Robust Interaction

ABSTRACT

Given a large dataset containing a lot of points, in order to find the point which is the most interesting to a user efficiently, we could ask the user *interactively* a number of questions, each requiring the user to compare 2 points for choosing a more preferred point, and obtain his/her answers to learn his/her preference (represented by a utility function). With the utility function learnt, the most interesting point could be determined easily. In the literature of the database community, this interaction becomes more and more popular because it does not require the user to know his/her utility function and could return the most interesting point to the user, which overcomes the disadvantages of both traditional top-k queries (which requires to know the utility function in advance) and skyline queries (which could return a lot of possible points as an output). 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.

ACM Reference Format:

. 2023. Robust Interaction. In *Proceedings of SIGMOD 2023 (Conference acronym '23)*. ACM, New York, NY, USA, 18 pages. https://doi.org/XXXXXXX.XXXXXXX

1 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* [5]. The top-k query measures the *utility* of tuples based on an *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 in the worst case.

Motivated by this, a novel interactive framework [16, 23, 25] 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 [16, 23, 25] 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/errors 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% [11]. 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 at least 10%-30% chance to make at least one mistake for 10 rounds. 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

Conference acronym '23, June 18–23, 2023, Seattle, WA, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

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 mediocre college with a similar name to his target university due to his mistake of choosing a wrong school in the tertiary school selection system [24]. The second case is about a huge financial loss due to a well-known "fat finger error" (which refers to 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 top-1 point retrieval problem* considering random user errors during user interaction, which is more realistic. Roughly speaking, our problem is to find the top-1 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 [11]. In our user study, θ is found to be 4.5%.

Unfortunately, most (if not all) existing interaction algorithms [16, 23, 25] without considering user errors may find the *undesirable* point based on the *wrongly learnt* utility function because they assume that the user never makes mistake. In our experiment, all of these algorithms return incorrect results. For example, on a dataset with 1,000,000 points, the accuracies of all closely related algorithms *UtilityApprox* [16], *UH-Simplex* [25] and *HDPI* [23] are at most 74% only.

Furthermore, all adapted versions of existing interactive algorithms considering user errors [10, 19] for this problem do not perform satisfactorily. In our experiment, the accuracies of the adapted versions of *Active-Ranking* [10] and *Preference-Learning* [19] on a dataset with size 1,000,000 are at most 67% only, which is not acceptable.

Contributions. We summarize our contributions as follows. Firstly, we are the first to propose the top-1 point retrieval problem considering random interaction errors during user interaction. Secondly, we prove a lower bound on the expected number of questions needed to determine the top-1 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 top-1 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 in finding the top-1 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 2 gives our problem definition. Section 3 shows the related work. We introduce the algorithm for the dataset containing two dimensions in Section 4 and two algorithms for the dataset containing at least two dimensions in Section 5. In Section 6, we present the experimental results. Section 7 concludes the paper.

2 PROBLEM DEFINITION

In this section, we provide a formal definition to our problem. Firstly, in Section 2.1, we introduce some basic terminologies. Then, we formally define the random user error setting in Section 2.2. In Section 2.3, we study the lower bound of the number of questions to return the top-1 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 many more than d attributes, but the user is interested in exactly d of them.

2.1 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 1. 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 [16, 19, 23, 25], 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 $p_h = argmax_{p \in D}\, u \cdot p$. Note that scaling u does not change the rank of points in D and thus, does not change the top-1 point. Therefore, without loss of generality, we assume that $\sum_{i=1}^{d} u[i] = 1$.

2.2 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 [10, 12, 14, 19] 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. 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 points with the higher utility in two points p_i and p_j, the user 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

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 1: Dataset and utility

$h_{i,j}$	$h^+_{i,j}$	$h^-_{i,j}$	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 2: 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 3: Table L after selecting $h_{2,5}$

error rate described in Section 1. If θ is greater than 0.5, there is no hope that we could find the top-1 point for this user (because the user already gives more than half of his/her answers wrongly.)

2.3 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 top-1 point? [25] proved that for any dimensionality d, there exists a dataset of size n such that any comparison-based interactive algorithm must ask $\Omega(\log_2 n)$ questions to determine the top-1 point. However, this study does not consider user errors and cannot be applied to our problem. Therefore, 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 interactive algorithm needs to ask $\Omega(\frac{4\theta^2(1-\theta)}{(1-2\theta)^2}(\log n)\log(\frac{\log n}{\delta}))$ rounds on expectation to determine the top-1 point with confidence at least $1-\delta$.*

Proof sketch. We first show that in order to find the top-1 point, $\Omega(\log n)$ questions must be asked and the expected number of errors we must handle is $m = \Omega(\theta \log n)$. We then prove that to let the total failure probability be less than δ, at least $k = \Omega(\frac{4\theta(1-\theta)}{(1-2\theta)^2}\log(\frac{\log n}{\delta}))$ additional questions must be asked to handle each error. The total number of rounds is therefore $\Omega(\log n + mk) = \Omega(\frac{4\theta^2(1-\theta)}{(1-2\theta)^2}(\log n)\log(\frac{\log n}{\delta}))$. □

3 RELATED WORK

Besides the traditional top-k and skyline queries mentioned in Section 1, various types of multi-criteria decision making queries were proposed. Two types of queries that does not require user interaction are the similarity query [2, 21] and the regret minimizing query [8, 17, 18]. The similarity query 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 [3], which may be an unrealistic assumption in practice. On the other hand, the regret minimizing query [8, 17, 18] defines a measurement called *regret ratio*, and returns a set of tuples that minimizes the regret ratio of the user. Although regret minimizing query is not as demanding as top-k and similarity query, it is hard to achieve both a small regret ratio and a small size of return set. When a small regret ratio is fixed, the output size is usually large [8, 25].

To overcome the limitations of the above queries, some existing studies [3, 10, 16, 19, 20, 23, 25] proposed to involve user interactions. The form of interactions varies. A form of interaction that is widely adopted in [10, 16, 19, 23, 25] is to ask the user to select the favorite point among a set of displayed points. [16] proposed the interactive regret minimizing query, which aims to lower the regret ratio while keeping the output size small. [25] follows the study on regret minimization and proposed two algorithms, namely *UH-Simplex* and *UH-Random*, that only displays real tuples inside the dataset. [23] 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.

Although there are studies focusing on other types of interactions, they typically require too much user effort. [26] developed the algorithms in [25] and reduced the number of rounds needed, but by asking the user to give a ranking instead of only selecting a single point. [15] asks the user to partition points into *desirable* and *undesirable* groups to learn his/her preference. [3, 4, 20] proposed the interactive similarity query that learns the distance function and query tuple through 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.

There are some studies [10, 19] considering preference learning with possible user errors (though their original aims are different from ours). Specifically, [19] proposed *Preference-Learning* to learn user's preference that copes user errors by introducing a slack variant in a linear SVM. [10] proposed *Active-Ranking* and resolves possible conflict using the majority vote. Although these algorithms can be *adapted* to find the top-1 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., [10, 19]) and some other algorithms (e.g., [3, 15, 26]) 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 top-1 point with a desired confidence level.

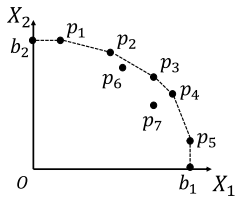

Figure 1: 2D convex hull

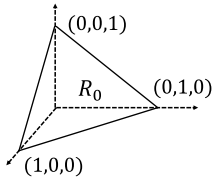

Figure 2: Utility space in 3D

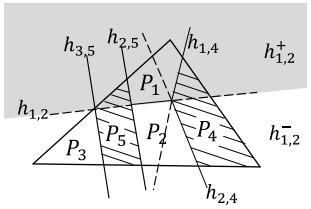

Figure 3: Partitions

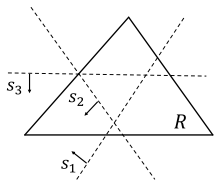

Figure 4: Halfspaces

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.
```

4 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 4.1. Then, in Section 4.2, we introduce a useful checking scheme that will be used in later algorithms. Finally, we show the details of *2RI* in Section 4.3. This algorithm uses a number of rounds that is asymptotically equal to the lower bound proved in Secion 2.3 .

4.1 Preliminaries

In geometry, the *convex hull* of a dataset D, denoted by $CH(D)$, is the smallest convex set containing all points in D [9]. 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 1 and its corresponding $conv(D)$ visualized in Figure 1. In this example, $conv(D) = \{p_1, p_2, p_3, p_4, p_5\}$.

One important conclusion is that the top-1 point must be in $conv(D)$, 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 $conv(D)$.

4.2 Checking Subroutine

Before we start to introduce our algorithm, we first introduce a checking subroutine, 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 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 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 displayed to the user and look different from the 2 original points, resulting in a question involving these new points which looks different from the original question involving the 2 original points.

In our algorithm, before invoking the checking subroutine on point p_i and p_j, (i.e., $check(p_i, p_j, k)$), the two points may have already been presented to the user once. This can also be considered as a round of checking (since the user also provided his/her preference between p_i and p_j), so instead of initializing both $select\_p_i$ and $select\_p_j$ to 0, we increment the corresponding counter by 1. For example, if the user previously indicated that p_i is preferred to p_j, then $select\_p_i$ is initialized to 1. By doing so, we can ask one fewer question for the checking subroutine, meanwhile not hurting the correctness of the algorithm.

Since the user makes random errors with probability at most θ, even this checking subroutine can fail to reveal the real relationship between p_i and p_j. Without loss of generality, assume that p_i is preferred to p_j. Let P_k denote the probability that a checking with k questions asked fails, that is, $P_k = P(Check(p_i, p_j, k) = p_j)$. Then, P_k can be computed as follows: $P_k \leq 1 - \sum_{t=0}^{\lfloor \frac{k}{2} \rfloor} \binom{k}{t} \theta^t (1-\theta)^{k-t}$.

One can observe that P_k monotonically decreases when k increases. A natural question would be how to efficiently determine the value of k if we want $P_k \leq \alpha$ where α is a desired failure probability. When α is relatively small (e.g., 0.00001), P_k can be approximated using a Gaussian function [22] and Lemma 1 can be used to compute the value of k. On the other hand, when α is relatively large (e.g., 0.01), the Gaussian approximation is not very tight, and thus we can apply a binary search on k to find the smallest value such that $P_k \leq \alpha$.

Lemma 1. *Given a user error rate θ and a desired failure probability α, the checking subroutine fails with probability at most α by setting* $k = \Theta(\frac{4\theta(1-\theta)}{(1-2\theta)^2} \log(\frac{1}{\alpha}))$.

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.

4.3 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 set to 1 and n initially, 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 the search range contains only one element.

Before we describe our algorithm *2RI*, we need to describe three concepts. The first concept is the increasing-and-decreasing trend of the utility values over the sorted list as shown in the following lemma.

Lemma 2. *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, $\forall h \leq i < j \leq n, u \cdot p_i > u \cdot p_j$.*

Proof. We first prove that $\forall 1 \leq i < j \leq h, u \cdot p_i < u \cdot p_j$. The other half can be proved symmetrically. Since p_h is the max-utility point, we have $u \cdot p_h - u \cdot p_{h-1} = u \cdot (p_h - p_{h-1}) \geq 0$. By the nature of convex hull in $2d$, for all i, j where $0 \leq i < j \leq h$, we must also have $u \cdot (p_j - p_i) = u \cdot p_j - u \cdot p_i > 0$, which concludes the proof. □

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 lower bound is correct, that is, if $L' \leq h$. The case of the upper bound 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 lower bound is trivially correct). According to Lemma 2, if $p_{L'}$ is preferred, then we can conclude that $L' \leq h$. Otherwise, the lower bound 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 Lemma 2, 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 4 in our Appendix. 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 $r = \lceil \frac{U'+L'}{2} \rceil$. Depending on the user's answer, by Lemma 2, the search range could be shrunk accordingly. This technique is similar to [16, 25] 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 lower bound L' or the upper bound 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 upper bound and the lower bound of the updated search range is 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 case is a general case and the second case 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 very low (which could be explained by Lemma 4 (in the appendix)).
  - 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 routine. 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

algorithm	Number of Rounds
Verify-Point	$O(\frac{c}{1-\theta}\log n(conj + \log\log n + \log\frac{1}{\delta}))$
Verify-Space	$O(\frac{d}{1-2\theta}\log n(conj + \log\log n + \log\frac{1}{\delta}))$

Table 4: Performance of Multi-dimensional Algorithms

shifts the search interval to $[2, 3]$, and enters a new iteration with this new range $[2, 3]$.

Theorem 2 presents the main results on *2RI*. Corollary 1 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 top-1 point with probability at least $1-\delta$ using $O(\frac{1}{-\log 2\theta}(\log n)\log\frac{\log n}{\delta})$ rounds on expectation.*

Proof sketch. We first prove that, the expected number of iterations is $O(\log_{\frac{1}{2\theta}} n)$. Then, 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 $1 - O(P_k \log_{\frac{1}{2\theta}} n)$. Setting $\delta = O(P_k \log_{\frac{1}{2\theta}} n)$ and applying Lemma 1 yields the theorem. □

Corollary 1. *2RI is asymptotically optimal.*

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

5 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 [16] and estimate the utility value of each dimension one by one through constructing 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 $d \geq 2$. The complexities of the number of questions asked by these two algorithms (also called *round complexities*) are summarized in Table 4. Here, n is the input size, δ is the failure probability of retrieving the top-1 point, $conj$ is the round complexity of *Conjecture Phase* of each of the two algorithms (to be introduced later), d is the dimensionality, and c is a data-dependent variable, which can be regarded as a small constant according to our experimental results. In our experimental results, c is at most 3. Both algorithms enjoy provable theoretical guarantee in terms of the success probability of top-1 point retrieval and 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 5.1, and then introduce the first phase, called *Conjecture Phase*, of the two algorithms in Section 5.2. We then describe the second phase, called *Verification Phase*, of *Verify-Point* and *Verify-Space* in Section 5.3.1 and 5.3.2 respectively.

5.1 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* [9] 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* [9]. 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 [25], 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 [10, 23, 25] 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.

The general framework of the algorithms works as follows. It runs for several *iterations* and each iteration consist 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 top-1 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 *top-1 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* top-1 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 top-1 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 top-1 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 back

to Conjecture Phase. It alternates between Conjecture Phase and Verification Phase until all but one point is pruned out for further consideration. Then, we return this point as the final answer.

5.2 Conjecture Phase

Given a utility range R' indicating a convex region where the utility vector u lies, and a set C' containing some possible top-1 point, the purpose of Conjecture Phase is to locate the user's "top-1" candidate point 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 pairwise comparison-based interactive algorithms considering no user errors. Here, we adapt from the algorithm *HD-PI* [23] since it has the highest round-efficiency in finding the top-1 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 $O(conj)$. When adapting *HD-PI* as 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'. Even though the worst case has a linear complexity, in our experiment, Conjecture Phase is very efficient.

5.3 Verification Phase

When there is no user error, p_c found in Conjecture Phase would be the true top-1 point. However, since the user may make mistakes, the result from Conjecture Phase may no longer be the true top-1 point. Therefore, Verification Phase is applied to make sure that the true top-1 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.

5.3.1 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 *top-1 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 2. 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 2, 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 top-1 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 5.2 (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 verify with the user using the checking subroutine developed in Section 4.2. 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. We can easily verify that after the above steps, the updated table L correctly stores the relations between hyperplanes and partitions within the updated utility range R.
  - 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 top-1, 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 (because 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.

As a running example, 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 top-1 candidate set C is $\{p_1, p_2, p_3, p_4, p_5\}$ and table L is shown in Table 2. 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 3.

The main theorem of *Verify-Point* is presented in Theorem 3.

Theorem 3. *Given the input size n, the error rate θ and a failure probability δ, Verify-Point returns the top-1 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 and $conj$ is the round complexity of Conjecture Phase.*

Proof sketch. We first show that in the worst case, the number of iterations is $O(\frac{c}{1-\theta}\log n)$ and each iteration consists of $O(conj + k)$ rounds where k is the number of rounds in a checking subroutine. Therefore, the total number of rounds is $O(\frac{c}{1-\theta}\log n)\cdot O(conj + k)$ and the total success probability is at least $1 - O(\frac{cP_k}{1-\theta}\log n)$. Lemma 1 shows that to let $\delta = O(\frac{cP_k}{1-\theta}\log n)$, it suffices to set $k = O(\log\log n + \log\frac{1}{\delta})$, which completes the proof. □

5.3.2 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 find 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 [6] 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* [7] to sample N points in R and use the sampled points to compute this ratio. Lemma 3 shows that sampling a small number of points suffices to compute all ratios with high accuracy.

Lemma 3. *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 $s \subseteq \mathbb{R}^d$, $\frac{vol(s\cap R)}{vol(R)} - \frac{|\{t\in T | t\in s\cap R\}|}{|\{t\in T | t\in R\}|} \le \varepsilon$.*

The above lemma is derived directly from Theorem 5 in [13]. 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 $s_2 \cap R$.

The main conclusion of *Verify-Space* is presented in Theorem 4.

Theorem 4. *Given the input size n, the error rate θ and a failure probability δ, Verify-Space returns the top-1 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.*

Proof sketch. We prove a special case where we set $\beta_2 = 0.5$, although in practice, β_2 can be set to another value to further improve the performance. We first prove that in the worst case, the expected number of iterations is $O(\frac{d}{1-2\theta} \log n)$, and each iteration uses $O(conj + k)$ rounds where k is the number of comparisons in the checking subroutine. The total success probability is $1 - O(\frac{d}{1-2\theta} \log nP_k)$ using union bound. Lemma 1 shows that to let $\delta = O(\frac{d}{1-2\theta} \log nP_k)$, it suffices to set $k = O(\log\log n + \log \frac{1}{\delta})$, which completes the proof. □

6 EXPERIMENT

We conducted experiments on a computer with 1.80 GHz CPU and 12 GB RAM. All programs were implemented in C/C++.

Datasets. The experiments were conducted on synthetic and real datasets that are used in [16, 23, 25]. Specifically, we generated *anti-correlated* datasets by a dataset generator developed for skyline operators [5]. Besides, we used 4 real datasets, namely *Island*, *Weather*, *Car* and *NBA*. *Island* is two-dimensional and contains 63,383 geographic locations. *Weather* includes 178,080 tuples described by 4 attributes. *Car* is a 4-dimensional dataset consisting of 68,005 used cars after cars whose attribute values fall out of a normal range are filtered out. *NBA* involves six attributes and contains 16,916 players after deleting records with missing values. Each dimension is normalized into range $[0, 1]$. Following the existing studies [23, 25], we preprocessed all the datasets to contain only the skyline points (which are all the possible top-1 point for any utility function), since we are only interested in the top-1 point.

Algorithms. We evaluated our 2-dimensional algorithm: *2RI* and two multi-dimensional algorithms: *Verify-Space* and *Verify-Point*. The competitor algorithms are: *Median* [25], *Hull* [25], *2D-PI* [23], *Active-Ranking* [10], *Preference-Learning* [19], *UtilityApprox* [16], *UH-Simplex* [25], *HD-PI* [23] and *RH* [23]. Since some of them cannot return the top-1 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 top-1 point so they are left unchanged. *2D-PI* returns one of top-k points where k is a user parameter, so we set k to 1 such that it returns the top-1 point. (2) Algorithm *Active-ranking* aims at learning the entire ranking of all points by interacting with the user. We return the top-1 point after the full ranking is determined. (3) Algorithm *Preference-learning* focuses on learning the utility vector of the user. According to the experiment result in [19], 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 top-1 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 [23], 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 top-1 point. (5) Algorithms *HD-PI* and *RH*, similar to the 2-d algorithm *2DPI*, aim at returning one of the top-k points. We ensure they return the top-1 point by setting $k = 1$.

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 3, and (6) the parameter k in the checking subroutine. Unless stated explicitly, for each synthetic dataset, we set $N = 10,000$ and $d = 4$. We set $\theta = 0.05$, which is a reasonable error rate upper bound according to the human reliability assessment data in [11]. According to the experimental results in Section 6.1, 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 top-1 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 top-1 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 6.1, we study the effect of different parameter settings of our algorithms. The performance of all algorithms on synthetic and real datasets are discussed in Section 6.2 and 6.3, respectively. We conducted a user study on a used car purchasing scenario in Section 6.4 to demonstrate the effectiveness of our algorithms when user errors are considered in a real life situation. Finally, we summarize the experiments in Section 6.5.

6.1 Study on Parameter Settings

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 value of parameter c in Theorem 3.

In Figure 5, we studied the effect of ε on *Verify-Space*. When varying ε between 0.05 and 0.25, we observe no significant change in terms of the number of questions and the accuracies from Figure 5 (a) and (b), respectively. However, in Figure 5 (a), when $\varepsilon > 0.15$, *Verify-Space* becomes less stable since the standard deviation of the number of questions asked by the algorithm increases, which can be attributed to the less accurate volume ratio estimation. Moreover,

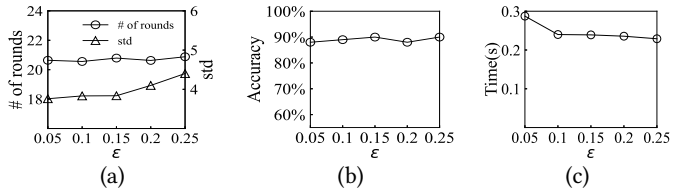

Figure 5: Effect of ε

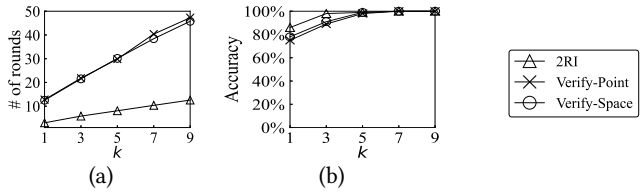

Figure 6: Effect of k

in Figure 5 (c), when $\varepsilon < 0.1$, the average processing time increases due to the increase in sample size. We decided to set $\varepsilon = 0.1$ since compared with $\varepsilon = 0.15$, it asks fewer questions.

We studied the effect of β_1 on *Verify-Point* and β_2 on *Verify-Space* when ranging them from 0 to 0.5. The related figures can be found in the appendix. Both algorithms use the least number of rounds when $\beta_1 = \beta_2 = 0.2$. On the other hand, changing β does not have obvious impact on the accuracy of finding the top-1 point. Therefore, in the rest of our experiments, we set $\beta_1 = \beta_2 = 0.2$.

In Figure 6, we evaluated the effect of different choices of k. The experiments of *2RI* were conducted on 2-d synthetic datasets and the experiments of *Verify-Point* and *Verify-Space* were conducted on 4-d synthetic datasets. Since the checking subroutine needs to make decisions based on the majority of answers, k only takes odd values. We observe that the number of rounds and the accuracy both increase with k. Specifically, when $k \geq 5$, the accuracies of all algorithms are very close to 1. But, the trade-off is that an additional number of questions needs to be asked. Therefore, we choose $k = 3$ 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 value of c shown in Theorem 3 on different datasets. Specifically, we evaluated c on 3 real datasets, namely *Car*, *NBA* and *Weather*, and 3 synthetic datasets, namely *Anti*, *Corr* and *Indep*, which were generated using the dataset generator developed in [5], consisting of anti-correlated data, correlated data and independent data with size 100,000, respectively. The results can be found in the appendix. For all the datasets we tested, the value of c never exceeds 3. Therefore, c can be regarded as a small constant.

6.2 Performance on Synthetic Datasets

We compared our two-dimensional algorithm *2RI* against *2DPI*, *Median* and *Hull* on a 2-dimensional synthetic dataset. For completeness, we also record the performance of all d-dimensional algorithms. Figure 7 presents the performance of all the algorithms when N varies from 100 to 1,000,000. According to Figure 7 (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 7 (b), *2RI* achieves the highest accuracy among all the algorithms under all settings, and finishes within only 5-6 questions as shown in Figure 7 (a). On the other hand, algorithm *2DPI*, *Median* and *Hull* cannot efficiently handle user errors and their top-1 retrieval accuracies is at least 10 percentage smaller than *2RI*. It is worth mentioning that a lower accuracy could lead to unforgettable and unchangeable consequences as described in Section 1.

In Figure 8, we evaluated the performance of our algorithms *Verify-Point* and *Verify-Space* against other existing d-dimensional algorithms on 4-dimensional synthetic datasets. We observe that *Verify-Point* and *Verify-Space* are the most accurate on finding the top-1 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. *HDPI* 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 handle user errors, their performance is poor in terms of both the number of questions asked and the top-1 retrieval accuracy. According to Figure 8 (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 9 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 increase 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 10 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 3-4 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 1 second, which is an acceptable interactive speed.

We measured the percentage of executions of *Verify-Point* and *Verify-Space* that finish with different numbers of iterations on $3d$, $4d$ and $5d$ synthetic datasets. Due to the lack of space, the related figures are put in the appendix. All executions finish within 3 iterations and there is no significant difference between these two algorithms. When dimensionality increases, the percentage of executions that finish with more than 1 iteration also slightly increases. For example, when $d = 3$, about 85% of executions finish within 1 iteration, only 15% use 2 iterations, and less than 1% use 3 iterations to finish. On the other hand, when $d = 5$, the percentage of executions that take 1, 2 and 3 iterations are about 73%, 24%

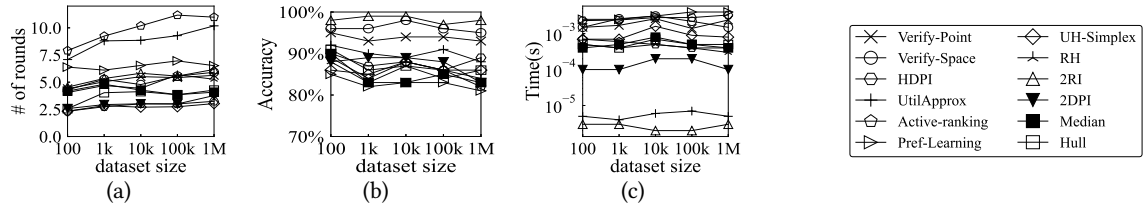

Figure 7: 2d dataset

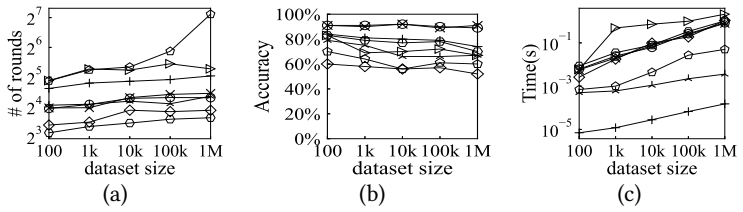

Figure 8: 4d dataset

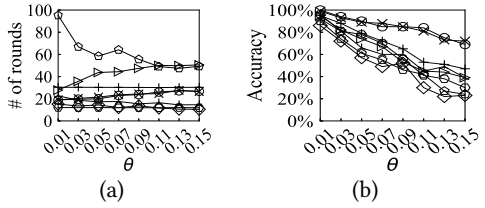

Figure 9: Effect of θ

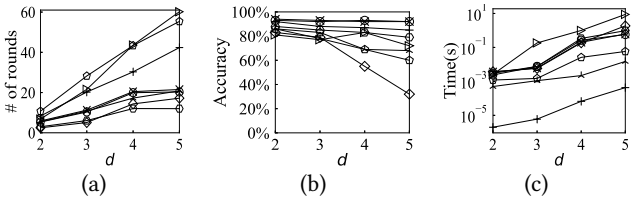

Figure 10: Effect of d

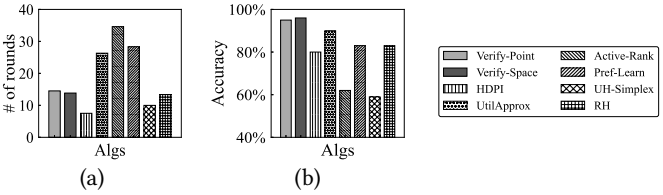

Figure 11: Results on dataset *Weather*

and 3%, respectively. One possible explanation is that when the dimensionality increases, an error in Conjecture Phase will have a higher impact due to more complex spatial structure, making the algorithm more likely to focus on the wrong region. As a consequence, an additional iteration must take place to perform a new round of searching.

We also studied the average number of questions in each iteration and each phase (i.e., Conjecture Phase and Verification Phase) on these 3 synthetic datasets. The results can be found in the appendix. When d increases, the ratio between the number of questions asked in Verification Phase and the number of questions asked in Conjecture Phase also increases. For example, when $d = 3$, this ratio is around 0.47, but when $d = 5$, this ratio grows to around 0.54. This is explainable since in Verification Phase, we verify the correctness of each subspace $s \in S$, where S contains the subspaces indicated by the user in Conjecture Phase. Recall that after checking some subspaces $s \in S$ for in the first several rounds, some other subspaces $s \in S$ may be removed from S and need not to be verified again since they contain the entire utility range and are no long useful for updating it. However, it may be harder to remove the subspaces in higher dimensions, resulting in more subspaces needed to be checked in Verification Phase.

6.3 Performance on Real Datasets

We conducted experiments on 4 real datasets, namely *Island*, *Car*, *Weather* and *NBA*. For four 2-dimensional algorithms *2RI*, *2DPI*, *Median* and *Hull*, their performance on the 2-d dataset *Island* is reported. For completeness, the performance of all 8 d-dimensional approaches are also reported. Due to the space constraint, the experimental results can be found in the appendix. Among all 2-d algorithms, *2RI* requires slightly more than 5 questions, and obtains the highest accuracy which is around 97%. Other competitors take 1-2 less questions compared with *2RI*, but their accuracies are much lower. Specifically, *2DPI* and *Median* obtain accuracies below 90%, and *Hull* is less than 80%.

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 11. The performance on *Island*, *Car* and *NBA* can be found in the appendix. *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.

We also studied the percentage of executions of *Verify-Point* and *Verify-Space* that finish with different numbers of iterations, as well as the number of questions in each phase, on all 4 real datasets. The performance of our algorithms on real datasets is consistent with synthetic datasets, and the results of these experiments could be found in the appendix due to space constraint.

6.4 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. Following the same settings in [19, 23, 25], 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 answer of some randomly selected questions with the prediction w.r.t. u. If 75% questions [19] can be correctly predicted, we stop and return the top-1 point w.r.t. u. (2) For *UH-Simplex*, we set the threshold $\epsilon = 0$ and this guarantees that the returned car is believed to be the top-1 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%.

Figure 12 (a) and (b) shows the average number of questions asked and the average dissatisfaction score of each algorithm. 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, *HDPI* 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.

In Figure 13 (a) and (b), we compared the user's preference on the recommendations of our algorithms *Verify-Point* and *Verify-Space*, respectively, against *HDPI*, *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 13 (a), when *Verify-Point* is compared with *HDPI*, 12 users favor the car recommended by *Verify-Point*, while only 4 think the recommendation of *HDPI* 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*.

We studied the percentage of *Verify-Point* and *Verify-Space* that finished on each iteration in user study. The figures for these experiments can be found in the appendix. In particular, almost 90% of users finished within only 1 iteration on *Verify-Point* and *Verify-Space*, and 10% of users finished in 2 iterations. Only 3% of users (i.e., one user) ended up with 3 iterations when using *Verify-Point*. Compared with the experimental results on multi-dimensional synthetic and real datasets, the percentage of users who need more than 1 iteration is smaller in our user study, probably because the user error rate in real world (as found by us as 4.5%) is even smaller than our default setting of θ (i.e., 5%).

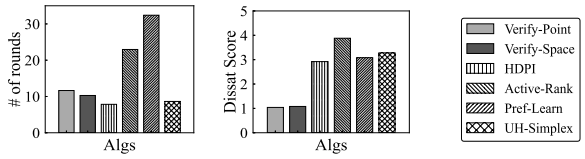

Figure 12: User study showing the no. of rounds and dissatisfaction scores

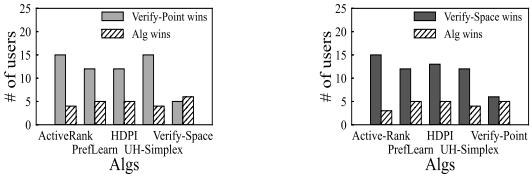

Figure 13: User study comparing whether Verify-Point/Verify-Space is better than each of the competitors

6.5 Summary

The experiments demonstrated the superiority of our 2-dimensional algorithm *2RI* and our d-dimensional algorithms, *Verify-Point* and *Verify-Space*, over the existing approaches: (1) we are efficient and effective. In particular, for all 2-dimensional experiments, *2RI* consistently outperforms all other algorithms and achieves an accuracy close to 1 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. Moreover, all three algorithms finish in a small number of rounds. (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 6-dimensional dataset *NBA*, 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 40 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%.

7 CONCLUSION

In this paper, we propose a more robust interactive model that returns the top-1 point in dataset under the setting of random errors. Our model is more practical than existing algorithms in a sense that ours can return top-1 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 top-1 point facing user errors compared with existing algorithms. In the future, we consider the case where user makes persistent errors when answering questions.

REFERENCES

[1] Yongkil Ahn. 2019. 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] Ilaria Bartolini, Paolo Ciaccia, Vincent Oria, and M Tamer Özsu. 2007. Flexible integration of multimedia sub-queries with qualitative preferences. *Multimedia Tools and Applications* 33, 3 (2007), 275–300.

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

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

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

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

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

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

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

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

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

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

[13] Yi Li, Philip M Long, and Aravind Srinivasan. 2001. Improved bounds on the sample complexity of learning. *J. Comput. System Sci.* 62, 3 (2001), 516–527.

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

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

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

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

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

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

[20] Gerard Salton. 1989. Automatic text processing: The transformation, analysis, and retrieval of. *Reading: Addison-Wesley* 169 (1989).

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

[22] Christian Walck et al. 2007. Hand-book on statistical distributions for experimentalists. *University of Stockholm* 10 (2007), 96–01.

[23] Weicheng Wang, Raymond Chi-Wing Wong, and Min Xie. 2021. Interactive Search for One of the Top-k. In *Proceedings of the 2021 International Conference on Management of Data*. 1920–1932.

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

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

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

A ADDITIONAL EXPERIMENTS

A.1 Additional experiments on synthetic datasets

In this section, we supplement the figures about the experimental results on synthetic datasets that are not shown in the main paper. Figure 14 measures the percentage of executions of *Verify-Point* and *Verify-Space* that finish with different numbers of iterations on 3d, 4d and 5d synthetic datasets. In this figure, the bar at iteration number i shows how many percentage of executions terminate with i iterations. We only show the results on iteration number 1, 2 and 3 since all executions finish within 3 iterations. Our discussions on this figure can be found in the main paper.

Figure 15 shows the average number of questions in each iteration and each phase (i.e., Conjecture Phase and Verification Phase) on these 3 synthetic datasets. Similarly, we only show the results in iteration 1, 2 and 3 since all executions finish within 3 iterations. In these figures, "Conj" represents Conjecture Phase and "Veri" represents Verification Phase. The discussion on these results can be found in the main paper.

A.2 Additional experiments on real datasets

In this section, we present the experiment results on real datasets that are not shown in the main body of the paper. In Figure 16, we show the results of our 2-d algorithms *2RI* with other 2-d competitors. For completeness, the results of other multi-dimensional algorithms are also reported. The performance of 2-d algorithms is discussed in the main paper. In Figure 17 and Figure 18, we compare our proposed algorithms, namely *Verify-Point* and *Verify-Space*, with other d-dimensional algorithms on real datasets *Car* and *NBA*, respectively. The results on these datasets are consistent with the results on *Weather*. Specifically, our algorithms obtain the highest accuracy among all competitors by only asking a small number of questions. Algorithm *HDPI* and *UH-Simplex* uses less number of questions than our algorithms. However, from Figure 17 (b) and Figure 18 (b), their accuracy is at least 10% lower than ours on both datasets.

Figure 19 shows the percentage of executions of *Verify-Point* and *Verify-Space* that finish with different numbers of iterations on all 4 real datasets, namely *Island*, *Car*, *Weather* and *NBA*. The bar at iteration i indicates the percentage of executions that terminate using i iterations. Since all the executions finished within 3 iterations, we only show the iteration number 1, 2 and 3 here. The results on these real dataset agrees with our analysis on synthetic datasets.

Figure 20 measures the average number of questions in each iteration and each phase (i.e., Conjecture Phase and Verification Phase) on all 4 real datasets. Similar to synthetic datasets, in these figures, "Conj" represents Conjecture Phase and "Veri" represents Verification Phase. Since all the executions finished within 3 iterations, we only show the results in iteration 1, 2 and 3. The experimental results on these real dataset are consistent with our analysis on synthetic datasets.

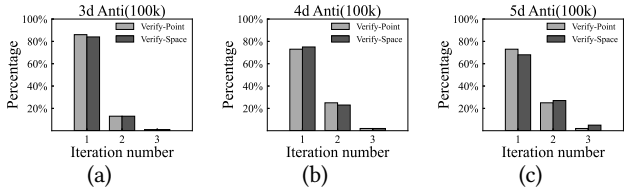

Figure 14: Percentage of iterations (synthetic datasets)

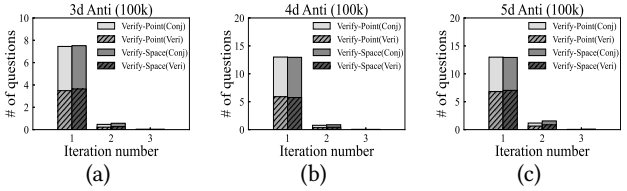

Figure 15: Average no. of questions in each phase (synthetic datasets)

A.3 Additional experiments on user study

Figure 21 presents the percentage of *Verify-Point* and *Verify-Space* that finished on each iteration in user study. An analysis to the figure is provided in Section 6.4.

A.4 Additional experiments on parameter settings

Figure 22 shows the effect of $\beta 1$ on *Verify-Point* and the effect of $\beta 2$ on *Verify-Space* when ranging them from 0 to 0.5. Since both algorithms take the least number of questions when $\beta 1 = \beta 2 = 0.2$, we set $\beta 1 = \beta 2 = 0.2$ in our other experiments.

Figure 23 shows the value of c in Theroem 3 on 3 synthetic datasets and 3 real datasets, as described in Section 6.1. Since the value of c never exceeds 3, it can be regarded as a small constant in practice.

B RELATED PROOFS

B.1 Proof of Theorem 1

Consider a 2-dimensional dataset D. The search of the top-1 point using questions in the form " Is point p_i preferred to point p_j?" can be modeled by a binary tree with each leaf corresponding to one point in $conv(D)$. Based on p_i or p_j is selected, we follow different branches of the tree. Note that when an user error occurs, we fall into the wrong subtree and cannot return to the correct track unless the error is handled. In other words, the top-1 point is found if and only if all errors occurred are handled sequentially.

Let n denote the number of points in $conv(D)$ (i.e., $n = |conv(D)|$). In order to find the top-1 point, $\Omega(\log n)$ questions must be asked since we must traverse $\Omega(\log n)$ steps in the tree. Since each question has probability θ to receive a wrong answer, the expected number of errors that will occur along the correct path of the entire search is $m = \Omega(\theta \log n)$.

As a running example, assume now the relation between two points, namely p_i and p_j, is $u \cdot p_i > u \cdot p_j$, that is, p_i should be preferred to p_j. However, due to a random user error, the user indicates that p_j is preferred to p_i. Since we cannot tell if the user just made an error by only looking at the answer of this question (i.e., we cannot deduce from other sources if p_i or p_j is preferred), we can only spend k questions on these two points, namely p_i and p_j, to verify their relationships and try to correct this error. When k is large, we can approximate the number of correct answers in all the k answers by a Gaussian distribution with mean $\mu = (1-\theta)k$ and variance $\sigma^2 = \theta(1-\theta)k$. To correct this error, at least $\frac{k}{2}$ of the questions must be answered correctly. Now assume that we want each error is corrected with probability at least $1-\alpha$, where α is a number to be determined later. Let z_α denote the α-quantile of $N(0,1)$, by solving $\frac{\frac{k}{2}-\mu}{\sigma} \le z_\alpha$, we obtain $k \ge \frac{4\theta(1-\theta)}{(1-2\theta)^2} z_\alpha^2$. Since $z_\alpha = \Theta(\sqrt{\log(\frac{1}{\alpha})})$, $k = \Theta(\frac{4\theta(1-\theta)}{(1-2\theta)^2}\log(\frac{1}{\alpha}))$. (Note that this also leads to the conclusion of Lemma 1).

If we want the total failure probability to be less than δ (i.e. the probability of failing to return the top-1 point is less than δ) with the smallest number of rounds, we need to distribute the failure probability evenly to each error. Recall that the number of errors is $m = \Omega(\theta \log n)$. Since all errors must be handled correctly, it requires that $(1-\alpha)^m \ge 1-\delta$, which yields $\frac{\delta}{m+1} \le \alpha \le \frac{\delta}{m}$. Summing these up, we obtain the total amount of questions required is $\Omega(\log n) + \Omega(mk) = \Omega(\log n) + \Omega(\frac{4\theta^2(1-\theta)}{(1-2\theta)^2}(\log n)\log(\frac{\log n}{\delta})) = \Omega(\frac{4\theta^2(1-\theta)}{(1-2\theta)^2}(\log n)\log(\frac{\log n}{\delta}))$.

B.2 Proof of Lemma 1

The proof of Lemma 1 can be found as a component of the proof of Theorem 1.

B.3 Lemma 4

Definition 1. *Given a range $R = [L, U]$ where L (resp. U) is the lower (resp. upper) bound of the range and a point p, 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 4. *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 top-1 point. Then, $P(Dist(p_h, R') \ge \lceil 2m\theta \rceil) \le \frac{1}{m\theta}$.* □

We now prove the correctness of Lemma 4.

Let $m = U - L + 1$ denote the size of the search range $R = [L, U]$, and θ denote the upper bound of user error rate. Further, denote p_h the real top-1 point and $R' = [L', U']$ the search range obtained just after Step 1 of *2RI*. We first prove that $E[Dist(p_h, R')] \le m\theta$.

The binary search terminates in $\log m$ rounds, regardless of the correctness of the user's choice in each round. Therefore, we can model an execution of the algorithm as a sequence $S \in \{0,1\}^{\log m}$, where $S_i = 1$ means that the i-th question is answered correctly and $S_i = 0$ otherwise. Now consider that we attribute $Dist(p_h, R')$ to each of the $\log m$ questions. Initially, $Dist(p_h, R') = 0$. Let D_i denote the increase of $Dist(p_h, R')$ "caused" by answering the i-th question, i.e., $Dist(p_h, R') = \sum_{i=0}^{\log m - 1} D_i$. Note that if $S_i = 1$, i.e., the i-the questions is answered correctly, then $D_i = 0$. If $S_i = 0$, indicating an error when answering the i-th questions, $Dist(p_h, R')$ can increase by at most $m/2^{i+1}$. For example, if the user is wrong when answering the first question but correctly answers the remaining questions, $Dist(p_h, R')$ is at most $\frac{m}{2}$. Since $P(S_i = 0) \le \theta$, $E[D_i] \le \frac{m\theta}{2^{i+1}}$, therefore, $E[Dist(p_h, R')] = \sum_{i=0}^{\log m - 1} E[D_i] \le m\theta$.

Since D_i are independent and $\sigma^2_{D_i} = \frac{m}{2^{i+1}}\theta(1-\theta)$, we have $\sigma^2_{Dist(p_h,R')} = \sum \sigma^2_{D_i} = m\theta(1-\theta)$. Applying Chebyshev inequality, we obtain $P(|Dist(p_h, R') - E[Dist(p_h, R')]| > m\theta) \le \frac{1-\theta}{m\theta} \le \frac{1}{m\theta}$, which leads directly to the Lemma.

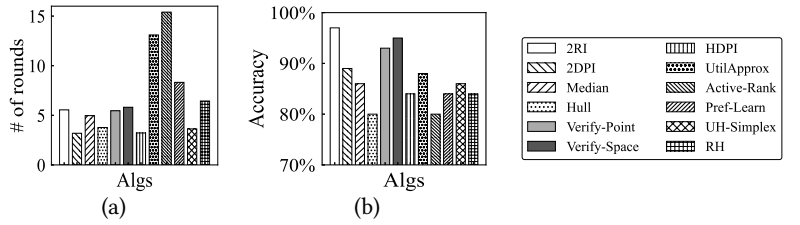

Figure 16: Results on dataset *Island*

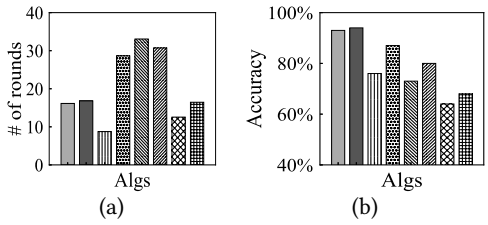

Figure 17: Results on dataset *Car*

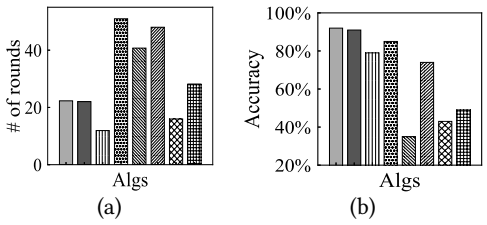

Figure 18: Results on dataset *NBA*

B.4 Proof of Theorem 2

When the context is clear, we use the index of point to denote the actual point (e.g., we use i to denote p_i which is the point at index i of the input array). Denote n the size of input (i.e., $n = |conv(D)|$) and m the size of search range at the beginning of an iteration, i.e., $m = U - L + 1$ where U(resp. L) is the upper (resp, lower) bound of the search range. We run binary search on range $[L, U]$ until the range is reduced to $R' = [L', U']$ such that $U' - L' + 1 \leq \lceil 2m\theta \rceil$. We then check if the real top-1 point $p_h \in R'$. If not, we check if $Dist(p_h, R') \leq \lceil 2m\theta \rceil$. Based on the two checking results, there are 3 cases: (1), $p_h \in R'$ (or $Dist(p_h, R') = 0$), (2) $0 < Dist(p_h, R') \leq \lceil 2m\theta \rceil$, and (3) $Dist(p_h, R') > \lceil 2m\theta \rceil$. Note that the special case (i.e., Case (2) described in Step (2) of *2RI*) also belongs to the second case here. We say that an iteration is "good" if the checking result is in case (1) or (2) since in these two cases, we can update the search range with size $\leq \lceil 2m\theta \rceil$. Applying Lemma 4, when $m\theta \geq 2$, the expected number of iterations from the current good iteration to the next good iterations is $\frac{1}{P(Dist(p_h,R') \leq \lceil 2m\theta \rceil)} \leq \frac{1}{P(Dist(p_h,R') \leq 2m\theta)} \leq \frac{1}{1-\frac{1}{m\theta}} \leq 1 + \frac{2}{m\theta}$. Also note that since $E[Dist(p_h, R')] \leq m\theta$, when $m\theta < 2$, $P(Dist(p_h, R') > \lceil 2m\theta \rceil) \leq P(Dist(p_h, R') > 2m\theta) \leq \frac{1}{2}$ by using Markov inequality, and the expected number of iterations from the current good iteration to the next good iterations is at most 2.

We now bound the expected total iterations required. Let the first good iterations where $m\theta < 2$ be the g-th good iteration (i.e., at good iteration $g-1$, $m\theta \geq 2$, and at good iteration g, $m\theta < 2$). Note that since $\theta < 0.5$, when $m\theta \geq 2$, $m \geq 4$. Since each good iteration reduces the search range size by at least a factor of $\Theta(2\theta)$, $g \leq \Theta(\log_{\frac{1}{2\theta}} \frac{n}{4})$. Denote Y the expected total number of iterations and Y_i the number of iterations between the $(i-1)$-th and the i-th good iterations, then

$$E[Y_i] \leq \begin{cases} 1 + \frac{2}{m_i\theta} & i < g \\ 2 & i \geq g \end{cases}$$

where m_i is the size of search range (i.e., $U-L+1$ in the algorithm) at the i-th good iteration. We know that $m_i\theta \geq 2 \cdot (\frac{1}{2\theta})^{g-i-1}$ since $m_{g-1}\theta \geq 2$ and $m_i \geq (\frac{1}{2\theta})^{g-i-1} m_{g-1}$. So when $i < g$, $E[Y_i] = 1 + \frac{2}{m_i\theta} \leq 1 + (2\theta)^{g-i-1}$. Therefore, $E[Y] = \sum_{i<g} E[Y_i] + \sum_{i \geq g} E[Y_i] \leq O(\log_{\frac{1}{2\theta}} n) + O(\log_{\frac{1}{2\theta}} 1) = O(\log_{\frac{1}{2\theta}} n)$.

Since Step 1 of *2RI* stops when the search range size shrinks to $\lceil 2m\theta \rceil$, each iteration uses at most $O(\log \frac{1}{2\theta})$ questions (Step 1) and $3k$ extra questions for checking (Step 2). Therefore, the expected number of rounds is $O(\log_{\frac{1}{2\theta}} n) \cdot (\log \frac{1}{2\theta} + 3k) = O(\frac{k}{-\log 2\theta} \log n)$. The failure probability is upper bounded by taking union bound on the total number of checkings multiplied by P_k (where P_k is the failure probability of a checking subroutine with k rounds), which is $O(\frac{P_k}{-\log 2\theta} \log n)$. Setting $\delta = O(\frac{P_k}{-\log 2\theta} \log n)$ and applying Lemma 1 yields the theorem.

B.5 Proof of Theorem 3

We first prove that, in the worst case, the number of iterations is $O(\frac{c}{1-\theta} \log n)$, where c is the constant appears in Theorem 3. In each round of Verification Phase, we pick the halfspace $s_{i,j} \in S$ that is expected to prune the largest number of partitions. Here, S is the output of Conjecture Phase containing all halfspaces indicated by the user. Let λ denote the average percentage of partitions pruned by the best halfspace in S (i.e., the halfspace with the highest $Num(\cdot)$). Since the utility vector u lies in this halfspace with probability at least $1-\theta$, in each round of Verification Phase, we prune at least λ portion of partitions with probability at least $1-\theta$. Since there is at least 1 round in each Verification Phase, by setting $c = \frac{1}{\log \frac{1}{1-\lambda}}$, the expected number of iterations is $O(\frac{c}{1-\theta} \log n)$.

Verification Phase ends when S is empty, or all the halfspaces in S cannot prune at least β_1 portion of the partitions in the set of all top-1 candidates C. In the worst case, Verification Phase only checks one halfspace before it is stopped, and the size of the utility range R shrinks at the slowest rate. In such a case, each iteration uses $O(conj)$ rounds (Conjecture Phase) plus k rounds (k questions need for the checking performed in Verification Phase), so the total number of rounds is $O(\frac{c}{1-\theta} \log n) \cdot O(conj + k)$. The failure probability is upper bounded by taking union bound on the total number of checkings multiplied by P_k (where P_k is the failure probability of a checking subroutine with k rounds), which is $O(\frac{cP_k}{1-\theta} \log n)$. Lemma 1 shows that by setting the total failure probability $\delta = O(\frac{cP_k}{1-\theta} \log n)$, it suffices to set $k = O(\log \frac{c \log n}{(1-\theta)\delta})$, which completes the proof.

B.6 Lemma 5

Lemma 5. *Let R_s be the utility range at the beginning of Conjecture Phase (i.e., R' just at the beginning of Conjecture Phase) and R_e be*

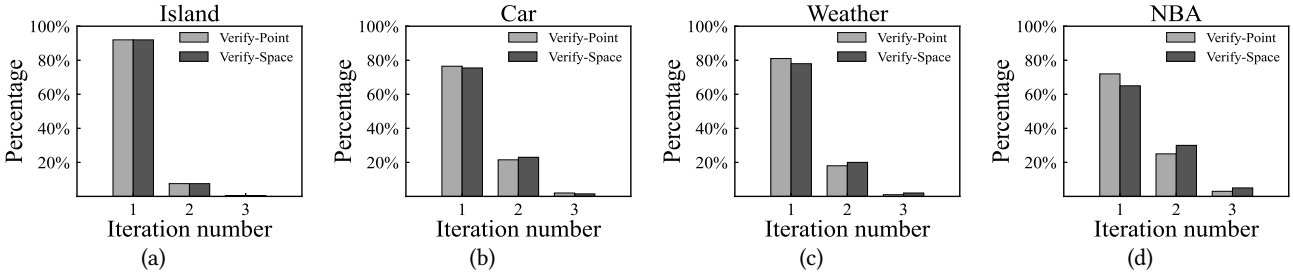

Figure 19: Percentage of iterations (real datasets)

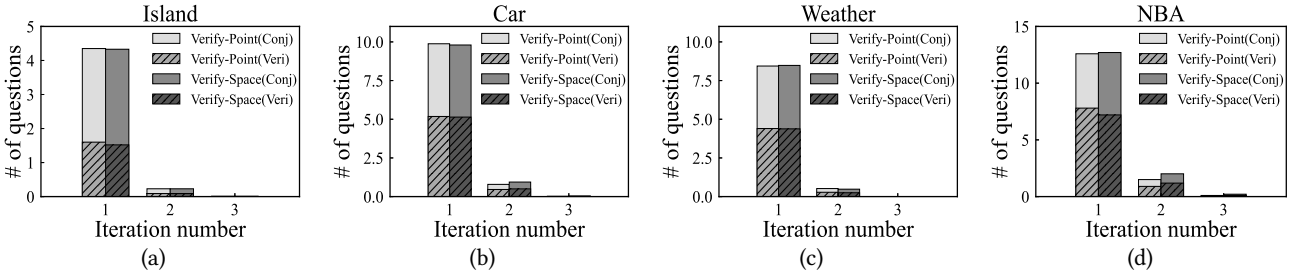

Figure 20: Average no. of questions in each phase (real datasets)

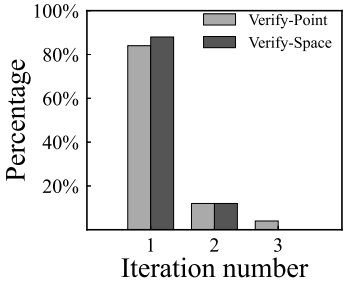

Figure 21: Percentage of iterations (user study)

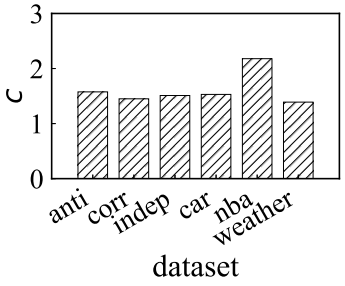

Figure 23: Effect of c

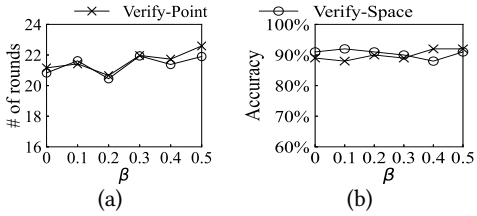

Figure 22: Effect of β

the utility range after Conjecture Phase (i.e., R' just at the end of Conjecture Phase). Let θ be the upper bound of the user error rate. Then, just after Conjecture Phase, the probability that no halfspace in S (i.e., the set containing all halfspaces indicated by the user) can prune at least half of the size of R_s is at most $\frac{vol(R_e)}{vol(R_s)} + \theta$, where $vol(R)$ denotes the size of a utility range R. Formally, $P(\frac{\min_{s \in S} vol(s \cap R_s)}{vol(R_s)} > \frac{1}{2}) \leq \frac{vol(R_e)}{vol(R_s)} + \theta$. □

Proof. Let R_s be the utility range just at the beginning of Conjecture Phase and R_e be the utility range just at the end of Conjecture Phase. For any hyperplane h selected to ask the user in Conjecture Phase, it divides R_s into two halfspaces and at least one halfspace s satisfies $vol(s \cap R_s) \leq \frac{vol(R_s)}{2}$. If at any round in Conjecture Phase the user choose the halfspace s that satisfies $vol(s \cap R_s) \leq \frac{vol(R_s)}{2}$, then the best halfspace in S can reduce the size of R_s by at least half. The only possibility that this would not happen is when the user always picks the larger halfspace until the end of Conjecture Phase.

Consider Figure 24 as a running example. Assume that in Conjecture Phase we asked questions related to h_1, h_2 and h_3. Each hyperplane is related to two halfspaces (e.g., h_1 is related to two halfspaces, namely h_1^+ and h_1^-), where at least one of them can prune the utility range R_s by at least half. These 3 hyperplanes divides R_s into a number of divisions (6 in this example), and each division is the intersection of one of the halfspaces related to each hyperplane. For example, $t_1 = h_1^+ \cap h_2^- \cap h_3^-$. Among these divisions, the only case we cannot find a halfspace that prune the utility range by at least half is when the utility vector u lies in t_c (marked by gray in

the figure). On the other hand, when u falls in t_1, t_2, t_3, t_4 or t_5, at least one of the halfspaces associated with that division can prune R_s by at least half.

Let B_S denote the set of divisions partitioned by the halfspaces in S on the utility range R_s. Let X_t denote the event that the utility vector u lies in division t, and Y_t denote the event that after Conjecture Phase, the algorithm decides that u lies in division t (i.e., $R_e = t$). Note that since there will be user errors, even if $R_e = t$ for some division t, the utility vector u may not lie in division t. Let the only division resulting from always picking the larger subspace be t_c. Our target is to bound $P(Y_{t_c})$.

Note that

$$P(Y_{t_c}) = P(X_{t_c} \cap Y_{t_c}) + \sum_{t \in B_S, t \neq t_c} P(X_t \cap Y_{t_c})$$

Further notice that $X_{t_c} \cap Y_{t_c}$ means that the utility vector u lies in t_c and the resulting utility space $R_e = t_c$, without any distribution information on u, we can assume u is uniformly distributed in the utility range, so $P(X_{t_c} \cap Y_{t_c}) \leq P(X_{t_c}) = \frac{vol(R_e)}{vol(R_s)}$. Also notice that for any $t \neq t_c$, in order to let $X_t \cap Y_{t_c}$ happen, a necessary condition is that the user must make an error on at least 1 specific question, which is the question associated to the hyperplane dividing t and t_c. Therefore, for any $t \neq t_c, P(X_t \cap Y_{t_c}) \leq \theta \cdot P(X_t)$. We conclude that $P(Y_{t_c}) \leq P(X_{t_c}) + \sum_{t \in B_S, t \neq t_c} P(X_t \cap Y_{t_c}) \leq P(X_{t_c}) + \sum_{t \in B_S, t \neq t_c} \theta \cdot P(X_t) \leq P(X_{t_c}) + \theta \sum_{t \in B_S, t \neq t_c} \cdot P(X_t) \leq \frac{vol(R_e)}{vol(R_s)} + \theta$. □

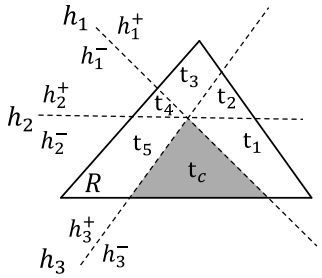

Figure 24: Divisions induced by hyperplanes

B.7 Proof of Theorem 4

Here, we prove a special case of Theorem 4 where we set $\beta_2 = 0.5$, meaning that Verification Phase stops when the "best" halfspace in S cannot prune the utility range R by at least half. Note that in practice, β_2 can be set to another value to further improve the performance.

For any two points p_i and p_j in $conv(D)$, the related hyperplane $h_{i,j}$ cuts the utility space into two subspaces. The collection of all hyperplanes partition the utility space R_0 into many cells, where each cell corresponds to a unique ranking of the points in $conv(D)$. Recall that each iteration consists of one Conjecture Phase and one Verification Phase. Clearly, Conjecture Phase should stop before the utility range copy R' contains 1 cell, and Verification Phase should stop before the utility range R contains only 1 cell, since the top-1 candidate/point must be already determined by then. Here we consider the case where Conjecture Phase ends when there is only one cell left in R', and Verification Phase ends when there is only one cell left in R, or none of the halfspace in S is good enough. Note that in practice, when Conjecture Phase and Verification Phase ends, R' and R may contain multiple cells, and thus, the algorithm will terminate even faster.

[10] proved that the number of possible rankings of n points in a d dimensional space is $\Theta(\frac{n^{2d}}{2^d d!})$, which means there are $O(\frac{n^{2d}}{2^d d!})$ cells in the utility space R_0. We use Σ to denote the collection of all cells in R_0. Since *Verify-Space* ends when R contains only 1 cell, the expected size of R when the algorithm terminates is $\frac{\alpha r_0}{|\Sigma|}$ where r_0 is the size of the utility space R_0 and α is a constant depending on the size distribution of different cells. Note that when $\frac{\alpha r_0}{|\Sigma|}$ is smaller, a larger portion of the utility space is expected to be pruned, and thus, more rounds are required. The worst case happens when all the cells have equal size, or equivalently, $\alpha = 1$. Let r denote the size of R just at the start of a Verification Phase and denote $P(r)$ the probability that the best hyperplane in S, the set containing all halfspaces indicated by the user in Conjecture Phase, fails to cut R of size r by at least half. Formally, $P(r) = P(\min_{s \in S} vol(s \cap R) > \frac{r}{2})$. Since each cell has the same size, there are $\frac{r}{r_0}|\Sigma|$ cells in R and Conjecture Phase ends with only 1 cell left. By Lemma 5, in the first round of this Verification Phase, $P(r) \leq \theta + \frac{r_0}{|\Sigma| r}$.

Assume that at the start of a Verification Phase, the utility range R has size $\frac{r_0}{2^{i+1}} \leq r \leq \frac{r_0}{2^i}$ where $i \in [0, \log|\Sigma|]$. Then, $P(r) = \theta + \frac{r_0}{|\Sigma| r} \leq \theta + \frac{1}{|\Sigma|/2^{i+1}}$. Denote $T(i)$ the expected number of iterations required to reduce r from $(\frac{r_0}{2^{i+1}}, \frac{r_0}{2^i}]$ to $(\frac{r_0}{2^{i+2}}, \frac{r_0}{2^{i+1}}]$, then, $T(i) \leq \frac{1}{1-P(r)} \cdot \frac{1}{1-\theta} \leq \frac{1}{1-(\theta + \frac{1}{|\Sigma|/2^{i+1}})} \cdot \frac{1}{1-\theta}$.

Note that there are 2 cases. Case (1), $\theta + \frac{1}{|\Sigma|/2^{i+1}} \leq \frac{1}{2}$; and Case (2), $\theta + \frac{1}{|\Sigma|/2^{i+1}} > \frac{1}{2}$. In case (1), when $\theta + \frac{1}{|\Sigma|/2^{i+1}} \leq \frac{1}{2}$, $T(i) \leq (1 + 2(\theta + \frac{1}{|\Sigma|/2^{i+1}}))\frac{1}{1-\theta}$. Since $i \in [0, \log|\Sigma|]$, the number of rounds required for this case is $\leq \sum_{i=0}^{\log|\Sigma|} T(i) \leq \frac{1+2\theta}{1-\theta} 2d \log n + O(1)$.

In Case (2), when $\theta + \frac{1}{|\Sigma|/2^{i+1}} > \frac{1}{2}$, this implies that $r \leq \frac{r_0}{2^i} \leq \frac{4r_0}{(1-2\theta)|\Sigma|}$. Since each iteration prunes at least 1 cell from R (since Verification Phase runs for at least 1 round and each round prunes at least 1 cell), the size of R is reduced by at least $\frac{r_0}{|\Sigma|}$. Therefore, for this case, we need at most $O(\frac{1}{(1-2\theta)})$ iterations. Summing both cases, the total expected number of iterations is $O(\frac{d}{1-2\theta} \log n)$.

Typically, Verification Phase performs several rounds of checkings before we return to Conjecture Phase. In the worst case, it only performs one checking and returns to Conjecture Phase since none of the remaining hyperplanes is "good" enough. Note that this is the worst case since R decreases with the slowest rate. In this case, each Conjecture Phase uses $O(conj)$ rounds and each Verification Phase uses k rounds. Since the expected number of iterations is $O(\frac{d}{1-2\theta} \log n)$, the total number of rounds is thus $O(\frac{d}{1-2\theta} \log n(conj + k))$. The failure probability is upper bounded by taking union bound on the total number of checkings multiplied by P_k (where P_k is the failure probability of a checking subroutine with k rounds), which is $O(\frac{dP_k}{1-2\theta} \log n)$ using union bound. Lemma 1

shows that by setting the total failure probability $\delta = O(\frac{dP_k}{1-2\theta} \log n)$, it suffices to set $k = O(\log \frac{d \log n}{(1-2\theta)\delta})$, which completes the proof.