On Obtaining Stable Rankings*

Abolfazl Asudeh†, H. V. Jagadish†, Gerome Miklau††, Julia Stoyanovich‡
†University of Michigan, ††University of Massachusetts Amherst, ‡New York University
†{asudeh, jag}@umich.edu, ††miklau@cs.umass.edu, ‡stoyanovich@nyu.edu

ABSTRACT

Decision making is challenging when there is more than one criterion to consider. In such cases, it is common to assign a goodness score to each item as a weighted sum of its attribute values and rank them accordingly. Clearly, the ranking obtained depends on the weights used for this summation. Ideally, one would want the ranked order not to change if the weights are changed slightly. We call this property *stability* of the ranking. A consumer of a ranked list may trust the ranking more if it has high stability. A producer of a ranked list prefers to choose weights that result in a stable ranking, both to earn the trust of potential consumers and because a stable ranking is intrinsically likely to be more meaningful.

In this paper, we develop a framework that can be used to assess the stability of a provided ranking and to obtain a stable ranking within an "acceptable" range of weight values (called "the region of interest"). We address the case where the user cares about the rank order of the entire set of items, and also the case where the user cares only about the top-k items. Using a geometric interpretation, we propose algorithms that produce stable rankings. In addition to theoretical analyses, we conduct extensive experiments on real datasets that validate our proposal.

PVLDB Reference Format:
Abolfazl Asudeh, H. V. Jagadish, Gerome Miklau, Julia Stoyanovich. On Obtaining Stable Rankings. *PVLDB*, 12(3): 237-250, 2018.
DOI: https://doi.org/10.14778/3291264.3291269

1. INTRODUCTION

It is often useful to rank items in a dataset. It is straightforward to sort on a single attribute, but that is often not enough. When the items have more than one attribute on which they can be compared, it is challenging to place them in ranked order. Consider, for example, the problem of ranking computer science departments. Various entities, such as U.S. News and World Report, Times Higher Education, and the National Research Council, produce such rankings. These rankings are impactful, yet heavily criticized. Several of these rankings have deficiencies in the attributes they choose to measure and in their data collection methodology, not of relevance to our paper now. Our concern is that even if these deficiencies were addressed, we are compelled to obtain a single score/rank for a department by combining multiple objective measures, such as publications, citations, funding, and awards. Different ways of combining values for these attributes can lead to very different rankings. There are similar problems when we want to rank/seed sports teams, rank order cars or other products, as Malcolm Gladwell has nicely described [1].

Differences in rank order can have significant consequences. For example, a company may promote high-ranked employees and fire low-ranked employees. In university rankings, it is well-documented that the ranking formula has a significant effect on policies adopted by universities [2, 3]. In other words, it matters how we choose to combine values of multiple attributes into a scoring formula. Even when there is lack of consensus on a specific way to combine attributes, we should make sure that the method we use is robust: it should not be the case that small perturbations, such as small changes in parameter values, can change the rank order.

In this paper we address the following problem: Assume that a linear combination of the attribute values is used for assigning a score to each item; then items are sorted to produce a final ranked order. We want this ranking to be *stable* with respect to changes in the weights used in scoring. Given a particular ranked list of items, one question a consumer will ask is: how robust is the ranking? If small changes in weights can change the ranked order, then there cannot be much confidence in the correctness of the ranking.

A given ranking of a set of items can be generated by many weight functions. Because this set of functions is continuous, we can think of it as forming a region in the space of all possible weight functions. We call a ranking of items *stable* if it is generated by a weight function that corresponds to a large region of this space.

Note that if some items are very close in score, it is possible that small changes to attribute values can change their relative ordering. Such effects tend to be local, indicating that the affected items are effectively "tied" so that the change in ranking is merely a breaking of the tie. Past work [4] has considered the implications of data uncertainty and sensitivity of rankings to imprecision; it is not our focus here. Instead, we address a much bigger problem, that of changes in the ranking even without any change to the attribute values, but due to a small change in the weighting function used to compute item scores. Such global changes can drastically affect the ranked order, with far-reaching economic and societal effects [1].

Stability is a natural concern for consumers of a ranked list (i.e. those who use the ranking to prioritize items and make decisions), who should be able to assess the magnitude of the region in the weight space that produces the observed ranking. If this region is large, then the same ranked order would be obtained for many

*This work was supported in part by NSF Grants No. 1741022, 1741254, 1741047, and 1250880.

Proceedings of the VLDB Endowment, Vol. 12, No. 3
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3291264.3291269

choices of weights, and the ranking is stable. But if this region is small, then we know that only a few weight choices can produce the observed ranking. This may suggest that the ranking was engineered or "cherry-picked" by the producer to obtain a specific outcome.

Data scientists often act as producers of ranked lists (i.e. they design weight functions that result in ranked lists), and desire to produce stable results. We argued in [5] that stability in a ranked output is an important aspect of algorithmic transparency, because it allows the producer to justify their ranking methodology, and to gain the trust of consumers. Of course, stability cannot be the only criterion in the choice of a ranking function: the result may be weights that seem "unreasonable" to the ranking producer. To support the producer, we allow them to specify a range of reasonable weights, or an *acceptable region* in the space of functions, and help the producer find stable rankings within this region.

Our work is motivated by the lack of formal understanding of ranking stability and the consequent lack of tools for consumers and producers to assess this critical property of rankings. We will show that stability hinges on complex geometric properties of rankings and weight functions. We will provide a novel technique to identify stable rankings efficiently.

Our technique does not stop at proposing just the single most stable choice, or even the h most stable choices for some predetermined fixed value of h. Rather, it will continue to propose weight choices, and the corresponding rank ordering of items, beginning with the most stable in the specified region of interest, and continuing in decreasing order of stability, until the user finds one that is satisfactory.

Alternatively, our technique can provide an overview of all the rankings that occupy a large portion in the acceptable region, and hence are stable, along with an indication of the fraction of the acceptable region occupied by each. Thereby, the user can see at a glance what the stable options are, and also how dominant these are within the acceptable region.

We now motivate our techniques with an example.

EXAMPLE 1. *CSMetrics [6] ranks computer science research institutions based on the measured (M) and predicted (P) number of citations. These values are appropriately scaled and used in a weighted scoring formula, with parameter $\alpha \in [0, 1]$ that sets their relative importance (see § 6.1 for details). CSMetrics includes a handful of companies in its ranking, but we focus on academic departments in this example.*

As α ranges from 0 to 1, CSMetrics generates 336 distinct rankings of the top-100 departments. Assuming (as a baseline) that each ranking is equally likely, we would expect an arbitrarily chosen ranking to occur 0.3% of the time, which we take to mean that it occupies 0.3% of the volume in the space of attributes and weights. We formalize this in § 2.2 and call it stability of a ranking.

Suppose that the ranking with $\alpha = 0.3$ is released, placing Cornell (a consumer) at rank 11, just missing the top-10. Cornell then checks the stability of the ranking (see § 2.2.3), and learns that it's value is 0.3%, matching that of the uniform baseline. With this finding, Cornell asks CSMetrics to justify its choice of α.

CSMetrics (the producer) can respond to Cornell by further interrogating, and potentially revising the published ranking. It first enumerates stable regions (see § 2.2.4) and finds that the most stable ranking indeed places Cornell at rank 10 (switching with the University of Toronto), and represents 2% of the volume — an order of magnitude more than the reference ranking. However, this stable ranking is very far from the default, placing more emphasis on measured citations with $\alpha = 0.608$. If this is unsatisfactory, CSRankings can propose another ranking closer to the reference ranking, but with better stability (see § 2.2.2). Interestingly, Cornell also appears at the top-10 in the most stable ranking that is within 0.998 cosine similarity from the original scoring function.

Our contributions include the following:

- We formalize a novel notion of the *stability* of a ranking, for rankings that result from a linear weighting of item attribute values. Stability captures the tolerance to changes in the weights.
- We propose algorithms that enable the efficient testing of ranking stability as well as the enumeration of the most-stable rankings, optionally constrained by a set of acceptable scoring functions. We propose both exact algorithms and approximation algorithms that are based on novel sampling methods.
- Our empirical evaluation demonstrates the efficiency of our techniques on real and synthetic datasets, and investigates the stability of real published rankings of computer science departments, soccer teams, and diamond retailers. We show that existing rankings in these domains are often unstable and that favoring stability can sometimes have a significant impact on the rank of some items. For instance, our findings cast doubt on the validity of the FIFA rankings which are used in making important decisions such as seeding the World Cup final draws.

2. PROBLEM SETUP

2.1 Preliminaries

2.1.1 Data model and rankings

We consider a fixed database $\mathcal{D}$ consisting of n items, each with d scalar scoring attributes. In addition to the scoring attributes, the dataset may contain non-scoring attributes that are used for filtering, but they are not our concern here. Thus we represent an item $t \in \mathcal{D}$ as a d-length vector of scoring attributes, $\langle t[1], t[2], \ldots, t[d] \rangle$. Without loss of generality, we assume that the scoring attributes have been appropriately transformed: normalized to non-negative values between 0 and 1, standardized to have equivalent variance, and adjusted so that larger values are preferred. Note that this normalization is not strictly required for the techniques we propose in the paper but they make the comparison between attribute value weights fair and our stability measure more meaningful.

We consider rankings of items that are induced by first applying a linear weight function to each item, then sorting the items by the resulting score to form a ranking.

DEFINITION 1 (SCORING FUNCTION). *A* scoring function $f_{\vec{w}} : \mathbb{R}^d \to \mathbb{R}$, *with weight vector $\vec{w} = \langle w_1, w_2, \ldots, w_d \rangle$, assigns the score $f_{\vec{w}}(t) = \Sigma_{j=1}^{d} w_j t[j]$ to any item $t \in \mathcal{D}$.*

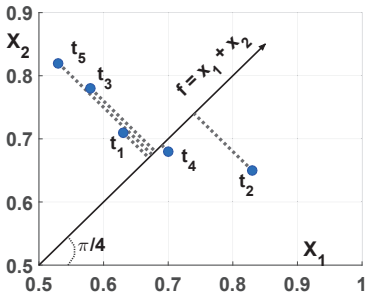
Without loss of generality, we assume that $w_j \in \vec{w} \geq 0$. This assumption is straightforward to relax with some additional notation and bookkeeping. When $\vec{w}$ is clear, we denote $f_{\vec{w}}(t)$ by $f(t)$.

We use $\mathcal{U}$ to refer to the set of all possible scoring functions. Given a score for each item, the ranking of items induced by f is the permutation of items in $\mathcal{D}$ defined by sorting them by their scores under f, in descending order, and breaking ties consistently by an item identifier. We use the notation $\nabla_f(\mathcal{D})$ to denote the ranking of items in $\mathcal{D}$ based on f.

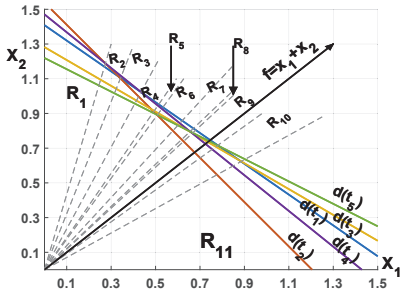
EXAMPLE 2. *The human resources (HR) department of a company wants to prioritize hiring candidates based on two criteria: an aptitude measure x_1 (e.g. a score on a qualifying exam) and an experience measure x_2 (e.g. the number of years of relevant experience). Figure 1a shows the candidates as well as their (normalized)*

	$\mathcal{D}$		f
id	x_1	x_2	$x_1 + x_2$
t_1	0.63	0.71	1.34
t_2	0.83	0.65	1.48
t_3	0.58	0.78	1.36
t_4	0.7	0.68	1.38
t_5	0.53	0.82	1.35

(a) A sample database, $\mathcal{D}$, of items with scoring attributes x_1 and x_2; and the result of scoring function $f = x_1 + x_2$.



(b) The *original space:* each item is a point. A scoring function is a ray ($f = x_1 + x_2$ is shown) which induces a ranking of the items by their projection.



(c) The *dual space:* items are the hyperplanes (lines here). Each scoring function is a ray; within a region bounded by the intersections of the item hyperplanes, all scoring functions induce the same ranking.

Figure 1: A sample database and its geometric interpretation in the original space and dual space.

values for x_1 and x_2. The score for each candidate is also shown, for weight vector $\vec{w} = \langle 1, 1\rangle$, computed as $f(t) = x_1 + x_2$.

Although we restrict our attention to linear scoring functions, our techniques can be used with more general scoring functions by applying non-linear transformations to $\mathcal{D}$ as a preprocessing step. Consider Example 2, and let the scoring function be $f(t) = x_1 + x_2 + 0.5x_1^2$. The quadratic term x_1^2 can be added as $x_3 = x_1^2$.

2.1.2 Geometry of ranked items

Our algorithms are based on a geometric interpretation of scored items and induced rankings. We now present two geometric views of the database, to which we respectively refer as (i) the *original space*, where every item corresponds to a point, and (ii) the *dual space*, where every item corresponds to a hyperplane.

The original space. The *original space* consists of $\mathbb{R}^d$ with each item in $\mathcal{D}$ represented as a point, and a linear scoring function $f_{\vec{w}}$ viewed as a ray starting from the origin and passing through the point $\vec{w} = (w_1, w_2, ..., w_d)$. The ranking $\nabla_{f_{\vec{w}}}(\mathcal{D})$ corresponds to the ordering of their projections onto this ray.

Continuing our example, Figure 1b shows the items of our sample database in the original space, as points in $\mathbb{R}^2$. The function $f = x_1 + x_2$ is shown as a ray passing through $\langle 1, 1\rangle$. The projection of the points onto the vector of f specifies the ranking: the further a point is from the origin, the higher its rank. The reason is that, for every score $f(t)$, $\sum w_j x_j = f(t)$ is the perpendicular hyperplane to the ray of f that passes through the point t. Hence, looking at Figure 1b, the candidates in Example 2 are ranked as $\langle t_2, t_4, t_3, t_5, t_1\rangle$ based on f. One can also easily imagine the ranking of items that would result from an extreme scoring function that ranks only by attribute x_1 (i.e. $f = x_1$) by considering the projections onto the x_1-axis (or respectively for the x_2-axis).

Viewing the items from $\mathcal{D}$ in the original space provides clarity about the range of rankings that can be induced by the scoring functions: all scoring functions are defined by rays in the first quadrant of $\mathbb{R}^d$ that is determined by the weight vector. It is sometimes convenient to use polar coordinates to represent a scoring function: a ray in $\mathbb{R}^d$ starting from the origin (corresponding to function $f_{\vec{w}}$) can be identified by $(d-1)$ angles $\langle \theta_1, \theta_2, \cdots, \theta_{d-1}\rangle$, each in the range $[0, \pi/2]$. Thus, given a function $f_{\vec{w}}$, its angle vector can be computed using the polar coordinates of w. For example, function f with weights $\langle 1, 1\rangle$ in Figure 1b is identified by a single angle $\langle \pi/4\rangle$. There is a one-to-one mapping between these rays and the points on the surface of the origin-centered unit d-sphere (the unit hypersphere in $\mathbb{R}^d$), or to the surface of any origin-centered d-sphere. Thus, the first quadrant of the unit d-sphere represents the universe of functions $\mathcal{U}$.

The dual space. We are particularly interested in reasoning about the transition points of the weight vector, where we move from one ranking to a different ranking. The *dual space* [7] consists of $\mathbb{R}^d$, but we represent an item t by a hyperplane $\mathsf{d}(t)$ given by the following equation of d variables $x_1 \ldots x_d$:

$$\mathsf{d}(t) : t[1] \times x_1 + \cdots + t[d] \times x_d = 1 \quad (1)$$

Continuing our example, Figure 1c shows the items in the dual space. In $\mathbb{R}^2$, every item t is a 2-dimensional hyperplane (i.e. simply a line) given by $\mathsf{d}(t) : t[1]x_1 + t[2]x_2 = 1$. In the dual space, functions are represented by the same ray as in the original space, passing through the point $\vec{w}$. Consider the intersection of a dual hyperplane $\mathsf{d}(t)$ with this ray. This intersection is in the form of $a \times \vec{w}$, because every point on the ray of f is a linear scaling of $\vec{w}$. Since this point is also on the hyperplane $\mathsf{d}(t)$, $t[1] \times a \times w_1 + \cdots + t[d] \times a \times w_d = 1$. Hence, $\sum t[j]w_j = 1/a$. This means that the dual hyperplane of any item with the score $f(t) = 1/a$ intersects the ray of f at point $a \times \vec{w}$. Following this, the ordering of the items based on a function f is determined by the ordering of the intersection of the hyperplanes with the vector of f. The closer an intersection is to the origin, the higher its rank. For example, in Figure 1c, the intersection of the line t_2 with the ray of $f = x_1 + x_2$ is closest to the origin, and so t_2 has the highest rank for f. We will show in the next section that the intersections of hyperplanes in the dual space define *regions*, within which rankings *do not* change under small changes of the weight vector.

2.2 Stability of a ranking

We now present our definition of stability and identify the key algorithmic problems for consumers and producers of rankings.

2.2.1 Definition of stability

Every scoring function in the universe $\mathcal{U}$ induces a single ranking of the items. But each ranking is generated by many functions. For database $\mathcal{D}$, let $\mathfrak{R}_\mathcal{D}$ be the set of rankings over the items in $\mathcal{D}$ that are generated by at least one scoring function $f \in \mathcal{U}$, that is, by at least one choice of weight vector. For a ranking $\mathfrak{r} \in \mathfrak{R}_\mathcal{D}$, we define its *region*, $R_\mathcal{D}(\mathfrak{r})$, as the set of functions that generate $\mathfrak{r}$:

$$R_\mathcal{D}(\mathfrak{r}) = \{f \mid \nabla_f(\mathcal{D}) = \mathfrak{r}\} \quad (2)$$

Figure 1c shows the boundaries (as dotted lines) of the regions for our sample database, one for each of the 11 feasible rankings.

We use the region associated with a ranking to define the ranking's stability. The intuition is that a ranking is stable if it can be

induced by a large set of functions. If the region of a ranking is large, then small changes in the weight vector are not likely to cross the boundary of a region and therefore the ranked order will not change. For every region R, we define its volume, $vol(R)$, to measure the bulk of the region. Specifically, we use the one-to-one mapping between the surface of the unit d-sphere and $\mathcal{U}$ for this purpose. The volume of a region is the area of the space carved out in the unit d-sphere by the set of functions in the region.

DEFINITION 2 (STABILITY OF $\mathfrak{r}$ AT $\mathcal{D}$). *Given a ranking $\mathfrak{r} \in \mathfrak{R}_\mathcal{D}$, the stability of $\mathfrak{r}$ is the proportion of ranking functions in $\mathcal{U}$ that generate $\mathfrak{r}$. That is, stability is the ratio of the volume of the ranking region of $\mathfrak{r}$ to the volume of $\mathcal{U}$. Formally:*

$$S_\mathcal{D}(\mathfrak{r}) = \frac{vol(R_\mathcal{D}(\mathfrak{r}))}{vol(\mathcal{U})} \tag{3}$$

We emphasize that stability is a property of a ranking (not of a scoring function) and it holds for a particular database, as indicated by the notation $S_\mathcal{D}(\mathfrak{r})$. For ease of notation, we denote $S_\mathcal{D}(\mathfrak{r})$ and $R_\mathcal{D}(\mathfrak{r})$ with $S(\mathfrak{r})$ and $R(\mathfrak{r})$, respectively, in the rest of this paper.

In the following, we define the scope for studying the stability of rankings, and we develop three alternative problems that build on the notion of stability in Definition 2 and correspond to the views of two different stakeholders: consumers and producers of rankings.

2.2.2 Acceptable scoring functions

When generating a ranking, the producer will often need to consider trade-offs between the choice of an acceptable scoring function and the stability of the generated ranking. Stable rankings are preferable because they are robust to small changes in scoring function weights. Furthermore, to the extent that consumers trust more stable rankings, producers are interested in earning this trust. Still, stability is not the only concern for the producer. We return to our running example to motivate this point.

EXAMPLE 3. *In producing a ranking of employees, an HR officer believes that aptitude (x_1) should be twice as important as experience (x_2), but this is only a rough guideline. Any weight with a ratio within 20% of 2 is acceptable. By testing different weights within this acceptable range, the officer observes different rankings of candidates and selects one that maximizes stability.*

We allow producers to constrain the scoring function by specifying an *acceptable region*, denoted $\mathcal{U}^* \subseteq \mathcal{U}$, in one of two ways:

- *A vector and angle distance*[1]: the acceptable region is identified by a hypercone around the central ray defined by the weight vector. For example, a user may equally prefer any function that has at most $\pi/10^\circ$ angle distance (at least 95.1% cosine similarity) with the function f with weight vector $\langle 1, 1\rangle$.
- *A set of constraints*: the acceptable region is a convex region identified by a set of inequalities. For example, if the user is interested in the functions that weigh x_2 no greater than x_1, then the acceptable region is constrained by $w_2 \leq w_1$.

We incorporate the notion of an acceptable region into the definition of stability in a natural way. Let $\mathfrak{R}^*$ be the set of rankings that are generated by at least one function $f \in \mathcal{U}^*$. The ranking region in $\mathcal{U}^*$ of a ranking $\mathfrak{r} \in \mathfrak{R}^*$ is: $R^*(\mathfrak{r}) = \{f \in \mathcal{U}^* \mid \nabla_f(\mathcal{D}) = \mathfrak{r}\}$. Accordingly, we modify the definition of stability of a ranking $\mathfrak{r} \in \mathfrak{R}^*$ to be: $S(\mathfrak{r}) = \text{vol}(R^*(\mathfrak{r}))/\text{vol}(\mathcal{U}^*)$.

[1]Note that this can be expressed by cosine similarity.

2.2.3 Consumer's stability problem

The basic problem for the consumer is *stability verification*, where the consumer seeks to validate the stability of a given ranking. A ranking with higher stability will be more robust and is less likely to be the result of an engineered scoring function.

PROBLEM 1 (STABILITY VERIFICATION). *For dataset $\mathcal{D}$ with n items over d scoring attributes, and ranking $\mathfrak{r} \in \mathfrak{R}$ of the items in $\mathcal{D}$, compute the ranking region $R_\mathcal{D}(\mathfrak{r})$ and its stability $S_\mathcal{D}(\mathfrak{r})$.*

2.2.4 Producer's stability problems

With all of the above machinery in place, we can return to helping the producer of a ranking choose one that is stable. To this end, we develop two related methods for a producer to explore stable rankings. We state these problems with respect to an acceptable region $\mathcal{U}^*$ and set $\mathcal{U}^* = \mathcal{U}$ when all scoring functions are acceptable.

First, the producer may wish to enumerate rankings, prioritizing those that are more stable. Below we consider an enumeration of rankings in order of stability, with stopping criteria based either on a stability threshold or on a bound on the number of desired rankings.

PROBLEM 2 (BATCH STABLE-REGION ENUMERATION). *For a dataset $\mathcal{D}$ with n items over d scoring attributes, a region of interest $\mathcal{U}^*$ (specified either by a set of constraints or by a vector-angle), and a stability threshold s (resp. a value h), find all rankings $\mathfrak{r} \in \mathfrak{R}^*$ such that $S(\mathfrak{r}) \geq s$ (resp. the top-h stable rankings).*

In many scenarios, rather than enumerating rankings, the producer may wish to incrementally generate stable regions, in the order of their stability, using the GET-NEXT primitive. So, the h-th call to GET-NEXT will return the h-th most stable ranking in $\mathcal{U}^*$.

PROBLEM 3 (ITERATIVE STABLE-REGION ENUMERATION). *For a dataset $\mathcal{D}$ with n items over d scoring attributes, a region of interest $\mathcal{U}^*$, specified either by a set of constraints or by a vector-angle, and the top-$(h-1)$ stable rankings in $\mathcal{U}^*$, discovered in the previous iterations of the problem, find the h-th stable ranking $\mathfrak{r} \in \mathfrak{R}$. That is, find:*

$$\underset{\mathfrak{r} \in\ \mathfrak{R} \setminus top-(h-1)}{argmax} \big(S(\mathfrak{r})\big) \tag{4}$$

Of course, the two enumeration problems are closely related. In fact, an algorithm for iterative ranking enumeration can be used directly for batch ranking enumeration, if it's called multiple times. In our algorithmic contributions we focus on efficiently evaluating an operator we call GET-NEXT, which can be used to solve both enumeration problems.

In the above, for convenience, we relate the stability enumeration to the producers and stability verification to the consumers of rankings. However, a producer can use verification for testing the stability of a ranking, while a consumer can use enumeration for identifying stable rankings.

2.2.5 Stability of the top-k items

So far we focused on complete rankings of n items in $\mathcal{D}$. However, when n is large, one may be interested in only the highest-ranked k items, for $k \ll n$. This motivates us to reformulate the problems above, focusing on the top-k portion of the ranked list.

We consider two notions of stability of the top-k items. With the first, weight vectors $\vec{w}$ and $\vec{w}'$ are said to generate the same result if they produce the same *set* of top-k items, while with the second, $\vec{w}$ and $\vec{w}'$ must both select the same set of top-k items and return

them in the same *order*. We present sampling-based randomized algorithms that support top-k partial rankings in § 4.3.

We will discuss the relationship between our approach and the rich body of work on top-k processing and skyline queries in Section 7. Here we note that the set of most-stable top-k items is in general different from the skyline [8], or any of its subsets [9–12]. The key difference is that the stable top-k items are not necessarily a subset of the skyline. Yet, these items are of high quality and so are potentially of interest to the user. Consider the toy example $\mathcal{D} = \{t_1(1,0), t_2(.99,.99), t_3(.98,.98), t_4(.97,.97), t_5(0,1)\}$. The skyline of this dataset is $\{t_1, t_2, t_5\}$, while the most stable top-3 items are $\{t_2, t_3, t_4\}$. Of these, only t_2 is part of the skyline.

3. TWO DIMENSIONAL (2D) RANKING

To develop our intuition, we start with the case of $d = 2$ scoring attributes. Using the geometric interpretation of items provided in § 2.1 while considering the dual representation of the items, we propose exact algorithms for stability verification and enumeration.

Consider a pair of items t_i and t_j presented in the dual space in $\mathbb{R}^2$. Recall that in 2D, every item t is transformed to the line:

$$\mathsf{d}(t) : t[1] \times x_1 + t[2] \times x_2 = 1 \tag{5}$$

Also, recall that every function f with the weight vector w is represented with the origin-starting ray passing through the point w, and consider points t_i and t_j. f ranks t_i higher than t_j if the intersection of $\mathsf{d}(t_i)$ with f is closer to the origin than the intersection of $\mathsf{d}(t_j)$ with f.

Consider f whose origin-starting ray passes through the intersection of $\mathsf{d}(t_i)$ and $\mathsf{d}(t_j)$. Since both lines intersect with the ray of f at the same point, f assigns an equal score to t_i and t_j. We refer to this function (and its ray) as the *ordering exchange* (first defined in [13]) between t_i and t_j, and denote it $\times_{t_i,t_j}$. The ordering between t_i and t_j changes on two sides of $\times_{t_i,t_j}$: t_i is ranked higher than t_j one side of the ray, and t_j is ranked higher than t_i on the other side. For example, consider t_1 and t_4 in Example 2, shown in Figure 1c in the dual space: the closest line to the origin on the x_1 axis represents $\mathsf{d}(t_2)$, and the next closest line is $\mathsf{d}(t_4)$. The left-most intersection in the figure is between $\mathsf{d}(t_1)$ and $\mathsf{d}(t_4)$. The top-left dashed line that starts from the origin and passes through this intersection shows $\times_{t_1,t_4}$: t_1 is preferred over t_4 on the left of $\times_{t_1,t_4}$, and t_4 is preferred over t_1 on the right.

An item t dominates [8, 14, 15] an item t', if $\nexists x_i$ s.t. $t'[i] > t[i]$ and $\exists x_i$ s.t. $t[i] > t'[j]$. If t dominates t', then these items do not exchange order. Consider two items t and t' that do not dominate each other. Equation 5 can be used for finding the intersection between the lines $\mathsf{d}(t)$ and $\mathsf{d}(t')$. Considering the polar coordinates of the intersection, $\times_{t,t'}$ is specified by the angle $\theta_{t,t'}$ (between the ordering exchange and the x-axis) as follows:

$$\theta_{t,t'} = \arctan \frac{t'[1] - t[1]}{t[2] - t'[2]} \tag{6}$$

The ordering exchanges between pairs of items of a database partition the space of scoring functions into a set of regions. Each region is identified by the two ordering exchanges that form its borders. Since there are no ordering exchanges within a region, all scoring functions inside a region induce the same ranking of the items. Thus, the number of regions is equal to $|\mathfrak{R}|$, as $\mathfrak{R}$ is the collection of rankings defined by these regions. For instance, Figure 1c shows regions R_1 through R_{11} that define the set of possible rankings of Example 2 for $\mathcal{U}$.

3.1 Stability Verification

The ordering exchanges are the key to figuring out the stability of a ranking. Consider a ranking $\mathfrak{r}$. For a value of $1 \leq i < n$, let t and t' be the i-th and $(i+1)$-th items in $\mathfrak{r}$. Following Equation 6, $\theta_{t,t'}$ specifies the ordering exchange between t and t'. If $t[1] < t'[1]$ (resp. $t[1] > t'[1]$), all functions with angles $\theta < \theta_{t,t'}$ (resp. $\theta > \theta_{t,t'}$) rank t higher than t'. The reason is that if $t[1] > t'[1]$, $t[2]$ should be smaller than $t'[2]$, otherwise t dominates t'. Hence $\frac{t[1]}{t[2]} > \frac{t'[1]}{t'[2]}$, i.e. the dual line $\mathsf{d}(t)$ has a larger slope than $\mathsf{d}(t')$, and intersects the rays in range $[0, \theta_{t,t'})$ closer to the origin.

We use this idea for computing the stability (and the region) of a given ranking $\mathfrak{r}$. The stability verification algorithm uses the angle range (θ_1, θ_2) for specifying the region of $\mathfrak{r}$. For each value of i in range $[1, n)$, the algorithm considers the items t and t' to be the i-th and $(i+1)$-th items in $\mathfrak{r}$, respectively. If t' dominates t, the ranking is not valid. Otherwise, if t does not dominate t', the algorithm computes the ordering exchange $\times_{t,t'}$ and, based on the values of $t[1]$ and $t'[1]$, decides to use it for setting the upper bound or the lower bound of the ranking region. After traversing the ranked list $\mathfrak{r}$, the algorithm returns $\frac{\theta_2 - \theta_1}{\pi/2}$ as the stability value and (θ_1, θ_2) as the region of $\mathfrak{r}$. Since the algorithm scans the ranked list only once, stability verification in 2D is in $O(n)$. The algorithm's pseudocode is provided in the technical report [16].

3.2 Stability Enumeration

In 2D, $\mathcal{U}^*$ is identified by two angles demarcating the edges of the pie-slice. For example, let $\mathcal{U}_1^*$ be defined by the set of constraints $\{w_1 \leq w_2, 2w_1 \geq w_2\}$. This defines the set of functions above the line $w_1 = w_2$ and below the line $2w_1 = w_2$, limiting the region of interest to the angles in the range $[\pi/4^\circ, \pi/3^\circ]$. Similarly, region $\mathcal{U}_2^*$ defined around $f = x_1 + x_2$ with the maximum angle $\pi/10^\circ$ corresponds to the angles in the range $[3\pi/20^\circ, 7\pi/20^\circ]$. In what follows, we use $[\,\mathcal{U}^*[1],\ \mathcal{U}^*[2]\,]$ to denote the borders of $\mathcal{U}^*$. Based on Definition 2, the stability of a ranking $\mathfrak{r} \in \mathfrak{R}$ in 2D is the span of its region – the distance between its two borders.

We propose the algorithm RAYSWEEPING (Algorithm 1) that starts from the angle $\mathcal{U}^*[1]$ and, while sweeping a ray toward $\mathcal{U}^*[2]$, uses the dual representation of the items for computing the ordering exchanges and finding the ranking regions. The algorithm stores the regions, along with the stability of their rankings, in a heap data structure that is later used by the GET-NEXT$_{2D}$ primitive.

Algorithm 1 starts by ordering the items based on $\mathcal{U}^*[1]$. It uses the fact that at any moment, an adjacent pair in the ordered list of items exchange ordering, and, therefore, computes the ordering exchanges between the adjacent items in the ordered list. The intersections that fall into the region of interest are added to the sweeper's min-heap. Until there are intersections over which to sweep, the algorithm pops the intersection with the smallest angle, marks the region between it and the previous intersection in the output max-heap, and updates the ordered list accordingly. Upon updating the ordered list, the algorithm adds the intersections between the new adjacent items to the sweeper. Since the total number of intersections between the items is bounded by $O(n^2)$, and the heap operation is in $O(\log n)$, RAYSWEEPING is in $O(n^2 \log n)$.

After finding the ranking regions and theirs spans, every call to GET-NEXT$_{2D}$ pops the most stable region from the heap and chooses a scoring function f inside the region. The algorithm returns the ranking $\nabla_f(\mathcal{D})$, along with the width of the region (its stability), to the user. Due to the space limitations, the pseudo code of GET-NEXT$_{2D}$ is provided in the technical report [16].

Since there are no more than $O(n^2)$ regions in the heap, GET-NEXT$_{2D}$ needs $O(\log n)$ to find the $(h+1)$-th stable region. Then,

Algorithm 1 RAYSWEEPING
Input: Two dimensional dataset $\mathcal{D}$ with n items and the region of interest in the form of an angle range $[\,\mathcal{U}^*[1],\, \mathcal{U}^*[2]\,]$
Output: A heap of ranking regions and their stability

```
sweeper = new min-heap([U*[2]]);
w = (cos U*[1], sin U*[1])
L = ∇_f(D)
for i = 1 to n − 1 do
    θ = arctan(L_{i+1}[1] − L_i[1])/(L_i[2] − L_{i+1}[2])
    if U*[1] < θ < U*[2] then sweeper.push((θ, L_i, L_{i+1}))
end for
h = new max-heap(); θ_p = U*[1]
while sweeper is not empty do
    (θ, t,t') = sweeper.pop()
    i, j = index of t, t' in L
    h.push ( (θ − θ_p)/(U*[2] − U*[1]), (θ_p, θ) )
    swap L_i and L_j and add the ordering exchanges between
    the new adjacent items to the sweeper
    θ_p = θ
end while
return h
```

it takes $O(n \log n)$ to compute the ranking for the region. As a result, the first call to GET-NEXT$_{2D}$ that creates the heap of regions takes $O(n^2 \log n)$, while subsequent calls take $O(n \log n)$. Note that subsequent GET-NEXT$_{2D}$ calls can be done in the order of $O(\log n)$, with memory cost of $O(n^3)$, by storing the ordered list L for every region in RAYSWEEPING algorithm.

4. MULTI DIMENSIONAL (MD) RANKING

Building upon the intuitions developed from the 2D case, we now turn to the general setting where $d > 2$. Again, we consider the items in dual space and use the ordering exchanges for specifying the borders of the ranking regions. Recall from Equation 1 that an item t is presented as the hyperplane $\mathsf{d}(t) : \sum_{i=1}^{d} t[i].x_i = 1$. For a pair of items t_i and t_j the ordering exchange $h = \times_{t_i,t_j}$ is a hyperplane that contains the functions that assign the same score to both items. Therefore:

$$\times_{t_i,t_j} = \sum_{k=1}^{d} (t_i[k] - t_j[k])x_k = 0 \tag{7}$$

Every hyperplane $h = \times_{t_i,t_j}$ *partitions* the function space $\mathcal{U}$ in two "half-spaces" [7]:

- $h^- : \sum_{k=1}^{d} (t_i[k] - t_j[k])x_k < 0$: for the functions in h^-, t_j outranks t_i.
- $h^+ : \sum_{k=1}^{d} (t_i[k] - t_j[k])x_k > 0$: for the functions in h^+, t_i outranks t_j.

Similar to § 3, we first show how ordering exchanges can be used for verifying the stability of a ranking and then focus on designing the GET-NEXT operator.

4.1 Stability Verification

Identifying the half-spaces defined by the ordering exchanges between adjacent items in a ranking $\mathfrak{r}$ is the key to figuring out its stability. For each value of i in range $[1, n)$, let t and t' be the i-th and $(i+1)$-th items in $\mathfrak{r}$. Using Equation 7, every function in the positive half-space $h^+ : \sum_{k=1}^{d} (t[k] - t'[k])x_k > 0$ ranks t higher than t'. The intersection of these half-spaces specifies an open-ended d-dimensional cone (referred to as d-cone) whose base is a $(d-1)$ dimensional convex polyhedron. Every function in this cone generates the ranking $\mathfrak{r}$, while no function outside it generates $\mathfrak{r}$. In other words, this d-cone is the ranking region of $\mathfrak{r}$. The algorithm for verifying stability finds the region of a ranking $\mathfrak{r}$ as the set of positive half-spaces defined by the ordering exchanges of the adjacent items in $\mathfrak{r}$.

Based on Definition 2, the volume ratio of the region of $\mathfrak{r}$ to the one of $\mathcal{U}$ (or, more generally, to $\mathcal{U}^*$) is its stability. However, since $\mathfrak{r}$ is a polyhedron, computing its volume is #P-hard [17]. Therefore, we use numeric methods and sampling for estimating this quantity. Throughout this section, we assume the existence of a stability oracle $S(R, \mathcal{U}^*)$ that, given a convex region R in the form of an intersection of half-spaces and a region of interest $\mathcal{U}^*$, returns the stability of R in $\mathcal{U}^*$. Due to the space limitations, We will describe in the technical report [16] how to construct such an oracle. Stability verification in MD is in $O(n + X_S)$ where X_S is the complexity of constructing the stability oracle.

4.2 Stability Enumeration

Similarly to verifying the stability of a ranking, ordering exchanges can be used for finding possible rankings in a region of interest $\mathcal{U}^*$. The set of ordering exchanges intersecting $\mathcal{U}^*$ define a dissection of $\mathcal{U}^*$ into connected convex d-cones, each showing a ranking region as the intersection of ordering exchange half-spaces and $\mathcal{U}^*$. This dissection is named the *arrangement* of ordering exchange hyperplanes [7]. For example, the ordering exchanges in $\mathbb{R}^3$ are the planes passing through the origin. Each plane dissects the space in two half-spaces. The intersection of the half-spaces forms an arrangement in the form of open-ended convex cones.

THEOREM 1. *Every ranking $\mathfrak{r} \in \mathfrak{R}^*$ is provided by the functions in exactly one convex region in the arrangement of ordering exchange hyperplanes in $\mathcal{U}^*$.*

Proof sketch: The proof follows the non-existence of ordering exchanges inside a region and the existence of at least one ordering exchange between two regions. Additional details are provided in the technical report [16].

Theorem 1 shows the one-to-one mapping between the rankings $\mathfrak{r} \in \mathfrak{R}^*$ and the regions of the arrangement. Following Theorem 1, the baseline for finding the stable regions in $\mathcal{U}^*$ is to first construct the arrangement and then, similarly to § 3, create a heap of regions and their stabilities. Then, each GET-NEXT operation is as simple as removing the most stable ranking from the heap. The construction of arrangements is extensively studied in the literature [7, 13, 18–21]. The problem with this baseline is that it first needs to construct the complete arrangement of ordering exchange hyperplanes, and to compute the stability of each. The number of ordering exchanges intersecting $\mathcal{U}^*$ is bounded by $O(n^2)$. Therefore the arrangement can contain as many as $O(n^{2d})$ regions [7]. Yet, the baseline needs to compute the stability of each ranking associated with every region. Considering that our objective is to find stable rankings, rather than to discover all possible rankings, and that the user will likely be satisfied after observing *a few* rankings, this construction if wasteful. Instead, since the objective is to find the next stable ranking (not to discover all rankings), we propose an algorithm that targets the construction of only the next stable ranking and delays the construction of other rankings.

Arrangement construction is an iterative process that starts by partitioning the space into two half-spaces by adding the first hyperplane $H[1]$ (it partitions the space into $H[1]^-$ and $H[1]^+$). The algorithm then iteratively adds the other hyperplanes; to add $H[i]$, it first identifies the set of regions in the arrangement of $H[1]$ to $H[i-1]$ that $H[i]$ intersects with, and then splits each such region into two regions (one including $H[i]^-$ and one $H[i]^+$).

The GET-NEXT$_{md}$ algorithm, however, only breaks down the largest region at every iteration, delaying the construction of the arrangement in all other regions. The algorithm uses the "region" data structure to record each region in the arrangement of ordering exchanges. This data structure contains the following fields: (a) `C`: the set of half-spaces defining the region; (b) `S`: the stability of the region, and (c) `pending`: the index of the next hyperplane to be added to the region. In addition, every region contains two extra fields `sb` and `se` that are used for determining the intersection of next hyperplanes with it. Due to the space limitations, we provide further details about these, as well as the pseudo code of the algorithm GET-NEXT$_{md}$ in the technical report [16].

While constructing the arrangement of hyperplanes, the algorithm keeps track of the stability of the regions, as it adds hyperplanes to the largest one. It uses a max-heap for this purpose. For the first GET-NEXT operation, the algorithm finds the set of ordering exchanges H that intersect with $\mathcal{U}^*$. It also creates the root region that contains all functions in $\mathcal{U}^*$ and adds it to the heap. While the heap is not empty, the algorithm pops the most stable region r from it. It then iterates over the pending hyperplanes that can be added to r, attempting to find one that intersects with that region. Testing whether a hyperplane intersects with a region is done by solving a linear program. Specifically, we solve a quadratic program that looks for a function in $\mathcal{U}^*$ that satisfies both the inequality constraints defined by the half-spaces of the region, and the equality constraints defined by the hyperplane. (Alternatively, sampling can be used for this purpose. We provide further details about this in the technical report [16].)

If no more hyperplanes can be added to region r, the algorithm returns r as the next stable region. Otherwise, if a hyperplane is found that intersects with r, then the algorithm uses it to break r into two regions, and adds them to the heap.

Still, in the worst case (where all regions are equally stable) the algorithm may need to construct the arrangement before returning even the first stable region. Therefore, the worst case cost of the algorithm is still $O(n^{2d})$.

Throughout this section, we assumed the existence of an oracle that, given a region in the form of a set of half-spaces, returns its stability. In next section, we discuss unbiased sampling from the function space that plays a key role in the design of the oracle. Such sampling will also enable the design of the randomized algorithm in § 4.3 that does not depend on the arrangement construction (and therefore, does not suffer its high complexity).

4.3 Randomized Get-Next

In a setting with many items, users are usually not interested in the complete ranking between all of the items. The top-k items model [22, 23] is a natural fit for such settings, and therefore, is used as the de-facto data retrieval model in the web [24,25]. In this model, the user is interested in the head (the top-k) of the ranked list, rather than the complete ordering. In the following, we propose a randomized algorithm that, in addition to being scalable for large settings, is applicable for enumerating the top-k items.

While every ranking is generated by continuous ranges of functions, every function f generates only one ranking of items. Moreover, the larger the volume of a ranking region (i.e. the more stable it is), the higher the chance of choosing a random function from it. Therefore, *uniform sampling of the function space allows sampling of rankings based on their stability distribution*. We delay the details of a sampler that generates functions uniformly at random from a region of interest $\mathcal{U}^*$ to § 5. Assuming the existence of such sampler, in this section, we use the Monte-Carlo methods for Bernoulli random variables [26–30] and design the randomized GET-NEXT$_r$ operator. This operator works both for finding the stable rankings in a region of interest, as well as the top-k results. In the following, we use ranking for the explanations but all the algorithms and explanations are also valid for top-k.

Monte-Carlo methods work based on repeated sampling and the central limit theorem in order to solve deterministic problems. We consider using these algorithms for numeric integration. Based on the law of large numbers [31], the mean of independent random variables can be used for approximating the integrals. That is possible, as the expected number of occurrences of each observation is proportional to its probability. At a high-level, the Monte-Carlo methods work as follows: first they generate a sufficiently large set of inputs based on a probability distribution over a domain; then they use these inputs to do estimation and aggregate the results.

We use sampling both for discovering the rankings as well as for estimating their stability. We design the GET-NEXT$_r$ operator to allow the user to either (i) specify the sampling budget, or (ii) the confidence interval. Each of these two approaches has their pros and cons. The running time in (i) is fixed but the error is variable. In (ii), on the other hand, the operator guarantees the output quality while the running time is not deterministic. In the following, we explain the details for (i). Due to space limitations, we show how this can be adopted for (ii) in the technical report [16].

The sampler explained in § 5.2 draws functions uniformly at random from the function space. Each function is associated with a ranking. The uniform samples on the function space provide ranking samples based on their stabilities (the portion of functions in $\mathcal{U}^*$ generating them). For each ranking $\mathfrak{r} \in \mathfrak{R}$, consider the distribution of drawing a function that generate it. The probability mass function of this distribution is:

$$p(\Theta; S(\mathfrak{r})) = \begin{cases} S(\mathfrak{r}) & \nabla_{f(\Theta)}(\mathcal{D}) = \mathfrak{r} \\ 1 - S(\mathfrak{r}) & \nabla_{f(\Theta)}(\mathcal{D}) \neq \mathfrak{r} \end{cases} \tag{8}$$

Let the random Bernoulli variable $x_\mathfrak{r}$ be 1 if $\nabla_{f(\Theta)}(\mathcal{D}) = \mathfrak{r}$ and 0 otherwise. Recall that the mean and standard deviation of a Bernoulli distribution with the success probability of $S(\mathfrak{r})$ are $\mu_\mathfrak{r} = S(\mathfrak{r})$ and $\sigma_\mathfrak{r} = S(\mathfrak{r})(1 - S(\mathfrak{r}))$. Let $m_\mathfrak{r}$ be the average of a set of N samples of the random variable $x_\mathfrak{r}$. Then, $E[m_\mathfrak{r}] = S(\mathfrak{r})$ and the standard deviation of samples are $s_\mathfrak{r} = m_\mathfrak{r}(1-m_\mathfrak{r})$. Based on the central limit theorem, we also know that the distribution of the sample average is $\mathrm{N}\big(\mu_\mathfrak{r}, \frac{\sigma_\mathfrak{r}}{\sqrt{N}}\big)$ – the Normal distribution with the mean $\mu_\mathfrak{r}$ and standard deviation $\frac{\sigma_\mathfrak{r}}{\sqrt{N}}$. For a large value of N, we can estimate $\sigma_\mathfrak{r}$ by $s_\mathfrak{r}$.

For a confidence level α, the confidence error e identifies the range $[m_\mathfrak{r} - e, m_\mathfrak{r} + e]$ such that:

$$p(m_\mathfrak{r} - e \leq \mu_\mathfrak{r} \leq m_\mathfrak{r} + e) = 1 - \alpha \tag{9}$$

Using the Z-table:

$$e = Z(1 - \frac{\alpha}{2})\frac{s_\mathfrak{r}}{\sqrt{N}} = Z(1 - \frac{\alpha}{2})\sqrt{\frac{m_\mathfrak{r}(1 - m_\mathfrak{r})}{N}} \tag{10}$$

Using this argument, we use a set of N samples of the function space for the design of GET-NEXT$_r$ with a fixed budget. Every time GET-NEXT$_r$ is called, we collect a set of N samples and use them for finding the next stable ranking and estimating its stability. In order to provide a more accurate estimation, in addition to the N new samples, it uses the aggregates of its previous runs. Algorithm 2 shows the pseudocode of GET-NEXT$_r$ with a fixed budget.

Algorithm 2 uses a hash data structure that contains the aggregates of the rankings it has observed so far. Upon calling the algorithm, it first draws N sample functions from the region of interest $\mathcal{U}^*$. For each sample function, the algorithm finds the corresponding ranking and checks if it has previously been discovered. If not,

Algorithm 2 GET-NEXT$_r$
Input: $\mathcal{D}$, $\mathcal{U}^*$, previous stable rankings $\mathfrak{R}_{h-1}$, hash of previous aggregates $cnts$, total number of previous samples N', confidence level α, and sampling budget N
Output: Next stable ranking and its stability measures

1: **for** $k = 1$ to N **do**
2: $w = \text{Sample}_{\mathcal{U}^*}(\mathcal{U}^*)$
3: $\mathfrak{r} = \nabla_f(\mathcal{D})$
4: **if** $\mathfrak{r}$ is in $cnts.keys$ **then** $cnts[\mathfrak{r}]$+=1 **else** $cnts[\mathfrak{r}] = 1$
5: **end for**
6: **if** $cnts.keys \backslash \mathfrak{R}_{h-1} = \emptyset$ **then return** null
7: $\mathfrak{r}_h = \underset{\mathfrak{r} \in\, cnts.keys \backslash \mathfrak{R}_{h-1}}{\text{argmax}} \big(cnts[\mathfrak{r}]\big)$
8: $S(\mathfrak{r}_h) = \frac{cnts[\mathfrak{r}_h]}{N+N'}$; $e = Z(1-\frac{\alpha}{2})\sqrt{\frac{S(\mathfrak{r}_h)(1-S(\mathfrak{r}_h))}{N+N'}}$
9: **return** $\mathfrak{r}_h, S(\mathfrak{r}_h), e$

it adds the ranking to the hash and sets its count as 1; otherwise, it increments the count of the ranking. If the number of discovered rankings is at most h, the algorithm fails to find a new ranking and returns null. The algorithm then chooses the ranking that does not belong to top-h and has the maximum count. It computes the stability and confidence error of the ranking and returns it.

Considering a budget of N samples while finding the ranking for each sample, the running time of GET-NEXT$_r$ is $O(N \times n \log n)$.

4.3.1 Stable top-k items

When n is large, instead of the complete ranking, the user may be interested in the top-k items. The top-k items may either be treated as a set or a ranked list. A company that considers its top-k candidates for the on-site interview is an example of the top-k set model, whereas for a student that wants to apply for the top colleges, the ranking between the top-k colleges is important.

Unfortunately the MD algorithm GET-NEXT$_{md}$ is not applicable here, as different ranking regions may share the same top-k items. Therefore, the algorithm cannot focus only on a single region, while delaying the others. Fortunately, the randomized algorithm GET-NEXT$_r$ can be used for partial rankings. Instead of maintaining the counts of complete rankings, it counts the occurrences of partial rankings. In § 6, we will show that GET-NEXT$_r$ is both effective and efficient for top-k items.

5. UNBIASED FUNCTION SAMPLING

A uniform sampler from the function space is a key component for devising Monte-Carlo methods, both for estimating the stability of rankings and for designing randomized algorithms for the problem. In the following, we first discuss sampling from the complete function space and then propose an efficient sampler for $\mathcal{U}^*$.

5.1 Sampling from the function space

In this subsection we discuss how to generate unbiased samples from the complete function space. Since every function is represented as a vector of $d-1$ angles, each in range $[0^\circ, \pi/2^\circ]$, one way of generating random functions is by generating angle vectors uniformly at random. This, however, does not provide uniform random functions sampled from the function space, except for 2D.

As mentioned in § 2.1, the set of points in the first quadrant of the unit d-sphere represent the function space $\mathcal{U}$. This is because of the one-to-one mapping between the points on the surface of the unit d-sphere and the unit origin-starting rays, each representing a function f. Hence, the problem of choosing functions uniformly at random from $\mathcal{U}$ is equivalent to choosing random points from the surface of a d-sphere. As also suggested in [32], we adopt a method for uniform sampling of the points on the surface of the unit d-sphere [33,34]. Rather than sampling the angles, this method samples the weights using the *normal distribution*, and normalizes them. This method works because the normal distribution function has a constant probability on the surfaces of d-spheres with common centers [34, 35]. Therefore, in order to generate a random function in $\mathcal{U}$, we set each weight as $w_i = |\mathcal{N}(0,1)|$, where $\mathcal{N}(0,1)$ draws a sample from the standard normal distribution.

5.2 Sampling from the region of interest

Drawing unbiased samples from a region of interest $\mathcal{U}^*$ is critical for finding stable regions. Given an unbiased sampler for the function space $\mathcal{U}$, an acceptance-rejection method [36] can be used for drawing samples from $\mathcal{U}^*$. The idea is simple: (i) draw a sample from the function space; (ii) test if the drawn sample is in the region of interest and, if so, accept it; otherwise reject the sample and try again. Testing if the drawn sample is in the region of interest can be done by: (a) computing its angle distance from the reference ρ and comparing it with the reference angle θ, if $\mathcal{U}^*$ is specified by a ray and angle, or by (b) checking if the sampled point satisfies the constraint, if $\mathcal{U}^*$ is specified by a set of constraints.

The efficiency of this method, however, depends on the acceptance probability p, defined by the volume ratio of $\mathcal{U}^*$ to $\mathcal{U}$. The expected number of trials for drawing a sample for such probability is $1/p$. Hence, it is efficient if the volume of $\mathcal{U}^*$ is not small.

Therefore, in the following, we alternatively propose an inverse CDF (cumulative distribution function) method [37] for generating random uniform functions from a region of interest. This method is preferred over the acceptance-rejection method when $\mathcal{U}^*$ is small. It generates functions with the maximum angular distance of θ from the reference ray ρ. For a region of interest specified by a set of constraints, the bounding sphere [38] for the base of its d-cone identifies the ray and angle distance that include $\mathcal{U}^*$. For such regions of interest, the inverse CDF method enables an acceptance-rejection method with higher acceptance rate, leading to better performance.

Consider $\mathcal{U}^*$ as the set of functions with the maximum angle θ around the ray of some function f. We model $\mathcal{U}^*$ by the surface unit d-spherical cap with angle θ around the d-th axis in $\mathbb{R}^d$ (Figure 2a). This is similar to the mapping of $\mathcal{U}$ to the surface of the unit hyperspherical, and is due to the one-to-one mapping between the rays in $\mathcal{U}^*$ and the points on the surface unit hyperspherical cap. We use a transformation that maps the ray of f to the d-D axis. After drawing a function we transform it around the ray of f.

For an angle θ, the d-dimension orthogonal plane $x_d = \cos\theta$ partitions the cap from the rest of the d-sphere. Hence, the intersection of the set of the (d-th axis orthogonal) planes $\cos\theta \le x_d \le 1$ with the d-sphere define the cap.

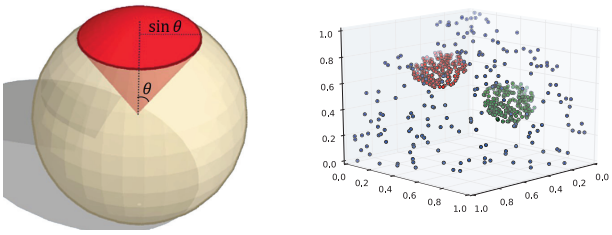
The intersection of each such plane with the d-sphere is a $(d-1)$-sphere. For example, in Figure 2a the intersection of a plane that is orthogonal to the z-axis with the unit sphere is a circle (2-sphere). An unbiased sampler should sample points from the surface of such $(d-1)$-spheres proportionally to their areas. In the following, we show how this can be done.

The surface area of a δ-sphere with the radius r is [39]:

$$A_\delta(r) = \frac{2\pi^{\delta/2}}{\Gamma(\delta/2)} r^{\delta-1} \tag{11}$$

Γ, in the above equation, is the gamma function.

Using this equation, the area of the unit d-spherical cap can be stated as the integral over the surface areas of the $(d-1)$-spheres, defined by the intersection of the planes $\cos\theta \le x_d \le 1$ with the



(a) (b)

Figure 2: Sampling from $\mathcal{U}^*$. a) $\mathcal{U}^*$ as a unit d-spherical cap around d-th axis. b) Samples generated using (blue – scattered over the space): § 5.1, (green – right cluster): Algorithm 3 and numeric inverse CDF, (red – left cluster): Algorithm 3 and Equation 14.

d-sphere, as follows [39]:

$$A_d^{cap}(1) = \int_0^\theta A_{d-1} \sin\phi d\phi = \frac{2\pi^{d/2}}{\Gamma(d/2)} \int_0^\theta \sin^{d-2}(\phi) d\phi \quad (12)$$

Therefore, considering the random angle $0 \leq x \leq \theta$, the cumulative density function (cdf) for x is given by:

$$F(x) = \frac{\int_0^x \sin^{d-2}(\phi) d\phi}{\int_0^\theta \sin^{d-2}(\phi) d\phi} \quad (13)$$

For a specific value of d, one can solve Equation 13, find the inverse of F and use it for sampling. For instance, for $d = 3$:

$$F(x) = \frac{1-\cos x}{1-\cos\theta} \Rightarrow F^{-1}(x) = \arccos\big(1-(1-\cos\theta)x\big) \quad (14)$$

For a general d, we use numerical methods for finding the inverse CDF. Details can be found in the technical report [16].

Algorithm 3 shows the pseudocode of the inverse CDF sampler. For example, consider the example in $\mathbb{R}^3$, where the objective is to generate random numbers around the ray $(\pi/6, \pi/4)$ with angle $\theta = \pi/20$. The algorithm starts by drawing a random uniform number in range $[0, 1]$. Let such a random number be 0.13. It takes the list L (computed using the function RiemannSums) as the input and draws a random function from $\mathcal{U}^*$. To do so, it first draws a random uniform number y in the range [0,1]. Next, it applies a binary search on the list of partial integrals to find the range to which y belongs. Considering a fine granularity of the partitions, we assume that the areas of all $(d-1)$-spheres inside each partition are equal. The algorithm, therefore, returns a random angle (drawn uniformly at random) from the selected partition. Obviously, instead, the algorithm can use the equation of the inverse function. Continuing with our example, while using Equation 14, the corresponding y value for 0.13 is $\pi/55.5$.

Algorithm 3 Sample $\mathcal{U}^*$
Input: The ray ρ, angle θ

1: $y = U[0, 1]$ `// draw a uniform sample in range [0,1]`
2: $x = F^{-1}(y)$
3: **for** $i = 1$ to $d-1$ **do** $\hat{w}_i = \mathcal{N}(0, 1)$
4: $\langle\theta_1, \cdots, \theta_{d-2}\rangle$ = the angles in polar representation of $\hat{w}$
5: $w =$ **toCartesian**$(1, \langle\theta_1, \cdots, \theta_{d-2}, x\rangle)$
6: **return Rotate**(w, ρ)

Recall that the angle x specifies the intersection of a plane with the d-spherical cap, which is a $(d-1)$-sphere. Hence, after finding the angle x, we need to sample from the surface of a $(d-1)$-sphere, uniformly at random. For our example in $\mathbb{R}^3$, the intersection is a circle (2-sphere) and, therefore, we need to sample from the surface of the circle. Also, recall from § 5.1 that the normalized set of $d-1$ random numbers drawn from the normal distribution provide a random sample point on the surface of $(d-1)$-sphere. The algorithm Sample$\mathcal{U}^*$ uses this for generating such a random point. It uses the angle combination of the drawn random point from the surface of a $(d-1)$-sphere and combines them with the angle x (with the d-th axis). In our example in $\mathbb{R}^3$, let the sampled point on the circle have the angle 0.8π. Hence, the angle combination is $\langle 0.8\pi, \pi/55.5\rangle$. After this step, the point specified by the polar coordinates $(1, \langle\theta_1, \cdots, \theta_{d-2}, x\rangle)$ is the random uniform point from the surface of d-spherical cap around the d-th axis. As the final step, the algorithm needs to rotate the coordinate system such that the center of the cap (currently on d-th axis) falls on the ray ρ. We rely on the existence of the function *Rotate* for this, presented in the technical report [16]. Figure 2b shows three cases of 200 samples in $\mathbb{R}^3$ drawn from (blue – scattered over the space) $\mathcal{U}$ using § 5.1 and (green and red – right and left clusters) $\mathcal{U}^*$ around the rays $(\pi/3, \pi/3)$ and $(\pi/6, \pi/4)$ with angle $\theta = \pi/20$.

6. EXPERIMENTS

Here we validate our stability measure and evaluate the efficiency of our algorithms on three real datasets used for ranking. In particular, we study the stability of two of our datasets in § 6.2, showing that the proposed reference rankings are not stable. In § 6.3, we study the running times of our algorithms, including stability verification, the GET-NEXT problems in 2D and MD, as well as the randomized algorithm and top-k items.

6.1 Experimental setup

Hardware and platform. The experiments were conducted using a 3.8 GHz Intel Xeon processor, 64 GB memory, running Linux. The algorithms were implemented using Python 2.7.

Datasets We use four real datasets CSMetrics ($d = 2$), FIFA ($d = 4$), Blue Nile ($d = 5$), and Department of Transportation ($d = 3$), as well as a set of three synthetic datasets described below.

CSMetrics [6]: CSMetrics ranks computer science research institutions based on publication metrics. For each institution, a combination of `measured` (M) citations and an attribute intended to capture future citations, called `predicted` (P), is used for ranking, according the score function: $(M)^\alpha(P)^{1-\alpha}$, for parameter α. This score function is not linear, but under a transformation of the data in which $x_1 = log(M)$ and $x_2 = log(P)$ we can write an equivalent score function linearly as: $\alpha x_1 + (1-\alpha)x_2$. The CSMetrics website uses $\alpha = .3$ as the default value, but allows other values to be selected. We use $\alpha = .3$ and restrict our attention to the top-100 institutions according to this ranking.

FIFA Rankings [40]: The FIFA World Ranking of men's national football teams is based on measures of recent performance. Specifically, the score of a team t depends on team performance values for A_1 (current year), A_2 (past year), A_3 (two years ago), and A_4 (three years ago). The given score function, from which the reference ranking is derived, is: $t[1] + 0.5t[2] + 0.3t[3] + 0.2t[4]$. FIFA relies on these rankings for modeling the progress and abilities of the national-A soccer teams [41] and to seed important competitions in different tournaments, including the 2018 FIFA World Cup. We consider the top 100 teams in our experiments.

Blue Nile [42]: Blue Nile is the largest online diamond retailer in the world. We collected its catalog that contained 116,300 diamonds at the time of our experiments. We consider the scalar attributes `Price`, `Carat`, `Depth`, `LengthWidthRatio`, and

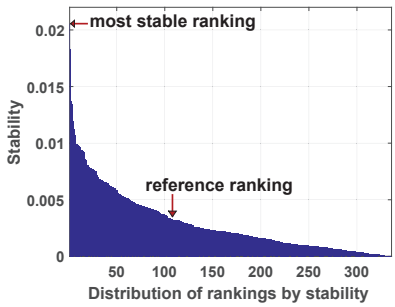


Figure 3: CSMetrics: overall distribution of rankings by stability.

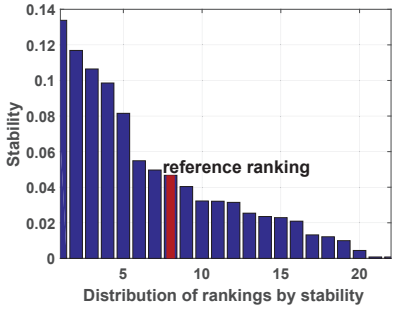


Figure 4: CSMetrics: stability around reference vector $\langle 0.3, 0.7\rangle$ with 0.998 cosine sim.

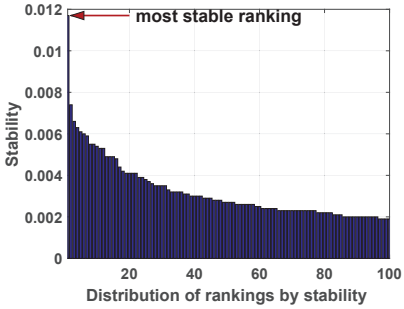


Figure 5: FIFA: stability around reference vector $\langle 1, 0.5, 0.3, 0.2\rangle$ with 0.999 cosine sim.

`Table` for ranking. For all attributes, except `Price`, higher values are preferred. We normalize each value v of a higher-preferred attribute A as $(v - \min(A))/(\max(A) - \min(A))$; for a lower-preferred attribute A, we use $(\max(A) - v)/(\max(A) - \min(A))$.

Department of Transportation (DoT) [43]: The flight on-time dataset is published by the US Department of Transportation. We collected a set of 1,322,023 records, for the flights conducted by the 14 US carriers in the last three months of 2017. We consider the attributes `air-time`, `taxi-in` and `taxi-out` for ranking.

Synthetic Data: In order to study the effect of the correlation between the attributes, using the code provided by [8], we generated three synthetic datasets (independent, correlated, anti-correlated), containing 10,000 items and three scoring attributes in range $[0, 1]$.

6.2 Stability investigation of real datasets

For the two datasets which provide a reference ranking (CSMetrics and FIFA) we assess these rankings below.

CSMetrics: Two attributes are used for ranking here (i.e. $d = 2$). We can therefore use the GET-NEXT operator repeatedly to enumerate all feasible rankings and their stability values. While an upper bound on the number of rankings for $n = 100$ and $d = 2$ is around $5,000$, the actual number of feasible rankings for this dataset is 336. Figure 3 shows the distribution of rank stability across all rankings, showing a few rankings with high stability, after which stability rapidly drops. The reference ranking is highlighted in Figure 3; using SV_{2D}, we calculated the stability of the reference ranking to be 0.0032. Notably, the reference ranking did not appear in the top-100 stable rankings (it is the 108^{th} most stable ranking).

Maximizing stability would cause a number of changes compared with the reference ranking. For example, Cornell University is not in the top-10 universities in the reference ranking, but replaces the University of Toronto in the top-10 in the most stable ranking. One of the bigger changes in rank position is Northeastern University which improves from 40 in the reference ranking to 35 in the most stable ranking.

We also study stability for an acceptable region close to the reference ranking. We choose 0.998 cosine similarity ($\theta = \pi/50$) around the weight vector of the reference score function. There are 22 feasible rankings in this acceptable region; their stability distribution is shown in Figure 4. Even in this narrow region of interest, the reference ranking is far below the maximum stability.

FIFA Rankings: Next, we evaluate the higher-dimensional FIFA rankings that are used for important decisions such as seeding different tournaments, including the 2018 FIFA World Cup. We focus on an acceptable region defined by 0.999 cosine similarity ($\theta = \pi/100$) around the reference weights used by FIFA, i.e. $w = \langle 1, 0.5, 0.3, 0.2\rangle$. Using the MD algorithm GET-NEXT_{md}, we conducted 100 operations to get the distribution of the top-100 stable rankings around the reference weight vector. We considered 10,000 samples drawn using Algorithm 3 for estimations. Figure 5 shows the distribution of stable rankings. First, since $d = 4$, there are many feasible rankings, even in such a narrow region of interest, with a significant drop in stability after the most stable rankings, as was the case for CSMetrics.

Perhaps the most interesting observation is that *the reference ranking did not appear in the top-100 stable rankings* (as a result it is not highlighted in Figure 5). While FIFA advertises this ranking as "a reliable measure for comparing the national A-teams" [41], our finding questions FIFA's trust in such an unstable ranking for making important decisions such as seeding the world cup. To highlight an example, while Tunisia holds a higher rank than Mexico in the reference ranking, Mexico is ranked higher in the most stable ranking. This supports the many critics that have questioned the validity of FIFA rankings in the recent past. Examples of controversial rankings include Brazil at 22 in 2014, the U.S. at 4 in 2006, and Belgium at 1 in 2015.

6.3 Algorithm performance

To evaluate the efficiency of our algorithms, we use the *Blue Nile* dataset, which consists of $116,300$ items over 5 ranking attributes. To vary the number of items, we take random samples; to vary the number of dimensions to $d = k < 5$ we project the first k attributes. We equally weight the attributes as the default function.

2D: First we study the impact of n, the database size, on the efficiency of SV_{2D} to compute the stability of the default ranking (i.e. $w = \langle 1, 1\rangle$. We vary n from 100 to $100,000$, measuring both time and the stability of the default ranking (Figure 6). As stated in § 3 computing the stability of a ranking in 2D is in $O(n)$. We find that the running time increased linearly and was only 0.12 seconds for the largest data set. We observe that the stability quickly drops from the order of 10^{-2} for $n = 100$ to less than 10^{-6} for $n = 100$K. This is because the number of ordering exchanges increase by n, leading to many small regions and low stability measures.

Next, we study the performance of GET-NEXT_{2D} under different database sizes. The first GET-NEXT call needs to perform ray sweeping to construct the heap of ranking regions, while subsequent calls just remove the next most stable ranking from the heap. Therefore, in Figure 7, we separate the first call from subsequent calls. As expected, as the number of items increases, the number of ordering exchanges increases and therefore, the efficiency of the operator drops. Also, subsequent GET-NEXT calls are significantly faster than the first. Still, even for the largest setting (i.e., $n = 100$K), the first call to the operator took less than 10 seconds.

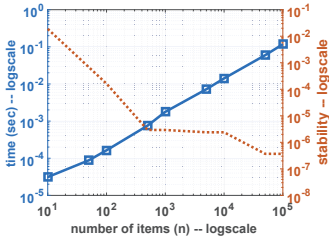


Figure 6: 2D: Stability verification, Impact of dataset size (n)

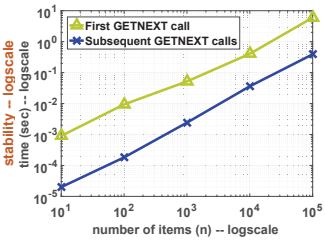


Figure 7: 2D: GET-NEXT, Impact of dataset size (n)

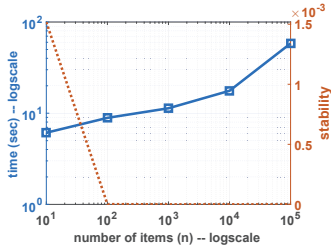


Figure 8: MD: Stability verification, Impact of dataset size

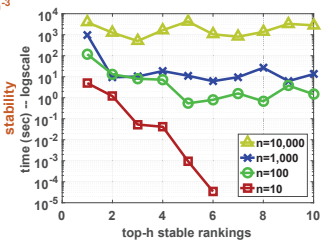


Figure 9: MD: stable rankings, Impact of dataset size (n)

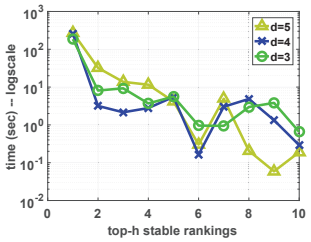


Figure 10: MD: stable rankings, impact of number of attributes (d).

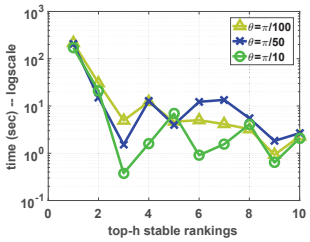


Figure 11: MD: stable rankings, impact of width of region of interest (θ).

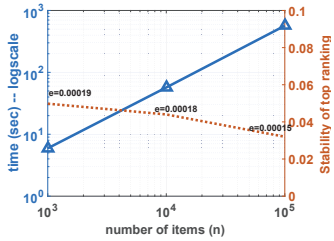


Figure 12: GET-NEXT$_r$: stable top-k items, impact of dataset size (n).

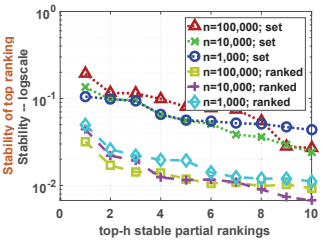


Figure 13: GET-NEXT$_r$: stable top-k items, impact of dataset size (n) on stability.

MD: Next we study the performance of the stability verification algorithm, SV, and the MD algorithm, GET-NEXT$_{md}$, for producing the stable rankings. We vary the number of items (n), number of attributes (d), and width of the region of interest (θ).

First we evaluate stability verification (Figure 8). Choosing the default weight vector $w = \langle 1, 1, 1\rangle$, while setting $d = 3$, we initiate the stability oracle with a set of 1M samples drawn from the entire function space $\mathcal{U}$ and vary the number of items from 100 to 10K. The stability verification algorithm in MD needs to iterate over the sample points, counting those falling inside the ranking region, described as a set of $O(n)$ constraints. This took less than a minute for $n = 10$K. On the other hand, the stability of the default ranking immediately drops to near zero, even for 100 items. Compared to 2D, this is due to the increase in the complexity of the function space.

Next, we evaluate the performance of the GET-NEXT$_{md}$ operator under varying n, d, and θ (the width of $\mathcal{U}^*$). The default values are $n = 100$, $d = 3$ and $\theta = \pi/100$. We use a set of 100K samples from the region of interest for the measurements. Figures 9, 10, and 11 show the performance of the GET-NEXT$_{md}$ algorithm for the top 10 stable rankings for varying (i) number of items from $n = 10$ to $n = 10$K, (ii) number of attributes from $d = 3$ to $d = 5$, and (iii) width of the region from $\theta = \pi/10$ to $\pi/100$, respectively.

Overall, the running time decreases for subsequent calls. This is because the algorithm initially builds part of the arrangement before it finds the most stable ranking. As shown in Figure 9, every GET-NEXT call took up to several thousand seconds for the large setting of 10K items. That is because of the complexity of the arrangement of $O(n^2)$ ordering exchanges which makes even the focus on the most stable region inefficient. In such complex situations, all the regions are very small and unstable, as too many ordering exchanges pass through a narrow region. Nevertheless, in a large setting, it is more reasonable to consider the top-k items rather than the complete list. Our proposal for such settings is the randomized operator.

The next observation is that the running times are similar for different values of d and θ. While the complexity of the space changes for the $O(n^2)$ ordering exchanges, the search is still done using a fixed set of samples and, using the Partition algorithm, only the subset of points falling into a region are used for constructing the arrangement in it. As a result, the lines in Figures 10 and 11 show similar behaviors for different settings.

Randomized algorithm: As the complexity of arrangement increases, GET-NEXT$_{md}$ becomes less efficient. On the other hand, when the number of items is large, users may be more interested in top-k items: that is they may focus on the top of the ranked list. In § 4.3, we proposed a Monte-Carlo-based randomized algorithm to handle these cases. As the last set of experiments, we evaluate the performance of the randomized algorithm under different settings. We look at two models of top-k items, (i) ranked top-k items and (ii) top-k sets. In (i) the user is interested in the orderings among the top-k items, whereas in (ii) the user's interest is in the top-k sets in the ranking lists. We consider a budget of $5,000$ samples (from the region of interest) for the first GET-NEXT$_r$ call and $1,000$ for subsequent calls. The default values are number of items $n = 10,000$, number of attributes $d = 3$, the width of the region of interest $\theta = \pi/50$, and $k = 10$.

Figures 12 and 15 show the running time of the first GET-NEXT$_r$ call and the stability of the most stable ranking for varying the number of items from 1K to 100K, and the number of attributes from 3 to 5, while considering the ranked top-k items (the running times are similar for top-k sets). The plots verify the scalability of the randomized algorithm for large settings, as it took a few minutes for 100K items while the running times for $d = 3$, 4, and 5 are similar. Looking at right y-axis in Figure 12, despite the increase in the number of items from 1K to 100K, the stability of the most stable ranked top-k did not noticeably decrease. This confirms the feasibility of considering the top-k items for the large settings.

Also, to evaluate the scalablity of our proposal for a very large setting, we use the DoT dataset and set the budget to 5K samples for the first GET-NEXT$_r$ call and 1K for subsequent calls. Similar to the previous experiment, we set d to 3, the width of the region of interest to $\theta = \pi/50$, and consider top-k sets for $k = 10$, while varying the number of items up to one million. Figure 14 shows the performance of the algorithm for each setting. As expected the run-time linearly increases with the number of items, while it takes on the order of an hour for the largest setting. Note that the number of samples plays an important role in the performance of the algorithm: the higher the sampling budget, the more accurate

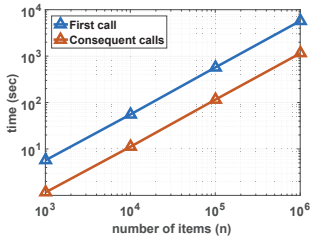


Figure 14: DoT, GET-NEXT$_r$: stable top-k items, Impact of dataset size (n)

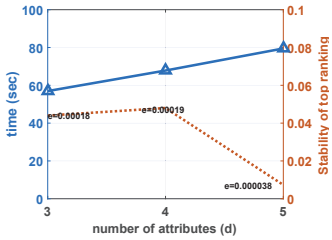


Figure 15: GET-NEXT$_r$: stable top-k items, Impact of number of attributes (d)

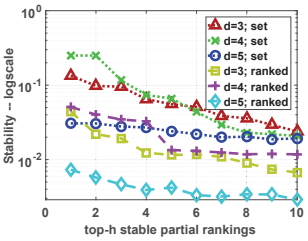


Figure 16: GET-NEXT$_r$: stable top-k items, Impact of number of attributes (d)

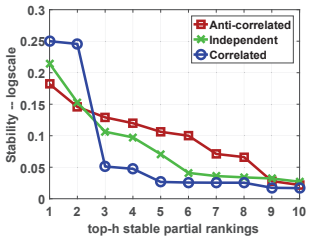


Figure 17: Synthetic data, GET-NEXT$_r$: stable top-k items, Impact of correlation

the results, and the run-time is also higher. This can be confirmed by comparing the lines for the first call (5000 samples) versus the consequent calls (1000 samples) of the primitive.

Figures 13 and 16 show the stability of the top-10 stable rankings for both ranked top-k items and the top-k sets. In both figures, the top-k sets are more stable than the top-k rankings. The reason is that the top-k sets do not consider the ordering between the items, and thus the variety of possible outcomes is reduced compared to top-k rankings. An observation in Figure 13 is the similarity of the stability distributions for different numbers of items, which, again, confirms the feasibility of considering the top-k items for large settings. In Figure 16, as expected, the number of attributes have a negative correlation with the stability of the top-k items.

The effect of attribute correlation Finally, we study the effect of attribute correlation on the stability of the rankings. To do so, we use the synthetic datasets (independent, correlated, anti-correlated), each containing 10K items and $d = 3$ attributes. Using a budget of 5000 samples for evaluation, we set the width of the region of interest to $\theta = \pi/50$, and k to 10. Figure 17 shows the stability of the most stable top-k sets. We find that strong attribute correlation leads to a greater skew in the distribution of stable regions: the most stable regions have higher stability. This is illustrated in Figure 17 where we see that the correlated dataset results in the greatest maximum stability but also has the steepest slope as we descend from the most-stable to the 10th-most-stable top-k set. Accordingly, the independent dataset has a slightly lower stability most-stable region with a reduced slope, and the anti-correlated dataset displays the least skew in the stabilities. This is expected since, in a dataset with highly correlated attributes, we are more likely to find items in dominance relationships with one another (i.e. the attributes of item X are greater than those of Y in all, or nearly all, dimensions). In that case, those items are almost always ranked in one way, reducing the number of feasible rankings, and resulting in a large number of relatively unstable rankings and a few highly stable rankings.

7. RELATED WORK

Given a dataset with multiple attributes, ordering the items and choosing a subset to support decision making is challenging. This has motivated a rich body of work on ranking [24, 44–47], top-k [22, 23], and skyline queries [8, 10, 48, 49]. Broadly speaking, ranking and top-k are employed when a user's preference in the form of a scoring function is available, while skyline queries are used when only the scoring attributes are known, but the scoring function is left unspecified. To the best of our knowledge, no existing work considers a *range of acceptable scoring functions*, and discovers *stable rankings* within that range. In our work, the region of interest can be as narrow as a single scoring function, or as wide as the entire space of scoring functions.

The work on ranking and top-k includes managing datasets with uncertainty and noise with respect to item existence or their attribute values [50–52], and using human computation to fill in missing information [15]. While the work on probabilistic rankings considers uncertainty in the data, in our work we focus on uncertainty in the scoring function that reflects a user's preferences. There has been extensive effort on efficient processing of top-k queries [22]: threshold-based algorithms [23] consider parsing presorted lists along each attribute, view-based approaches [53, 54] utilize presorted lists that are built on various angles of the function space, and indexing-based methods [55] create layers of extreme points for efficient processing of queries. Ranking has also been considered in spatial databases [56].

In the absence of a scoring function, the effort is on finding the set of potentially high-scoring representatives such as the skyline [8, 14, 57], also known as the pareto-optimal set [15] — the set of non-dominated items. Since the number of skyline points can be large [48], works such as [9–12, 32, 48] look for smaller representative subsets. For example, [9] finds a subset of k skyline points that dominate the maximum number of points, while [12] picks the top-k combinatorial skyline based on an importance ordering of the attributes. Also, extensive recent work [10, 11] aims to find a small subset of the skyline that minimizes some notion of regret. A key difference between the stable top-k set and these proposals is that a top-k set is not necessarily a subset of the skyline.

In this paper, we used notions such as half-space, duality, and arrangement from combinatorial geometry that are explained in detail in [7, 58]. Arrangement of hyperplanes, its complexity, construction, and applications are studied in [7, 18–21]. Geometric aspects of top-k queries are presented in a recent tutorial [59].

8. FINAL REMARKS

In this paper, we studied the problem of obtaining stable rankings for databases with multiple attributes when the rankings are produced through a goodness score for each item as a weighted sum of its attribute values. A stable ranking is more meaningful than one susceptible to small changes in scoring weights, and hence engenders greater trust. We developed a framework that gives consumers the facility to assess the stability of a ranking and enables producers to discover stable rankings. We devised an unbiased function sampler that enables Monte-Carlo methods. We designed a randomized algorithm for the problem that works both for the complete ranking of items, as well as the top-k partial rankings. The experiments on three real datasets demonstrated the validity of our proposal. Our current definition of stability considers two rankings to be different if they differ in one pair of items. An alternative is to allow minor changes in the ranking. Similarly, we note that a weight vector is a single point in a stable region. It would be nice, for some applications, to characterize the boundaries of the stable region. We will consider these in future work.

9. REFERENCES

[1] M. Gladwell. The order of things: What college rankings really tell us. *The New Yorker Magazine*, Feb 14, 2011.

[2] N. A. Bowman and M. N. Bastedo. Anchoring effects in world university rankings: exploring biases in reputation scores. *Higher Education*, 61(4):431–444, 2011.

[3] J. Monks and R. G. Ehrenberg. The impact of us news and world report college rankings on admission outcomes and pricing decisions at selective private institutions. Technical report, National Bureau of Economic Research, 1999.

[4] A. Langville and C. Meyer. *Who's #1? The Science of Rating and Ranking*. Princeton University Press, 2012.

[5] K. Yang, J. Stoyanovich, A. Asudeh, B. Howe, H. Jagadish, and G. Miklau. A nutritional label for rankings. In *SIGMOD*, pages 1773–1776. ACM, 2018.

[6] CSMetrics. `www.csmetrics.org/`. [Online; accessed April 2018].

[7] H. Edelsbrunner. *Algorithms in combinatorial geometry*, volume 10. Springer Science & Business Media, 2012.

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

[9] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang. Selecting stars: The k most representative skyline operator. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 86–95. IEEE, 2007.

[10] D. Nanongkai, A. D. Sarma, A. Lall, R. J. Lipton, and J. Xu. Regret-minimizing representative databases. *PVLDB*, 3(1-2):1114–1124, 2010.

[11] S. Chester, A. Thomo, S. Venkatesh, and S. Whitesides. Computing k-regret minimizing sets. *PVLDB*, 7(5):389–400, 2014.

[12] I.-F. Su, Y.-C. Chung, and C. Lee. Top-k combinatorial skyline queries. In *International Conference on Database Systems for Advanced Applications*, pages 79–93. Springer, 2010.

[13] A. Asudeh, H. Jagadish, J. Stoyanovich, and G. Das. Designing fair ranking schemes. In *SIGMOD*. ACM, 2019.

[14] A. Asudeh, S. Thirumuruganathan, N. Zhang, and G. Das. Discovering the skyline of web databases. *PVLDB*, 9(7):600–611, 2016.

[15] A. Asudeh, G. Zhang, N. Hassan, C. Li, and G. V. Zaruba. Crowdsourcing pareto-optimal object finding by pairwise comparisons. In *CIKM*, pages 753–762. ACM, 2015.

[16] A. Asudeh, H. Jagadish, G. Miklau, and J. Stoyanovich. On obtaining stable rankings. *CoRR*, abs/1804.10990, 2018.

[17] M. E. Dyer and A. M. Frieze. On the complexity of computing the volume of a polyhedron. *SIAM Journal on Computing*, 17(5):967–974, 1988.

[18] P. Orlik and H. Terao. *Arrangements of hyperplanes*, volume 300. Springer, 2013.

[19] B. Grünbaum. Arrangements of hyperplanes. In *Convex Polytopes*. Springer, 2003.

[20] V. V. Schechtman and A. N. Varchenko. Arrangements of hyperplanes and lie algebra homology. *Inventiones mathematicae*, 106(1), 1991.

[21] P. K. Agarwal and M. Sharir. Arrangements and their applications. *Handbook of computational geometry*, 2000.

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

[23] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4), 2003.

[24] A. Asudeh, N. Zhang, and G. Das. Query reranking as a service. *PVLDB*, 9(11):888–899, 2016.

[25] Y. D. Gunasekaran, A. Asudeh, S. Hasani, N. Zhang, A. Jaoua, and G. Das. QR2: A third-party query reranking service over web databases. In *ICDE*, pages 1653–1656. IEEE, 2018.

[26] H. Blaker. Confidence curves and improved exact confidence intervals for discrete distributions. *Canadian Journal of Statistics*, 28(4):783–798, 2000.

[27] F. J. Hickernell, L. Jiang, Y. Liu, and A. B. Owen. Guaranteed conservative fixed width confidence intervals via monte carlo sampling. In *Monte Carlo and Quasi-Monte Carlo Methods 2012*, pages 105–128. Springer, 2013.

[28] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American statistical association*, 58(301):13–30, 1963.

[29] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *The Annals of Mathematical Statistics*, pages 493–507, 1952.

[30] C. P. Robert. *Monte carlo methods*. Wiley Online Library, 2004.

[31] R. Durrett. *Probability: theory and examples*. Cambridge university press, 2010.

[32] A. Asudeh, A. Nazi, N. Zhang, G. Das, and H. Jagadish. RRR: Rank-regret representative. In *SIGMOD*. ACM, 2019.

[33] M. E. Muller. A note on a method for generating points uniformly on n-dimensional spheres. *Communications of the ACM*, 2(4), 1959.

[34] G. Marsaglia et al. Choosing a point from the surface of a sphere. *The Annals of Mathematical Statistics*, 43(2), 1972.

[35] H. Cramér. *Mathematical methods of statistics (PMS-9)*, volume 9. Princeton university press, 2016.

[36] S. Lucidl and M. Piccioni. Random tunneling by means of acceptance-rejection sampling for global optimization. *Journal of optimization theory and applications*, 62(2):255–277, 1989.

[37] L. Devroye. Sample-based non-uniform random variate generation. In *Proceedings of the 18th conference on Winter simulation*, pages 260–265. ACM, 1986.

[38] K. Fischer, B. Gärtner, and M. Kutz. Fast smallest-enclosing-ball computation in high dimensions. In *European Symposium on Algorithms*, pages 630–641. Springer, 2003.

[39] S. Li. Concise formulas for the area and volume of a hyperspherical cap. *Asian Journal of Mathematics and Statistics*, 4(1):66–70, 2011.

[40] T. F. I. de Football Association (FIFA). Fifa rankings. `www.fifa.com/fifa-world-ranking/ranking-table/men/index.html`. [Online; accessed April 2018].

[41] FIFA. Fifa/coca-cola world ranking procedure. `http://www.fifa.com/fifa-world-ranking/procedure/men.html`, 28 March 2008.

[42] BlueNile. `www.bluenile.com/diamond-search?` [Online; accessed Feb. 2018].

[43] US Department of Transportation's dataset. `http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time`.

[44] F. Geerts, H. Mannila, and E. Terzi. Relational link-based ranking. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 552–563. VLDB Endowment, 2004.

[45] S. Chaudhuri and G. Das. Keyword querying and ranking in databases. *PVLDB*, 2(2):1658–1659, 2009.

[46] R. Agrawal, R. Rantzau, and E. Terzi. Context-sensitive ranking. In *SIGMOD*, pages 383–394. ACM, 2006.

[47] P. K. Agarwal, L. Arge, J. Erickson, P. G. Franciosa, and J. S. Vitter. Efficient searching with linear constraints. *JCSS*, 61(2), 2000.

[48] A. Asudeh, A. Nazi, N. Zhang, and G. Das. Efficient computation of regret-ratio minimizing set: A compact maxima representative. In *SIGMOD*, pages 821–834. ACM, 2017.

[49] J. Stoyanovich, W. Mee, and K. A. Ross. Semantic ranking and result visualization for life sciences publications. In *ICDE*, pages 860–871, 2010.

[50] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic ranking of database query results. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 888–899. VLDB Endowment, 2004.

[51] J. Li, B. Saha, and A. Deshpande. A unified approach to ranking in probabilistic databases. *PVLDB*, 2(1):502–513, 2009.

[52] J. Li and A. Deshpande. Ranking continuous probabilistic datasets. *PVLDB*, 3(1-2):638–649, 2010.

[53] V. Hristidis and Y. Papakonstantinou. Algorithms and applications for answering ranked queries using ranked views. *The VLDB Journal*, 13(1):49–70, 2004.

[54] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis. Answering top-k queries using views. In *Proceedings of the 32nd international conference on Very large data bases*, pages 451–462. VLDB Endowment, 2006.

[55] Y.-C. Chang, L. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The onion technique: indexing for linear optimization queries. In *SIGMOD*, 2000.

[56] G. R. Hjaltason and H. Samet. Ranking in spatial databases. In *SSTD*. Springer, 1995.

[57] M. F. Rahman, A. Asudeh, N. Koudas, and G. Das. Efficient computation of subspace skyline over categorical domains. In *CIKM*, pages 407–416. ACM, 2017.

[58] M. De Berg, O. Cheong, M. Van Kreveld, and M. Overmars. *Computational Geometry: Introduction*. Springer, 2008.

[59] K. Mouratidis. Geometric approaches for top-k queries. *PVLDB*, 10(12):1985–1987, 2017.