **Verification Phase:** Verification Phase runs for several rounds.
  - In each round, it selects the best halfspace in S and verifies with the user using the checking subroutine developed in Section IV-B. Formally, we select $s_{i,j} = \arg\max_{s \in S} Num(s)$ and then perform $Check(p_i, p_j, k)$ for checking. Based on the checking result, the correct halfspace may be $h^+_{i,j}$ or its counterpart $h^-_{i,j}$. For the ease of illustration, assume that $h^+_{i,j}$ is correct. We then update the data structures R, C, S and L as follows: (1) update the utility range R to $R \cap h^+_{i,j}$; (2) update the set S of halfspaces: remove $s_{i,j}$ from S; then, for each halfspace $s \in S$, if R is contained completely in s, or if R is completely outside s (which may happen due to user errors), remove s from S because no useful information can be obtained from s (because the question generated from s cannot help us further reduce the size of R); (3) for each partition P_l in $h^-_{i,j}$, we delete P_l from all rows of L and remove the corresponding point p_l from the set C; (4) for each partition P_l which intersects $h^+_{i,j}$, we update P_l to $P_l \cap h^+_{i,j}$, and update each row in L (corresponding to a hyperplane) by recomputing the relation of the updated P_l with the hyperplane; and (5) for each row in L and its corresponding hyperplane h, if R lies completely on one side of h (i.e., h^+ or h^-), we remove this row from L because we already know that all remaining partitions in R lies on one side of h (i.e., h^+ or h^-), so maintaining the relationship between h and partitions is no longer needed. One can verify that after the above steps, the updated table L correctly stores the relations between hyperplanes and partitions within the updated utility range.
  - Verification Phase keeps selecting the next halfspace for checking until (1) there is only 1 point left in C, which is returned as the final best point, or (2) after running for at least 1 round, all halfspaces $s \in S$ cannot prune at least β_1 portion of the remaining partitions in R, where β_1 is a non-negative real number and a user parameter. When the first stopping condition is satisfied, we could terminate the whole algorithm (since we find the answer already). When the second stopping condition is satisfied, we need to enter the next iteration and re-set variable S to an empty set.

Consider the example in Figure 3. The utility range R is the outer triangle and the partitions are polygons bounded by solid lines. The corresponding set C containing possible best points is $\{p_1, p_2, p_3, p_4, p_5\}$ and table L is shown in Table III. Assume that the algorithm enters Verification Phase with $S = \{h^+_{2,5}, h^+_{1,4}, h^+_{2,4}\}$. Verification Phase chooses to check $h^+_{2,5}$ first, because $Num(h^+_{2,5})$ is the largest (i.e., 2 in this case). Assume that after the user is checked via the checking subroutine, $h^+_{2,5}$ is verified to be correct. Then, the algorithm will update the utility range R to $R \cap h^+_{2,5}$, remove p_3 and p_5 from C (since P_3 and P_5 are in $h^-_{2,5}$) and remove $h^+_{2,5}$ from S. The updated L is shown in Table IV.

It is shown in [9] that given the input size n, the error rate θ and a failure probability δ, *Verify-Point* returns the best point with probability at least $1-\delta$, using an expected number of $O(\frac{c}{1-\theta}\log n(conj + \log\log n + \log\frac{1}{\delta}))$ rounds where c is a data-dependent parameter denoting the pruning power (whose value is studied in Section VI-A) and $conj$ is the round complexity of Conjecture Phase. The processing time of each round in Verification Phase is $O(n^3 + n^2 d\,|V|)$, where $|V|$ is the maximum number of vertices of all polytopes processed in Verification Phase.

It is worth mentioning that $|V|$ is not large in our typical setting. Note that $|V| = O(m^{\lfloor \frac{d}{2} \rfloor})$ [36] where m is the largest number of halfspaces bounding a polytope. Typically, m is small (i.e., $m << n$) although $m = O(n)$ in the worst case. In our experiment, when $d = 5$ and $n = 10,000$, we have $m < 100$. Besides, following [27, 37, 40], due to the limited number of attributes for user's decision making, d is not large too (usually, d is at most 7).

2) Verification Phase of Verify-Space: Our second algorithm, *Verify-Space*, is closely related to *Verify-Point*: Its Verification Phase also runs for several rounds and in each round, it selects the best halfspace in S for checking (i.e., calling the checking subroutine), where S stores all halfspaces indicated by the user in Conjecture Phase. However, the key difference is that instead of finding the halfspace that prunes the largest *number* of partitions, *Verify-Space* directly finds the halfspace that prunes the largest *space* of utility range R, which makes it less data-dependent.

Verify-Space shares the same structure with *Verify-Point*. The main difference is that in *Verify-Space*, whenever we want to decide the next halfspace for checking, we find the halfspace $s \in S$ that is expected to remove the *largest* space of the utility space (i.e., the halfspace $s \in S$ that *minimizes* the resulting utility space R). To find this "best" halfspace, one major step is to compute the volume of the polytope formed by intersecting the halfspace s with the current utility range R (i.e., $s \cap R$). Let $vol(P)$ denote the volume of a $(d-1)$-dimensional polytope P. Our task is to find $s_{i,j} = \arg\min_{s \in S} vol(s \cap R)$. However, computing the volume of a polytope in a high-dimensional space is time-consuming. There are several approximation algorithms that can estimate the volume of polytope, but to the best of our knowledge, even the fastest algorithm among them [7] takes $O(d^3)$ time , which is very time-consuming if we compute the volume of $s \cap R$ for each $s \in S$. Fortunately, what we really want is to minimize the ratio between the new utility range and the old one (i.e., $\frac{vol(s \cap R)}{vol(R)}$), and thus, it is not necessary to compute the exact volume. Therefore, we apply a random sampling technique called *Billiard Walk* [8] to sample a number of points in R and use the sampled points to compute this ratio. Each point can be sampled within $O(|S|\,d)$ time [7]. Lemma 2 shows that sampling a small number of points suffices to compute all ratios with high accuracy.

Lemma 2. *Let $R \subseteq \mathbb{R}^d$ be the utility range. Let T be a set of points randomly sampled from R. Then, given a non-negative real number $\rho \in [0,1]$ and a non-negative real number ε, if sample size $|T| = O(\frac{1}{\varepsilon^2}(\log\frac{1}{\rho} + d))$, then with probability $1-\rho$, for any halfspace $s \subseteq \mathbb{R}^d$, $\frac{vol(s \cap R)}{vol(R)} - \frac{|\{t \in T | t \in s \cap R\}|}{|\{t \in T | t \in R\}|} \leq \varepsilon$.*

The above lemma is derived directly from Theorem 5 in [23]. In order to make the relative errors of all intersections smaller than ε, we need only to sample $|T| = O(\frac{1}{\varepsilon^2}(\log\frac{|S|}{\rho} + d))$ points. The failing probability can be easily proved to be less than ρ using the union bound. As a convention, in our experiments, we set $\rho = 0.1$.

To illustrate Verification Phase of *Verify-Space*, consider the example shown in Figure 4. Assume that now the utility range is R and after Conjecture Phase, we record $S = \{s_1, s_2, s_3\}$ from the user. After sampling, the algorithm first chooses to verify s_2 because it has the smallest intersection with R. If the user confirms that s_2 is correct, then the utility range is updated to $s_2 \cap R$, and s_2 is removed from S. The algorithm then continues to select from S the next halfspace that has the smallest intersection with $s_2 \cap R$.

It is shown in [9] that given the input size n, the error rate θ and a failure probability δ, *Verify-Space* returns the best point with probability at least $1-\delta$, using an expected number of $O(\frac{d}{1-2\theta}\log n(conj + \log\log n + \log\frac{1}{\delta}))$ rounds, where d is the dimensionality and $conj$ is the round complexity of Conjecture Phase. The processing time of each round in Verification Phase is $O(n^3 + n^2 d\,|V|)$, where $|V|$ is the maximum number of vertices of all polytopes processed in Verification Phase.

VI. Experiment

We conducted experiments on a computer with 1.80 GHz CPU and 12 GB RAM. All programs were implemented in C/C++. The source code and the datasets used could be found in https://github.com/qixuchen/Robust_Interact.

Datasets. The experiments were conducted on synthetic and real datasets that are used in [28, 37, 40]. Specifically, we generated *anti-correlated* datasets by a dataset generator developed for skyline operators [6].

Besides, we used 4 real datasets, namely *GasSensor* [13], *AirQuality* [12], *Weather* [15] and *HTRU* [14]. *GasSensor* contains 928,991 two-dimensional points. *AirQuality* includes 420,478 tuples described by 4 attributes. *Weather* is a 6-d dataset consisting of 96,483 weather records. *HTRU* involves 7 attributes and contains 17,898 points. Each dimension is normalized into range $[0,1]$. Following the existing studies [37, 40], we preprocessed all the datasets to contain only the skyline points (which are all the possible best points for any utility function), since we are only interested in the best point. As can be found in [9], this preprocessing step does not unfairly favor our algorithms. The skyline sizes of synthetic datasets with size 100,000 and d = 2, 3, 4, and 5 are 35, 578, 3780 and 11439, respectively. The skyline sizes of real datasets *GasSensor*, *AirQuality*, *Weather* and *HTRU* are 2074, 3948, 2110 and 11720, respectively.

Algorithms. We evaluated our 2-d algorithm: *2RI* and two multi-dimensional algorithms: *Verify-Space* and *Verify-Point*. The competitor algorithms are: *Median* [40], *Hull* [40], *2D-PI* [37], *Active-Ranking* [19], *Preference-Learning* [31], *UtilityApprox* [28], *UH-Simplex* [40], *HD-PI* [37] and *RH*

[37]. Since some of them cannot return the best point directly, when comparing them with our algorithms, we made the following adaptions.

(1) Algorithms *Median*, *Hull* and *2D-PI* are designed for 2 dimensional tasks. *Median* and *Hull* already aim at returning the user's best point so they are left unchanged. *2D-PI* returns one of top-l points where l is a user parameter, so we set l to 1 such that it returns the best point. (2) Algorithm *Active-ranking* aims at learning the entire ranking of all points by interacting with the user. We return the best point after the full ranking is determined. (3) Algorithm *Preference-learning* focuses on learning the utility vector of the user. According to the experimental result in [31], the utility vector learnt is very close to the theoretical optimum if the error threshold ϵ of the learnt utility vector is set to a value below 10^{-5} (e.g., 10^{-6}). In our experiment, we set ϵ to 10^{-6} (since the learnt utility vector could achieve the optimum), and return the best points w.r.t. the learnt utility vector. (4) Algorithm *UtilityApprox* and *UH-Simplex* focus on reducing the regret ratio below a given threshold ϵ. Following [37], we set $\epsilon = 1 - f(p_2)/f(p_1)$, where p_1 and p_2 are the top-1 and top-2 points w.r.t. the user's utility vector, respectively. In this way, if no user error is made, the returned point is guaranteed to be the best point. (5) Algorithms *HD-PI* and *RH*, similar to the 2-d algorithm *2DPI*, aim at returning one of the top-l points. We make them return the best point by setting $l = 1$.

Parameter Setting. We evaluate the performance of each algorithm by varying different parameters: (1) the dataset size N; (2) the dimensionality d, (3) the user error rate upper bound θ, (4) the stopping threshold β_1 in *Verify-Point* and β_2 in *Verify-Space*, (5) the variable ε in Lemma 2, and (6) the parameter k in the checking subroutine. Unless stated explicitly, for each synthetic dataset, we set $N = 100,000$ and $d = 4$. We set $\theta = 0.05$, which is a reasonable error rate upper bound according to the human reliability assessment data in [21]. According to the results in Section VI-A, we set the default value of $\beta_1 = 0.2$, $\beta_2 = 0.2$, $k = 3$ and $\varepsilon = 0.1$.

Performance Measurement. We evaluate the performance of each algorithm with the following measurements: (1) *Accuracy* which is the probability of retrieving the best point. Formally, we define the accuracy to be $\frac{N_c}{N_e}$ where N_e is the total number of experiments and N_c is the number of times the best point is returned. (2) *Number of questions* required to return the result, and (3) *Execution time* which is the average processing time to determine the question asked in each round. For each setting, we repeat the algorithm 100 times and the average value is reported.

In Section VI-A, we study different configurations of our algorithms. The performance of all algorithms on synthetic and real datasets are discussed in Section VI-B and VI-C, respectively. We conducted a user study with user errors in Section VI-D. We summarize the experiments in Section VI-E.

A. Study on Configuration of Our Algorithms

In this section, we study the effect of different settings of parameters β_1, β_2, ε and k on our algorithms *2RI*, *Verify-Point* and *Verify-Space*. We also study the empirical value of parameter c in Section V-C. We briefly describe the experiments and results here and the related figures are included in [9] for the sake of space.

We studied the effect of ε by varying ε between 0.05 and 0.25. We observe no significant change in the number of questions and the accuracies. But, when $\varepsilon > 0.15$, the standard deviation of the number of questions increases, and when $\varepsilon < 0.1$ the average processing time increases. We eventually decided to set $\varepsilon = 0.1$.

We studied the effect of β_1 on *Verify-Point* and β_2 on *Verify-Space* by varying them from 0 to 0.5. Changing β does not have obvious impact on the accuracy, but both algorithms use the least number of rounds when $\beta_1 = \beta_2 = 0.2$. Therefore, in the rest of our experiments, we set $\beta_1 = \beta_2 = 0.2$.

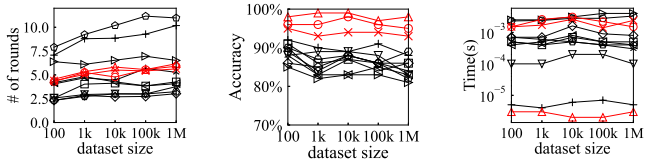
We evaluated the effect of different choices of k on *2RI* (on 2-d synthetic dataset), *Verify-Point* and *Verify-Space* (on 4-d synthetic dataset). We chose $k = 3$ for later experiments because it asks a small number of questions while achieving a satisfactory level of accuracy (i.e., 90%).

We studied on the empirical value of c in Section V-C. For all the datasets we tested, the value of c never exceeds 3. Therefore, c can be regarded as a small constant.

B. Performance on Synthetic Datasets

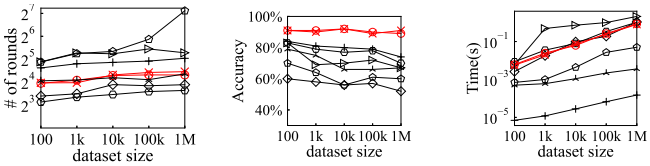
We compared our 2-d algorithm *2RI* against *2DPI*, *Median* and *Hull* on a 2-d synthetic dataset. For completeness, we also record the performance of all d-dimensional algorithms. Figure 5 presents the performance of all the algorithms when N varies from 100 to 1,000,000. According to Figure 5 (c), all algorithms determine the next question to be asked within 10^{-2} second and their execution times do not increase significantly when the input size grows. As shown in Figure 5 (b), *2RI* achieves the highest accuracy among all the algorithms under all settings, and finishes within only 5-6 questions as shown in Figure 5 (a). On the other hand, *2DPI*, *Median* and *Hull* cannot efficiently handle user errors and their accuracies is at least 10 percentage lower than *2RI*. It is worth mentioning that a lower accuracy could lead to unforgettable and unchangeable consequences as described in Section I.

In Figure 6, we evaluated the performance of our algorithms *Verify-Point* and *Verify-Space* against other existing d-dimensional algorithms on 4-d synthetic datasets. The number of questions required by *Verify-Point* and *Verify-Space* gradually increases along with the input size. We observe that our algorithms are the most accurate on finding the best point, and their accuracies decrease with the slowest rate along with the increase in the input size. Algorithm *UtilityApprox* has the highest accuracy among the remaining algorithms, but, it is still 10% lower than ours and it asks around 10 more questions. *HD-PI* and *UH-Simplex* ask slightly fewer questions than our algorithms. But, their accuracies are more than 10 percent and 30 percent lower than ours, respectively. Algorithm *Active-Ranking* aims at learning the entire ranking and algorithm *Preference-Learning* aims at learning the user's utility vector. Therefore, although they claim that they can



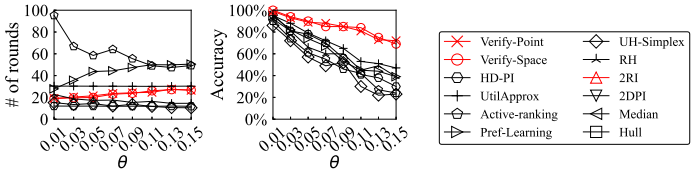
(a) (b) (c)

Fig. 5: Effect of input size on 2d dataset



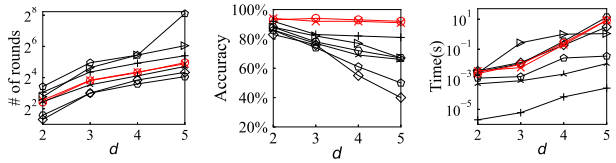
(a) (b) (c)

Fig. 6: Effect of input size on 4d dataset



(a) (b)

Fig. 7: Effect of θ



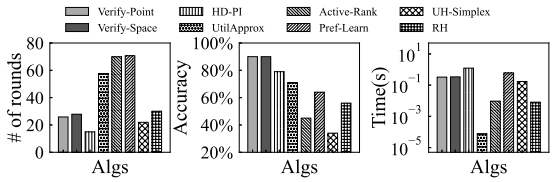
(a) (b) (c)

Fig. 8: Effect of d

handle user errors, their performance is poor in terms of both the number of questions asked and the accuracy. According to Figure 6 (c), for all algorithms, the running time on deciding the next question to be asked increase along with the input size. But, *Verify-Point* and *Verify-Space* finish within 1 second on 1M datasets, which is acceptable for real-time interaction.

We studied the effect of different values of the user error rate θ on d-dimensional algorithms. Figure 7 shows the result. When θ grows, the degeneration of accuracy can be observed on all the algorithms. However, the performance of *Verify-Point* and *Verify-Space* degenerates at the slowest rate and their accuracies remain the highest for all settings of θ. In particular, when θ is high (i.e., 0.15), our algorithms can still reach 70% accuracy while all other algorithms are lower than 50%. We notice that the number of questions asked by *Verify-Point* and *Verify-Space* gradually increases when θ increases, which can be attributed to the extra checking rounds required to resolve conflicts caused by extra user errors. The number of questions grows in a slow rate and both algorithms finish within 30 questions even if the error rate is high (i.e., 0.15). Figure 8 presents our experiment on evaluating the scalability on the dimensionality d. Compared with the existing algorithms, *Verify-Point* and *Verify-Space* constantly achieve higher accuracies for all dimensional settings, and the gap gets even larger when d increases. Meanwhile, the number of questions asked by our algorithms grows only by 6-7 questions for each dimensionality increase. As for the running time, all algorithms require more time when the dimensionality is larger. However, for $d = 5$, our algorithms still finish within several seconds, which is an acceptable interactive speed.

We studied the P95, median and accumulated processing times of algorithms (in addition to the average time). Firstly, we observe that no matter what measurements we used (i.e., P95, median and average time costs), *2RI* runs within microseconds, and at least 95% questions in *Verify-Point* and *Verify-Space* can be determined in 1-2 seconds, which proves that our proposed algorithms run in an interactive speed. Secondly, *Verify-Point* and *Verify-Space* have the accumulated processing time at most 6 seconds but the average processing time per round is less than 0.3 seconds. Detailed results could be found in Section B3 of our technical report [9].



(a) (b) (c)

Fig. 9: Results on dataset *Weather*

Furthermore, we experimented on how different preferences (i.e., the weight on different dimensions may vary for distinct users) affect the performance of algorithms. We observe that the superiority of *Verify-Point* and *Verify-Space* is consistent in different preference settings since they achieve the highest accuracy among all competitors in all cases. Detailed results could be found in Section B4 of our technical report [9].

C. Performance on Real Datasets

We conducted experiments on 4 real datasets, namely *GasSensor*, *AirQuality*, *Weather* and *HTRU*. For four 2-d algorithms *2RI*, *2DPI*, *Median* and *Hull*, their performance on the 2-d dataset *GasSensor* is reported. For completeness, the performance of all d-dimensional approaches are also reported. Among all 2-d algorithms, *2RI* requires less than 5 questions, and obtains the highest accuracy which is around 98%. Other competitors take 1-2 less questions compared with *2RI*, but their accuracies are much lower. Specifically, the accuracies of *2DPI*, *Median* and *Hull* are 90%, 90% and 86%, respectively. Detailed experimental results can be found in [9].

For d-dimensional algorithms, we studied their performance on all 4 datasets. Due to the lack of space, we only show the results on *Weather* in Figure 9. The performance on *GasSensor*, *AirQuality*, and *HTRU* can be found in [9]. *Verify-Point* and *Verify-Space* obtain the highest accuracy using a small number of questions on all datasets, which is consistent with their performance on synthetic datasets.

D. User Study

To see the impact of user errors and how our algorithms can help improve the quality of the returned point, we conducted a user study on the *Car* dataset [37], which consists of 68,005 used cars. Following the same settings in [31, 37, 40], we randomly selected 1000 candidate cars from the database and

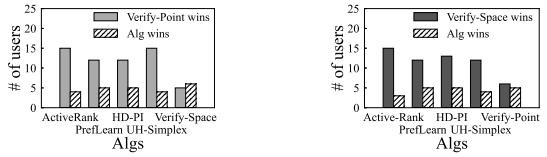
each record is described by 4 attributes, namely price, year of purchase, power and used kilometers. We compared our algorithms *Verify-Point* and *Verify-Space* against 4 existing algorithms, namely *HD-PI*, *UH-Simplex*, *Preference-learning* and *Active-Ranking*. For each algorithm, the users were asked to select the preferred car from a pair of cars for several rounds until a car is returned. 25 participants were recruited and their average results were reported. Since we cannot directly obtain the user's utility vector, *UH-Simplex* and *Preference-Learning* were re-adapted (different from the way described previously): (1) Algorithm *Preference-Learning* maintains an estimated user's utility vector u during the interaction. We compared the user's answers of some randomly selected questions with the prediction w.r.t. u. If 75% questions [31] can be correctly predicted, we stop and return the best point w.r.t. u. (2) For *UH-Simplex*, we set the threshold $\epsilon = 0$, which guarantees that the returned car is the best point in the algorithm's view.

Each algorithm was measured via the following metrics: (1) *the number of questions asked*; and (2) *the dissatisfactory level*, which is an integer score ranging from 0 to 10 given by each participant. It indicates how dissatisfied the participant feels about the returned car, where 0 indicates the least dissatisfied and 10 the most dissatisfied.

We also measured how frequently user errors occur during interactions. Since we cannot directly verify the correctness of each answer, we created a new version of *Verify-Point*, called *Verify-Point-Adapt*, and changed the checking subroutine as follows: Instead of stopping immediately once the majority is determined, it always checks a pair of points for exactly k times before returning the answer. Modified in this way, let n_c denote the number of checking subroutines invoked, and denote w_i the number of questions asked and m_i the number of majority answers of the user in the i-th checking, $1 - \frac{\sum_{i=1}^{n_c} m_i}{\sum_{i=1}^{n_c} w_i}$ will be an unbiased estimator of the user error rate. Among all the 220 checking questions asked by *Verify-Point-Adapt*, users made 10 errors, yielding an overall error rate of 4.5%.

In the experiment, we observe that *Verify-Point* and *Verify-Space* obtain the lowest dissatisfaction score, which is slightly above 1 (out of 10). On the other hand, the dissatisfaction scores of all other competitors are at least 3. As for the number of questions asked, *HD-PI* and *UH-Simplex* use the least number of rounds which is around 8. *Verify-Point* and *Verify-Space* also finish with slightly over 10 rounds. On the other hand, *Active-Ranking* and *Preference-Learning* require a lot more rounds. They use on average 23 and 32 rounds, respectively. The related figures can be found in [9].

In Figure 10 (a) and (b), we compared the user's preference on the recommendations of our algorithms *Verify-Point* and *Verify-Space*, respectively, against *HD-PI*, *UH-Simplex*, *Preference-Learning* and *Active-Ranking*. Specifically, for each user, if the dissatisfaction score of algorithm *A* is lower than algorithm *B*, then the recommendation of *A* is better than *B*. For example, in Figure 10 (a), when *Verify-Point* is compared with *HD-PI*, 12 users favor the car recommended by *Verify-Point*, while only 4 think the recommendation of



(a) *Verify-Point* (b) *Verify-Space*

Fig. 10: User study comparing whether Verify-Point/Verify-Space is better than each of the competitors

HD-PI is more preferred. Overall, the recommendation made by our algorithms are much more preferred than the existing algorithms. On the other hand, there is no significant gap between the performance of *Verify-Point* and *Verify-Space*.

E. Summary

The experiments demonstrated the superiority of our 2-d algorithm *2RI* and our d-dimensional algorithms, *Verify-Point* and *Verify-Space*, over the existing approaches: (1) we are efficient and effective. We achieve nearly 100% accuracy in most experiments within a small number of rounds. In particular, for all 2-d experiments, *2RI* consistently outperforms all other algorithms and obtains an accuracy close to 100% even when the input size is large (e.g., 1,000,000). In d-dimensional experiments, *Verify-Point* and *Verify-Space* maintain the highest accuracy among all competitors. (2) The scalability of *Verify-Point* and *Verify-Space* is demonstrated. Specifically, they are scalable to the input size and the dimensionality. For example, on the 7-d dataset *Diamond*, our algorithms obtain over 90% accuracies using only slightly over 20 questions, while *UtilitApprox*, *Preference-Learning* and *Active-Ranking* are significantly lower than ours using over 50 questions. (3) Our algorithms are capable to handling many user errors. When the user error rate is very large (e.g., 0.15), only our algorithms can keep the accuracies above 70%, while all other algorithms drop under 50%.

VII. Conclusion

In this paper, we propose a more robust interactive model that returns the best point in dataset under the setting of random errors. Our model is more practical than existing algorithms in a sense that ours can return the best point with high confidence even if the user makes mistake when interacting with the system. Specifically, we propose a 2-d algorithm that is asymptotically optimal in terms of the number of rounds required and two multi-dimensional algorithms with provable guarantee and superior empirical performance. We conducted extensive experiments to show that our algorithms is both efficient and effective in determining the best point facing user errors compared with existing algorithms. In the future, we consider the case where user makes persistent errors when answering questions.

Acknowledgements

We are grateful to the anonymous reviewers for their constructive comments on this paper. The research of Qixu Chen and Raymond Chi-Wing Wong is supported by PRP/026/21FX and HK PhD fellowship.

REFERENCES

[1] Yongkil Ahn. The economic cost of a fat finger mistake: a comparative case study from samsung securities's ghost stock blunder. *Journal of Operational Risk*, 16(2), 2019.

[2] C Bradford Barber, David P Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software (TOMS)*, 22(4):469–483, 1996.

[3] Ilaria Bartolini, Paolo Ciaccia, Vincent Oria, and M Tamer Özsu. Flexible integration of multimedia subqueries with qualitative preferences. *Multimedia Tools and Applications*, 33(3):275–300, 2007.

[4] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. Domination in the probabilistic world: Computing skylines for arbitrary correlations and ranking semantics. *ACM Transactions on Database Systems (TODS)*, 39(2):1–45, 2014.

[5] Ilaria Bartolini, Paolo Ciaccia, and Florian Waas. Feedbackbypass: A new approach to interactive similarity query processing. In *VLDB*, pages 201–210, 2001.

[6] Stephan Borzsony, Donald Kossmann, and Konrad Stocker. The skyline operator. In *Proceedings 17th international conference on data engineering*, pages 421–430. IEEE, 2001.

[7] Apostolos Chalkis, Ioannis Z Emiris, and Vissarion Fisikopoulos. A practical algorithm for volume estimation based on billiard trajectories and simulated annealing. *arXiv preprint arXiv:1905.05494*, 2019.

[8] Apostolos Chalkis and Vissarion Fisikopoulos. volesti: Volume approximation and sampling for convex polytopes in r. *arXiv preprint arXiv:2007.01578*, 2020.

[9] Qixu Chen and Raymond Chi-Wing Wong. Finding best tuple via interaction with error-prone user input (technical report). http://home.cse.ust.hk/~raywong/paper/interaction-error-technicalReport.pdf, 2022.

[10] Sean Chester, Alex Thomo, S Venkatesh, and Sue Whitesides. Computing k-regret minimizing sets. *Proceedings of the VLDB Endowment*, 7(5):389–400, 2014.

[11] Paolo Ciaccia and Davide Martinenghi. Reconciling skyline and ranking queries. *Proceedings of the VLDB Endowment*, 10(11):1454–1465, 2017.

[12] AirQuality dataset. https://archive.ics.uci.edu/ml/datasets/Beijing+Multi-Site+Air-Quality+Data.

[13] GasSensor dataset. https://archive.ics.uci.edu/ml/datasets/Gas+sensors+for+home+activity+monitoring.

[14] HTRU dataset. https://archive.ics.uci.edu/ml/datasets/HTRU2.

[15] Weather dataset. https://www.kaggle.com/datasets/muthuj7/weather-dataset.

[16] Mark Theodoor De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational geometry: algorithms and applications*. Springer Science & Business Media, 2000.

[17] Brian Eriksson. Learning to top-k search using pairwise comparisons. In *Artificial Intelligence and Statistics*, pages 265–273. PMLR, 2013.

[18] Kilem L Gwet. Intrarater reliability. *Wiley encyclopedia of clinical trials*, 4, 2008.

[19] Kevin G Jamieson and Robert Nowak. Active ranking using pairwise comparisons. *Advances in neural information processing systems*, 24, 2011.

[20] Yiling Jia, Huazheng Wang, Stephen Guo, and Hongning Wang. Pairrank: Online pairwise learning to rank by divide-and-conquer. In *Proceedings of the Web Conference 2021*, pages 146–157, 2021.

[21] Barry Kirwan. *A guide to practical human reliability assessment*. CRC press, 2017.

[22] Jongwuk Lee, Gae-won You, Seung-won Hwang, Joachim Selke, and Wolf-Tilo Balke. Interactive skyline queries. *Information Sciences*, 211:18–35, 2012.

[23] Yi Li, Philip M Long, and Aravind Srinivasan. Improved bounds on the sample complexity of learning. *Journal of Computer and System Sciences*, 62(3):516–527, 2001.

[24] Tie-Yan Liu et al. Learning to rank for information retrieval. *Foundations and Trends® in Information Retrieval*, 3(3):225–331, 2009.

[25] Denis Mindolin and Jan Chomicki. Discovering relative importance of skyline attributes. *Proceedings of the VLDB Endowment*, 2(1):610–621, 2009.

[26] Kyriakos Mouratidis, Keming Li, and Bo Tang. Marrying top-k with skyline queries: Relaxing the preference input while producing output of controllable size. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1317–1330, 2021.

[27] Kyriakos Mouratidis and Bo Tang. Exact processing of uncertain top-k queries in multi-criteria settings. *Proceedings of the VLDB Endowment*, 11(8):866–879, 2018.

[28] Danupon Nanongkai, Ashwin Lall, Atish Das Sarma, and Kazuhisa Makino. Interactive regret minimization. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 109–120, 2012.

[29] Danupon Nanongkai, Atish Das Sarma, Ashwin Lall, Richard J Lipton, and Jun Xu. Regret-minimizing representative databases. *Proceedings of the VLDB Endowment*, 3(1-2):1114–1124, 2010.

[30] Peng Peng and Raymond Chi-Wing Wong. Geometry approach for k-regret query. In *2014 IEEE 30th International Conference on Data Engineering*, pages 772–783. IEEE, 2014.

[31] Li Qian, Jinyang Gao, and HV Jagadish. Learning user preferences by adaptive pairwise comparison. *Proceedings of the VLDB Endowment*, 8(11):1322–1333, 2015.

[32] Assessing Questionnaire Reliability. https://select-statistics.co.uk/blog/assessing-questionnaire-reliability/, 2022.

[33] Wenbo Ren, Jia Kevin Liu, and Ness Shroff. On sample complexity upper and lower bounds for exact ranking from noisy comparisons. *Advances in Neural Information Processing Systems*, 32, 2019.

[34] Gerard Salton. Automatic text processing: The trans-

formation, analysis, and retrieval of. *Reading: Addison-Wesley*, 169, 1989.

[35] Thomas Seidl and Hans-Peter Kriegel. Efficient user-adaptable similarity search in large multimedia databases. In *VLDB*, volume 97, pages 506–515, 1997.

[36] Csaba D Toth, Joseph O'Rourke, and Jacob E Goodman. *Handbook of discrete and computational geometry*. CRC press, 2017.

[37] Weicheng Wang, Raymond Chi-Wing Wong, and Min Xie. Interactive search for one of the top-k. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1920–1932, 2021.

[38] A Student with Top-tier Score Admitted by mediocre University (Chinese version only). https://news.southcn.com/node_6854f1135c/4357641930.shtml, 2020.

[39] Xingxing Xiao and Jianzhong Li. Rank-regret minimization. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 1848–1860. IEEE, 2022.

[40] Min Xie, Raymond Chi-Wing Wong, and Ashwin Lall. Strongly truthful interactive regret minimization. In *Proceedings of the 2019 International Conference on Management of Data*, pages 281–298, 2019.

[41] Jiping Zheng and Chen Chen. Sorting-based interactive regret minimization. In *Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint International Conference on Web and Big Data*, pages 473–490. Springer, 2020.

APPENDIX