Designing Fair Ranking Schemes

Abolfazl Asudeh*, H. V. Jagadish†, Julia Stoyanovich‡, Gautam Das§

*†University of Michigan; ‡New York University; §University of Texas at Arlington
{asudeh,jag}@umich.edu; stoyanovich@nyu.edu; gdas@uta.edu

ABSTRACT

Items from a database are often ranked based on a combination of criteria. The weight given to each criterion in the combination can greatly affect the fairness of the produced ranking, for example, preferring men over women. A user may have the flexibility to choose combinations that weigh these criteria differently, within limits. In this paper, we develop a system that helps users choose criterion weights that lead to greater fairness. We consider ranking functions that compute the score of each item as a weighted sum of (numeric) attribute values, and then sort items on their score. Each ranking function can be expressed as a point in a multi-dimensional space. For a broad range of fairness criteria, including proportionality, we show how to efficiently identify regions in this space that satisfy these criteria. Using this identification method, our system is able to tell users whether their proposed ranking function satisfies the desired fairness criteria and, if it does not, to suggest the smallest modification that does. Our extensive experiments on real datasets demonstrate that our methods are able to find solutions that satisfy fairness criteria effectively (usually with only small changes to proposed weight vectors) and efficiently (in interactive time, after some initial pre-processing).

KEYWORDS

Data Ethics; Responsible Data Management; Fairness

ACM Reference Format:
Abolfazl Asudeh, H. V. Jagadish, Julia Stoyanovich, Gautam Das. 2019. Designing Fair Ranking Schemes. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3300079

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3300079

1 INTRODUCTION

Ranking of individuals is commonplace today, and is used, for example, to establish credit worthiness, desirability for college admissions and employment, and attractiveness as dating partners. Properly, topics such as ranking, top-k query processing, and building indexes to efficiently answer such queries, have recently been increasingly relevant to database research. A prominent family of ranking schemes are score-based rankers, which compute the score of each individual from some database $\mathcal{D}$, sort the individuals in decreasing order of score, and finally return either the full ranked list, or its highest-scoring sub-set, the top-k. Many score-based rankers compute the score as a linear combination of attribute values, with non-negative weights.

This sort of linear-weighted scoring and ranking is ubiquitous. Many sports use such schemes. For example, tennis players have an ATP rank based on a score that weights each level of success (winner, finalist, semi-finalist, and so on) at each type of tournament, and adds these up. A score with more serious implications is the credit score that each person has in many countries, meant to indicate creditworthiness. Even in the context of academic research, we see such scoring: many funding agencies compute a score for a research proposal as a weighted sum of scores for its attributes.

Because of the potential impact of such rankings on individuals and on population groups, issues of algorithmic bias and discrimination are coming to the forefront of societal and technological discourse [1]. In their seminal work Friedman and Nissenbaum [2] define a biased computer system as one that (1) systematically and unfairly discriminates against some individuals or groups in favor of others, and (2) joins this discrimination with an unfair outcome.

We desire a ranking scheme that is fair, in the sense that it mitigates *preexisting bias with respect to a protected feature* embodied in the data. In line with prior work [3–7], a protected feature denotes membership of an individual in a legally-protected category, such as persons with disabilities, or under-represented minorities by gender or ethnicity. We refer to such categories (e.g., minority ethnicity) as *protected groups*, and to the attributes that define them (e.g., ethnicity) as *sensitive attributes*. Interpreting the definition of Friedman and Nissenbaum [2] for rankings, discrimination occurs

when the outcome is systematic and unfavorable, for example, when minority ethnicity or female gender systematically lead to placing individuals at lower ranks.

Numerous fairness definitions have been considered in the recent literature [7, 8]. A useful dichotomy is between *individual fairness* and *group fairness*, also known as statistical parity. The former requires that similar individuals be treated similarly, while the latter requires that demographics of those receiving a particular outcome are identical or similar to the demographics of the population as a whole [8]. These two requirements represent intrinsically different world views, and accommodating both requires trade-offs [9]. We focus on group fairness in this paper. While our techniques apply to a broad range of group fairness criteria, to make our discussion concrete we will define fairness in terms of minimum bounds on the number of selected members of a protected group at the top-k, for some reasonable value of k [10].

Designing a ranking scheme amounts to selecting a set of weights, one for each attribute. In some situations, we may have access to good labeled training data, and be able to use machine learning techniques. But in many situations, such as to rank tennis players, research proposals, or academic departments, we do not have access to any ground truth. Therefore, we resort to subjectively selected weights in a simple additive scoring function. Such a function is typically defined over a handful of attributes, due to the cognitive burden of selecting the scoring criteria and coming up with an appropriate weight vector. The question we address in this paper is how to introduce fairness into this subjective weight selection process. Consider an example.

Example 1. *A college admissions officer is designing a ranking scheme to evaluate a pool of applicants, each with several potentially relevant attributes. For simplicity, let us focus on two of these attributes — high school GPA and SAT score. Suppose that our fairness criterion is that the admitted class comprise at least 40% women. As the first step, to make the score components comparable, GPA and SAT scores may be normalized and standardized. We will denote the resulting values g for GPA and s for SAT. The admissions officer may believe a priori that g and s should have an approximately equal weight, computing the score of an applicant $t \in \mathcal{D}$ as $f(t) = 0.5 \times s + 0.5 \times g$, ranking the applicants, and returning the top 500 individuals.*

*Upon inspection, it may be determined that an insufficient number of women is returned among the top-k: at least 200 women were expected to be among the top-*500*, and only 150 were returned, violating our fairness constraint. This violation may be due to a gender disparity in the data: in 2014, women scored about 25 points lower on average than men in the SAT test [11]. Note that the admissions officer was not looking at the sensitive attribute (gender, in our example), and proposed a scoring function that is not obviously biased against women: the lack of fairness is only observed in the outcome.*

Our goal in this paper is to build a system that will assist the admissions officer in identifying alternative scoring functions that meet the fairness constraint and are close to the original function f in terms of attribute weights, thereby reflecting the admission officer's a priori notion of quality. After a few cycles of such interaction with the system, the admissions officer may choose $f'(t) = 0.45 \times s + 0.55 \times g$ as the final scoring function.

As underscored by Example 1, we wish to produce results that are both fair — as stated by the fairness constraints, and of high quality — as stated by the initial scoring function weights. These initial scoring function weights will only approximate quality, for two reasons. First, observational data usually contains imperfect proxies of "true" aspects of quality (e.g., SAT score vs. intelligence, and GPA vs. grit) [9]. Second, future outcomes cannot be perfectly predicted based on present observations, irrespective of whether a simple score-based ranker or a complex learned model is used.

Rather than adjusting the scoring function, one could meet fairness constraints by having different cutoff scores for different demographic groups. For example, the admissions officer in Example 1 could have stuck with the original function, and used a lower score threshold for admitting women compared to the one for men. While such a fix is technically easy, it is illegal in many jurisdictions, because it amounts to *disparate treatment* — to the explicit use of a protected characteristic such as gender or race to make decisions. Our proposal in this paper navigates the trade-off between disparate treatment and *disparate impact* — providing outputs that hurt members of a protected group more frequently than members of other groups. If a small adjustment to the weights of the scoring function can achieve fairness, that may be both preferable for legal reasons, and acceptable from the point of view of utility, particularly since the original weights likely were approximate values chosen subjectively.

One may also argue that if a scoring function is specified by a human expert, algorithmic bias is not an issue. Yet, there is a long history of people using justifiable models to be able to discriminate. For example, legacy was added to the variables considered at admission, and given a high weight, to keep down the number of Jewish students, since "too many" of them would have been admitted considering academic achievement alone [12, 13].

Of course, our proposed methods will not prevent institutional racism and other kinds of intentional discrimination. That said, it is increasingly recognized that this challenge cannot be addressed by technology alone, and that responsibility to determine the context of use of a tool should fall squarely on legal and policy frameworks. Rather than dictating a particular choice of policy, we enable decision makers to

transparently enact a policy of their choosing, by supporting an explicit specification of fairness constraints, and incorporating them into ranking scheme design. *Transparency by design* is now required by legal frameworks like the New York City algorithmic transparency law [14].

Whether a fairness criterion is met can only be assessed with respect to a specific dataset. However, we can assess the fairness of a scoring function if we have a characterization of the distribution of data points in any data set to which the function will be applied. In other words, we can work with a representative "training" data set to design a fair scoring function. We can then expect this scoring function to remain fair until the data distribution changes substantially.

Our technical goal is to build a system to assist a human designer of a scoring function in tuning attribute weights to achieve fairness. Since this tuning process does not occur too often, it may be acceptable for it to take some time. However, we know that humans are able to produce superior results when they get quick feedback in a design or analysis loop. Indeed, it is precisely this need that is a central motivation for OLAP, rather than having only long-running analytics queries. Ideally, a designer of a ranking scheme would want the system to support her work through interactive response times. Our goal is to meet this need, to the extent possible.

As we will later show, it is computationally challenging to find a scoring function that is both fair and close to the user-specified scoring function, particularly when more than two scoring attributes must be considered. In order to overcome this challenge, we introduce techniques from combinatorial geometry and, since the direct application of existing algorithm does not scale in practice, we propose the arrangement tree data structure. We then propose a grid-partitioning preprocessing method that enables approximate query answering. Preprocessed data can be reused if the dataset does not change significantly over time, thus amortizing the cost of pre-computation. In addition to the offline preprocessing method, we also study sampling for on-the-fly query answering based on it. We provide a negative result that states that methods based on function sampling cannot provide any guarantees for the discovery of an approximate solution.

In the remainder of this paper, we will present a *query answering system* that assists the user in designing fair score-based rankers. The system first preprocesses a dataset of candidates off-line and then handles user requests in real time. The user specifies a *query* in the form of a scoring function f that associates non-negative weights with item attributes and computes items scores. Items are then sorted on their scores. We assume the existence of a *fairness oracle* that, given an ordered list of items, returns *true* if the list meets fairness criteria and so is *satisfactory*, and returns $false$ otherwise. The fairness oracle relies on its knowledge about the supported fairness definitions to achieve scalability. If the list of items is found to be unsatisfactory, we suggest to the user an alternative scoring function f' that is both satisfactory and close to the query f. The user may accept f', or she may decide to manually adjust the query and invoke our system once again.

Summary of contributions: We propose a system that assists the user in designing fair score-based ranking schemes.

- We characterize the space of linear scoring functions (with their corresponding weight vectors), and characterize portions of this space based on the ordering of items induced by these functions. We develop algorithms to determine boundaries that partition the space into regions where the desired fairness constraint is satisfied, called *satisfactory regions*, and regions where the constraint is not satisfied. Given a user's query, in the form of a scoring function f, we develop efficient exact algorithms to find the nearest scoring function f' that satisfies the constraint, or to state that the constraint is not satisfiable.
- We develop approximation algorithms for efficiently identifying and indexing satisfactory regions. We also introduce sampling heuristics for on-the-fly query processing in large-scale settings.
- We conduct an extensive experimental evaluation on real datasets that validates our proposal.

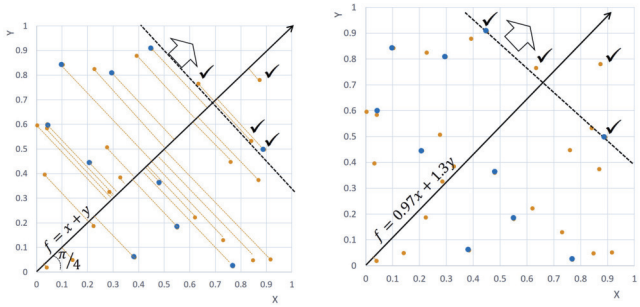
While fairness in algorithmic systems is an active area of research [7], our work is among a small handful of studies that focus on fairness in ranking [5, 6, 10], and is *the first to support the user in designing fair ranking schemes.*

Finally, we note that the methods of this paper have applications beyond fairness, and can be used more generally to engineer ranking functions that satisfy input constraints. We choose to focus on fairness to make our discussion concrete, and to contribute to the important emerging area of research in fairness, accountability and transparency (FAT).

2 PRELIMINARIES

Data model: We consider a dataset $\mathcal{D}$ of n items, each with d scalar scoring attributes. (Additional non-scalar attributes are considered in the fairness model.) We represent an item t as a d-long vector of scoring attributes, $\langle t[1], t[2], \ldots, t[d]\rangle$. Without loss of generality, we assume that each scoring attribute is a non-negative number and that larger values are preferred. This assumption is straightforward to relax with some additional notation and bookkeeping.

Ranking model: We focus on the class of linear scoring functions that use a weight vector $\vec{w} = \langle w_1, w_2, \ldots, w_d\rangle$ to compute a goodness score $f_{\vec{w}}(t)$ of item t as $\Sigma_{j=1}^{d} w_j t[j]$. To simplify notation, we use $f(t)$ to refer to $f_{\vec{w}}(t)$. Without loss of generality, we assume each weight $w_j \geq 0$. The scores of items are used for ranking them. We assume that an item



(a) function $f = x + y$ **(b) function $f = 0.97x + 1.3y$**

Figure 1: Effect of weight choice on output fairness

with a higher score outranks an item with a lower score. We denote the ranking of items in $\mathcal{D}$ based on f with $\nabla_f(\mathcal{D})$.

Our ranking model has an intuitive geometric interpretation: items are represented by points in $\mathbb{R}^d$, and a linear scoring function f is represented by a ray starting from the origin and passing through the point $\vec{w} = \langle w_1, w_2, \ldots, w_d \rangle$. The score-based ordering of the points induced by f corresponds to the ordering of their projections onto the ray for $\vec{w}$. Figure 1 shows the items of an example dataset with $d = 2$ as points in $\mathbb{R}^2$. The function $f = x + y$ is represented in Figure 1a as a ray stating from the origin and passing through the point $\langle 1, 1 \rangle$. Projections of the points onto the ray specify their ordering based on f.

Note that the rays corresponding to functions f and f' are the same if the weight vector of f' is a linear scaling of the weight vector of f. This is because a weight vector $\vec{w} = \langle w_1, w_2, \ldots, w_d \rangle$ induces the same ordering on the items as does its linear scaling $\vec{w'} = \langle c \times w_1, c \times w_2, \ldots, c \times w_d \rangle$, for any $c > 0$. Hence, the *distance between two functions* f and f' is considered as the angular distance between their corresponding rays in $\mathbb{R}^d$. For example, the distance between $f = x + y$ and $f' = 100x + 100y$ is 0, while the distance between $f = x + y$ and $f'' = x$ is $\frac{\pi}{4}$, the angular distance between the ray corresponding to f in Figure 1a and the x-axis. For every item $t \in \mathcal{D}$, *contour* of t on f is the value combinations in $\mathbb{R}^d$ with the same score as $f(t)$ [15, 16]. For linear functions, the contour of an item t is the hyperplane h that is perpendicular to the ray of f and passes through t.

Fairness model: We adopt a general ranked fairness model, in which a fairness oracle O takes as input an ordered list of items from $\mathcal{D}$, and determines whether the list meets fairness constraints: $O : \nabla_f(\mathcal{D}) \rightarrow \{\top, \bot\}$. A scoring function f that gives rise to a fair ordering over $\mathcal{D}$ is said to be *satisfactory*.

In addition to scoring attributes, discussed in the data model, items are associated with one or several type attributes. A type corresponds to a *protected feature* such as gender or race. We discussed bias with respect to a protected feature in the introduction. In the example in Figure 1, there is a single binary type attribute, denoted by blue and orange colors. Suppose that the fairness oracle returns true if the top-4 items contain an equal number of items of each type. Function $f = x + y$ in Figure 1a is not satisfactory as it has 3 orange points and one blue point in its top-4, while $f' = 0.97x + 1.3y$ in Figure 1b contains two points of each type in its top-4 and is satisfactory.

While our fairness model is general, in our experimental evaluation we focus on fairness constraints that were considered in recent literature [4, 6, 10]. We work with proportionality constraints that bound the number of items belonging to a particular demographic group (as represented by an assignment of a value to a categorical type attribute) at the top-k, for some given value of k.

2.1 Problem statement

A given query f, with a weight vector $\vec{w}$, may not satisfy the required fairness constraints. Our problem is to propose a scoring function f', with a weight vector similar to that of f, that does satisfy the constraints, if one exists.

Of course, the user may not accept our proposal. Instead, she may try a different weight vector of her liking, which we can again examine and either approve or propose an alternative. The final choice of an acceptable scoring function is up to the user. We now formally state our problem.

Closest Satisfactory Function: *Given a dataset $\mathcal{D}$ with n items over d scalar scoring attributes, a fairness oracle $O : \nabla_f(\mathcal{D}) \rightarrow \{\top, \bot\}$, and a linear scoring function f with the weight vector $\vec{w} = \langle w_1, w_2, \cdots, w_d \rangle$, find the function f' with the weight vector $\vec{w'}$ such that $O(\nabla_{f'}(\mathcal{D})) = \top$ and the angular distance between $\vec{w}$ and $\vec{w'}$ is minimized.*

High-level idea: From the system's viewpoint, the challenge is to propose similar weight vectors that satisfy the fairness constraints, in interactive time. To accomplish this, our solution will operate with an offline phase and then an online phase. In the offline phase, we will process the (representative sample) dataset, and develop data structures that will be useful in the online phase. In the online phase, we will exploit these data structures to interactively propose similar satisfactory weight vectors to assist the human designer. Once the human designer has selected the satisfactory weights, it may continue to be satisfactory, as long as the datasets have roughly the same distribution. (Later, we also develop a function sampling technique that removes the need for pre-processing and can sometimes be effective).

In the next section, we consider the easier to visualize 2D case, in which the dataset contains 2 scalar scoring attributes. The terms and techniques discussed in § 3 will help us in § 4 for developing algorithms for the general multi-dimensional case, where the number of scalar scoring attributes is $d > 2$.

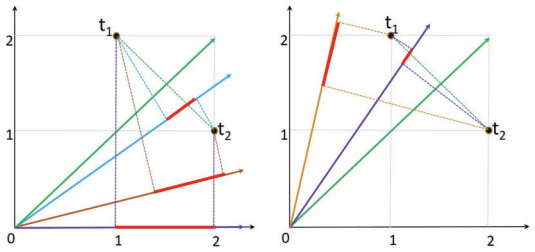


Figure 2: ordering exchange between a pair of points

3 THE TWO-DIMENSIONAL CASE

In this section we consider a simplified version of the problem in which only two scalar attributes (x and y) participate in the ranking. We begin by introducing the central notion of ordering exchange that partitions the space of linear functions into disjoint regions. Then, we use this concept to develop two algorithms: an offline algorithm to identify and index the satisfactory regions, and an online algorithms that can be used repeatedly, as the domain expert interactively tunes weights, to obtain a desired ranking function.

3.1 Ordering exchange

Each item in a 2-dimensional dataset can be represented as a point in $\mathbb{R}^2$, and each ranking function f can be represented as a ray starting from the origin. The ordering of the items is the ordering of their projections on the ray of f. For instance, Figure 1 specifies the projection of the points on the ray of $f = x + y$. One can see that the set of rays between the x and y axes represents the set of possible ranking functions in 2D. Even though an infinite number of rays exists between x and y, the number of possible orderings of n items is limited to $n!$, the number of their permutations. Our central insight is that we do not need to consider every possible ranking function: we only need to consider at most as many as there are orderings of the items, as we discuss next.

Consider two points $t_1\langle 1, 2\rangle$ and $t_2\langle 2, 1\rangle$, shown in Figure 2. The projections of t_1 and t_2 on the x-axis are the points $x = 1$ and $x = 2$, respectively. Hence, the ordering based on $f = x$ is $t_2 \succ t_1$, which denotes that t_2 is preferred to t_1 by f. Moving away from the x-axis towards the y-axis, the distance between the projections of t_1 and t_2 on the ray decreases, and becomes zero at $f = x + y$. Then, moving from $f = x + y$ to the y-axis, the ordering between these two points changes to $t_1 \succ t_2$. As we continue moving towards the y-axis, the distance between the projections of t_1 and t_2 increases, and their order remains $t_1 \succ t_2$. Using this observation, we can partition the set of scoring functions based on their angle with the x-axis into $F_1 = [0, \pi/4]$ and $F_2 = [\pi/4, \pi/2]$, such that for every $f \in F_1$ the ordering is $t_2 \succeq t_1$ and for every $f' \in F_2$ the ordering is $t_1 \succeq t_2$.

We define the *ordering exchange* as the ranking functions that score t_1 and t_2 equally. In 2D, the ordering exchange of a pair of points is at most a single function.

For any specified ordering of items, the fairness constraint either is satisfied or it is not. If this ordering is changed, the satisfaction of the fairness constraint may change as well. Therefore, in the space of possible ranking functions, every boundary between a satisfactory region and an unsatisfactory region must comprise ordering exchange functions.

3.2 Offline processing

Offline processing is for identifying and indexing the satisfactory functions, for efficient answering of online queries. Following the example in Figure 2, we propose a *ray sweeping* algorithm for identifying satisfactory functions in 2D.

To identify the ordering exchanges of pairs of items, we transform items into a dual space [17], where every item t is transformed into the line $\mathrm{d}(t)$, as follows:

$$\mathrm{d}(t) : t[1]x + t[2]y = 1 \tag{1}$$

The ordering of the items based on a function f with the weight vector $\langle w_1, w_2\rangle$ is the ordering of the intersections of the lines $\mathrm{d}(t)$ with the ray starting from the origin and passing through the point $\langle w_1, w_2\rangle$, where the closer intersections to the origin are ranked higher. For example, Figure 4 shows the dual transformation (using Equation 1) of the 2D dataset provided in Figure 3. Therefore, the ordering exchange of a pair t_i and t_j is the intersection of $\mathrm{d}(t_i)$ and $\mathrm{d}(t_j)$. For example, in Figure 4, the ordering exchange of t_1 and t_2 is the top-left intersection (of lines $\mathrm{d}(t_1)$ and $\mathrm{d}(t_2)$).

Using Equation 1, the intersection of the lines $\mathrm{d}(t_i)$ and $\mathrm{d}(t_j)$ can be computed by the following system of equations:

$$\times_{\mathrm{d}(t_i),\mathrm{d}(t_j)} : \begin{cases} t_i[1]x + t_i[2]y = 1 \\ t_j[1]x + t_j[2]y = 1 \end{cases}$$

The ordering exchange is identified by the angle:

$$\theta_{t_i,t_j} = \arctan \frac{t_j[1] - t_i[1]}{t_i[2] - t_j[2]} \tag{2}$$

Now, we use ordering exchanges to design the ray sweeping algorithm 2DRAYSWEEP (Algorithm 1). The algorithm uses a min-heap to maintain the ordering exchanges. It uses the fact that, sweeping from the x to y-axis, at any moment a pair of items that are adjacent in the ordered list may exchange ordering. Therefore, it first orders the items based on the x-axis and gradually updates the order as it sweeps the ray toward the y-axis (angle $\pi/2$), by swapping the order of pairs of items in their ordering exchanges.

The algorithm initially computes the ordering exchanges between all pairs of adjacent items that do not dominate

t_1	1	3.5
t_2	1.5	3.1
t_3	1.91	2.3
t_4	2.3	1.8
t_5	3.2	0.9

Figure 3: A 2D dataset

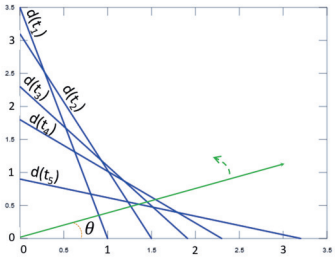


Figure 4: Fig. 3 in dual space

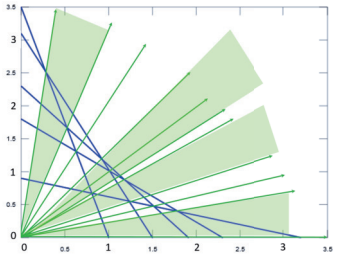


Figure 5: Satisfactory sectors

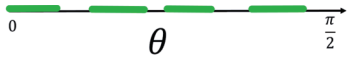


Figure 6: Satisfactory regions

Algorithm 1 2DRAYSWEEP
Input: dataset $\mathcal{D}$ and fairness oracle O
Output: sorted satisfactory regions S

```
1: Ω = ∇_{f_{1,0}}(D)
2: heap = new min-heap()
3: for i = 1 to n − 1 do
4:     if Ω[i][2] ≥ Ω[i + 1][2] then continue
5:     heap.push(( Ω[i+1][1]−Ω[i][1] ) / ( Ω[i][2]−Ω[i+1][2] ), Ω[i], Ω[i + 1])
6: end for
7: θ = 0;
8: while heap is not empty do
9:     if O(Ω) = ⊤ then append(S, ⟨θ, 0⟩); break
10:    (θ, a, b) = heap.pop()
11:    swap a and b in Ω; add the new intersection to heap
12: end while
13: if heap is empty then return ∅ else flag = ⊤
14: while heap is not empty do
15:    (θ, a, b) = heap.pop()
16:    swap a and b in Ω; add the new intersection to heap
17:    sign = O(Ω)
18:    if flag = ⊤ and sign = ⊥ then append(S, ⟨θ, 1⟩)
19:    else if flag = ⊥ and sign = ⊤ then append(S, ⟨θ, 0⟩)
20:    flag = sign
21: end while
22: if flag = ⊤ then append(S, ⟨π/2, 1⟩)
23: return S
```

each other[1] using Equation 2, and adds them to the heap. Next, the algorithm starts sweeping toward the y-axis by removing the ordering exchange with the smallest angle from the heap. Upon visiting an ordering exchange, the algorithm swaps the items that exchange order in the ordered list Ω. The two swapped items can exchange order with their new neighbors in Ω. The algorithm updates the heap by adding these new ordering exchanges. Upon finding a satisfactory sector, the algorithm continues attaching neighboring sectors as long as they are satisfactory, to generate a satisfactory region. Algorithm 1 stores the borders of satisfactory regions in S as pairs $\langle\theta, 0/1\rangle$, where $\langle\theta, 0\rangle$ represents that θ is the start of a satisfactory region, while $\langle\theta, 1\rangle$ represents that θ is the end of the region. Consider Figure 5 and suppose that the green sectors are labeled as satisfactory by the fairness oracle. Figure 6 shows the satisfactory regions produced by Algorithm 1. Note that the third region from the left is the union of two neighboring satisfactory sectors in Figure 5.

[1] t dominates t' if $\nexists i \in [1, d]$, $t'[i] > t[i]$ and $\exists j \in [1, d]$ such that $t[j] > t'[j]$. [18]

THEOREM 1. *Algorithm 1 has time complexity* $O(n^2(\log n + \Upsilon_n))$, *where* Υ_n *is the time complexity of the fairness oracle for input of size of* n.

The proofs of all theorems are provided in Appendix F.

In our experiments we focus on fairness constraints that bound the number of items belonging to a particular type at the top-k. In general, we expect the fairness oracle to decide in a single pass over the ranking, and so Υ_n is typically $O(n)$.

3.3 Online processing

Having the sorted list of 2D satisfactory regions constructed in the offline phase allows us to design an efficient algorithm for online answering of queries. Recall that a query is a proposed set of weights for a linear ranking function. Our task is to determine whether these weights result in a fair ranking, and to suggest weight modifications if they do not. Online processing is implemented by Algorithm 2 that, given f, applies binary search on the sorted list of satisfactory regions. If f falls in a satisfactory region, the algorithm returns f, otherwise it returns the satisfactory border closest to f.

Algorithm 2 2DONLINE
Input: sorted satisfactory regions S, function $f : \langle w_1, w_2\rangle$
Output: weight vector $\langle w_1', w_2'\rangle$

```
1: (r, θ) = (√(w1^2 + w2^2), arctan w2/w1); low = 1; high = |S|
2: while (high−low) > 1 do
3:     mid = (low+high)/2
4:     if S[mid][1] < θ then low = mid
5:     else high = mid
6: end while
7: if S[low][2] = 0 then
8:     return ⟨w1, w2⟩ // input vector is satisfactory
9: end if
10: if (θ − S[low][1]) < (S[high][1] − θ) then
11:    return ⟨r cos(S[low][1]), r sin(S[low][1])⟩
12: end if
13: return ⟨r cos(S[high][1]), r sin(S[high][1])⟩
```

THEOREM 2. *Algorithm 2 has time complexity* $O(\log n)$.

4 THE MULTI-DIMENSIONAL CASE

If two attributes are used for ranking, we only have one degree of freedom — the relative weights of the two attributes — so the problem is simple. When there are three or more attributes, there are many ways in which we can perturb a given weight vector. We now extend the basic framework introduced in § 3 to handle multi-dimensional cases.

Regions of interest are no longer simple planar wedges, as in the 2D case. Rather, they are high-dimensional objects, with multiple bounding facets. To manage the geometry better, we first introduce an angle coordinate system, and show that ordering exchanges form hyperplanes in this system. Identifying and indexing satisfactory regions during offline processing is similar to constructing the *arrangement* of these hyperplanes [17]. We then propose an exact online algorithm that works based on the indexed satisfactory regions.

4.1 The geometry of ordering exchanges

Consider function f with weight vector $\vec{w} = \langle w_1, w_2, \cdots, w_d \rangle$. The score of each tuple t_i based on f is $\Sigma_{k=1}^{d} w_k t_i[k]$. For every pair of items t_i and t_j, the ordering exchange is the set of functions that give the same score to both items. As in the previous section, we consider the dual space, transforming item t into a $(d-1)$-dimensional hyperplane in $\mathbb{R}^d$:

$$\mathrm{d}(t) : \sum_{k=1}^{d} t[k].x_k = 1 \tag{3}$$

For a pair of items t_i and t_j, the intersection of $\mathrm{d}(t_i)$ and $\mathrm{d}(t_j)$ is a $(d-2)$-dimensional structure. For instance, in $\mathbb{R}^3$ the dual transformation of an item is a plane, and the intersection of two planes is a line. The intersection between $\mathrm{d}(t_i)$ and $\mathrm{d}(t_j)$ can be computed using the system of equations:

$$\times_{\mathrm{d}(t_i),\mathrm{d}(t_j)} : \begin{cases} \sum_{k=1}^{d} t_i[k].x_k = 1 \\ \sum_{k=1}^{d} t_j[k].x_k = 1 \end{cases} \tag{4}$$

The set of origin-starting rays passing through the points $p \in \times_{\mathrm{d}(t_i),\mathrm{d}(t_j)}$ represents the ordering exchange of t_i and t_j. Hence, the $(d-1)$-dimensional hyperplane defined by $\times_{\mathrm{d}(t_i),\mathrm{d}(t_j)}$ and the origin (Equation 5) contains these rays.

$$\sum_{k=1}^{d} (t_i[k] - t_j[k]) w_k = 0 \tag{5}$$

Consider items $t_1 = \{1, 2, 3\}$ and $t_2 = \{2, 4, 1\}$ in Figure 7. Using Equation 5, the ordering exchange of t_1 and t_2 is defined by the magenta plane $w_1 + 2w_2 - 2w_3 = 0$ in Figure 8.

As explained in § 2, linear functions over d attributes (rays in $\mathbb{R}^d$) are identified by $d-1$ angles, each between 0 and $\pi/2$. For instance, in § 3, we identify every function in 2D by an angle $\theta \in [0, \pi/2]$. Similarly, in multiple dimensions, we identify the functions by their angles. We now introduce the angle coordinate system for this purpose.

Angle coordinate system: Consider the $\mathbb{R}^{d-1}$ coordinate system, where every axis $\theta_i \in [0, \pi/2]$ stands for the angle θ_i in the polar representation of points in $\mathbb{R}^d$. Every function (ray in $\mathbb{R}^d$) is represented by the point $\langle \theta_1, \theta_2, \cdots, \theta_{d-1} \rangle$ in the angle coordinate system. For example, as depicted in Figure 9, a function f in $\mathbb{R}^3$ is the combination of two angles θ_1 and θ_2, each over the range $[0, \pi/2]$.

Following Equation 5, the ordering exchange of a pair of items forms a $(d-2)$-dimensional hyperplane in the angle coordinate system. For example, in 3D, the ordering exchange of t_i and t_j forms a line. We use $h_{i,j}$ to refer to the ordering exchange of t_i and t_j in the angle coordinate system. In Appendix A, we discuss how to compute ordering exchanges in the angle coordinate system.

4.2 Construction of satisfactory regions

The construction of satisfactory regions relates to the arrangement [17] of ordering exchange hyperplanes in the angle coordinate system. Consider the arrangement of $h_{i,j}$, $\forall t_i, t_j \in \mathcal{D}$. Items t_i and t_j switch order on the two sides of $h_{i,j}$, while inside each convex region in the arrangement their relative ordering does not change. In the following, we construct all convex regions in the arrangement and check if the ordering inside each is satisfactory.

A convex region is defined as the intersection of a set of half-spaces [17]. Every hyperplane h divides the space into two half-spaces h^+ and h^-. The ordering between t_i and t_j switches for each hyperplane $h_{i,j}$, moving from $h_{i,j}^+$ to $h_{i,j}^-$.

Inspired by the algorithm proposed in [17], we develop the function SATREGIONS (Algorithm 5 in Appendix A), an incremental algorithm for discovering the convex regions in the arrangement. Intuitively, the algorithm adds the hyperplanes one after the other to the arrangement. At every iteration, it finds the set of regions in the arrangement with which the new hyperplane intersects. Recall that $h_{i,j}$ is in the form of $\sum_{k=1}^{d-1} h_{i,j}[k]\theta_k = 1$. Hence, the half-space $h_{i,j}^+$ can be considered as the constraint $\sum_{k=1}^{d-1} h_{i,j}[k]\theta_k \geq 1$ and $h_{i,j}^-$ as $\sum_{k=1}^{d-1} h_{i,j}[k]\theta_k \leq 1$. The set of points inside a convex region $R = \{(h_{R1}, +/-), (h_{R2}, +/-), \cdots\}$ satisfy constraints σ_R as defined in Equation 6.

$$\sigma_R : \begin{cases} \forall \text{ half-space}(h', +) \in R,\ \sum_{k=1}^{d-1} h'[k]\theta_k \geq 1 \\ \forall \text{ half-space}(h', -) \in R,\ \sum_{k=1}^{d-1} h'[k]\theta_k \leq 1 \end{cases} \tag{6}$$

Using Equation 6, a hyperplane h intersects with a convex region R if there exists a point $p \in h$ such that the constraints in σ_R are satisfied. The existence of such a point can be determined using linear programming (LP). If the new hyperplane intersects with R, Algorithm 5 breaks it down into two convex regions that represent the intersections of R with half-spaces h^+ and h^-.

id	A_1	A_2	A_3
t_1	1	2	3
t_2	2	4	1
t_3	5.3	1	6
t_4	3	7.2	2

Figure 7: A dataset

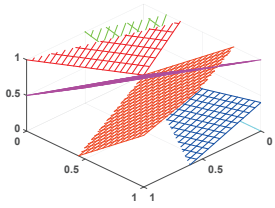


Figure 8: Ordering exchanges of Fig. 7

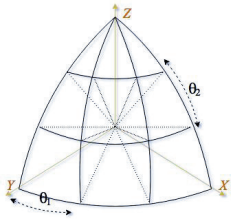


Figure 9: Angles in $\mathbb{R}^3$

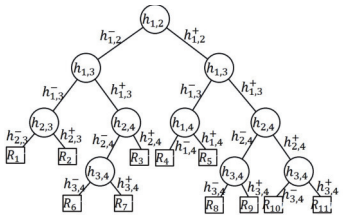


Figure 10: Arrangement tree

Having constructed the arrangement, SATREGIONS uses linear programming to find a point θ that satisfies σ_R, and uses θ to check if R is satisfactory. If R is not satisfactory, it is removed from the set of satisfactory regions $\mathcal{R}$.

THEOREM 3. *For a fixed number of dimensions, the time complexity of the function* SATREGIONS *(Algorithm 5) is* $O\big(n^{2d-1}(n\,Lp(n^2)+\Upsilon_n \log n)\big)$, *where* $Lp(n^2)$ *is the time of solving a linear programming problem of* n^2 *constrains and a fixed number of variables, and* Υ_n *is the time complexity of the fairness oracle for an input of size n.*

To add a new hyperplane, the algorithm SATREGIONS checks the intersection of every region with the hyperplane. But in practice most regions do not intersect with it. We define the *arrangement tree* (Appendix B), which keeps tracks of the space partitioning in a hierarchical manner, and can quickly rule out many regions. While this does not change the asymptotic worst case complexity, we find that it greatly helps in practice, as we will demonstrate experimentally in § 7.4.

4.3 Online processing

Thus far in this section, we studied how to preprocess the data and construct satisfactory regions $\mathcal{R}$ in multiple dimensions. Next, given a query (a function f) and $\mathcal{R}$, we aim to find the closest satisfactory function f' to f. To do so, MDBASELINE (Algorithm 6 in Appendix C) solves a non-linear programming problem for each satisfactory region to find the closest point of the region to f, and returns the function with the minimum angle distance with f.

THEOREM 4. *For a constant number of dimensions, the time complexity of Algorithm 6 is* $O(n^{2(d-1)}NLp(n^2))$, *where* $NLp(n^2)$ *is the time for solving a non-linear programming problem of* n^2 *constraints and a fixed number of variables.*

5 APPROXIMATION

A user developing a scoring function requires interactive response time from the system. MDBASELINE is not practical for query answering as it needs to solve a non-linear programming problem for each satisfactory region, before answering each query. In this section, we propose an efficient algorithm for obtaining approximate answers quickly. Our approach relies on first partitioning the angle space, based on a user-controlled parameter N, into N *cells*, where each cell c is a hypercube of $(d-1)$-dimensions. To do so, we use the equi-volume partitioning proposed in [19]. During preprocessing, we assign a satisfactory function f'_c to every cell c such that, for every function f, the angle between f and f'_c is within a bounded threshold (based on the value of N) from f and its optimal answer. To do so, we first identify the cells that intersect with a satisfactory region, and assign the corresponding satisfactory function to each such cell. Then, we assign the cells that are outside of the satisfactory regions to the nearest discovered satisfactory function.

5.1 Identifying cells in satisfactory regions

After partitioning the angle space, our objective here is to find cells in *Cells*, the set of all cells, that intersect with at least one satisfactory region $R \in \mathcal{R}$. Formally,

$$C = \{c \in Cells \mid \exists R \in \mathcal{R} \text{ s.t. } R \cap c \neq \emptyset\} \tag{7}$$

A brute force algorithm follows Equation 7 literally. This algorithm needs to first construct a complete arrangement and then check the intersection of all $N \times |\mathcal{R}|$ pairs of cells and satisfactory regions. This is inefficient when N and the size of $\mathcal{R}$ are large. As discussed in § 4, and experimentally shown in § 7, the complexity of the arrangement and the running time of the algorithm SATREGIONS highly depends on the number of hyperplanes in the arrangement. Even though the first few hyperplanes are quickly added to the arrangement, adding the later hyperplanes is more time consuming. This observation motivates us to limit the construction of the arrangement to subsets of hyperplanes, as opposed to constructing the complete arrangement all at once. Besides, the changes in the ordering in every cell is limited to the hyperplanes passing through it. As a result, for finding out if a cell intersects with a satisfactory region, it is enough to only consider the arrangement of these hyperplanes. In Appendix D, we explain how to efficiently identify the hyperplanes passing through each cell.

After identifying $\mathcal{HC}$ (the sets of hyperplanes passing through the cells), for each cell $c \in Cells$, we limit the arrangement to $\mathcal{HC}[c]$. Moreover, note that in this step our goal is to find a satisfactory function inside c. This is different from our objective in SATREGIONS , where we wanted to find *all* satisfactory regions. This gives us the opportunity

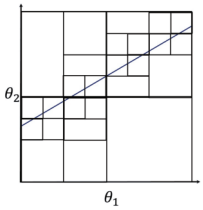


Figure 11: Cells that intersect a plane

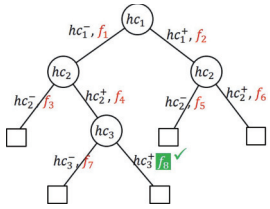


Figure 12: Early stopping in arrangement construction

to apply a *stop early* strategy, as follows: at every iteration, while using the arrangement tree for construction, check a function inside the newly added regions, and stop as soon as a satisfactory function is discovered.

We develop the algorithm MARKCELL (Algorithm 7 in Appendix D) that assigns a satisfactory function to the cells that intersect with a satisfactory region R. Using an arrangement tree, the algorithm iteratively adds the hyperplanes passing through each cell and checks if a function inside the new regions is satisfactory. Upon finding a satisfactory function, the algorithm stops and assigns the function to the cell. Figure 12 illustrates how MARKCELL finds a satisfactory function for a cell c. After adding hyperplanes hc_1 and hc_2, since functions f_1 to f_6 are unsatisfactory (denoted by red color), MARKCELL adds hc_3 to the construction. In this example, hc_3 does not pass through $\{hc_1^-, hc_2^-\}$, but it passes through $R = \{hc_1^-, hc_2^+\}$, dividing it into $R_l = R \cup hc_3^-$ and $R_r = R \cup hc_3^+$. Although $f_7 \in R_l$ is unsatisfactory, $f_8 \in R_r$ is satisfactory. The algorithm assigns f_8 to c and stops.

Let $|\mathcal{HC}[c]|$ be the number of hyperplanes passing through cell c; the complexity of the arrangement of c is $O(|\mathcal{HC}[c]|^{d-1})$. Then the complexity of MARKCELL is $O\big(|\mathcal{HC}[c]|^d Lp(|\mathcal{HC}[c]|) + |\mathcal{HC}[c]|^{d-1} n \log n \mathbb{O}_n\big)$ when d is fixed, following Theorem 3.

So far, we identified cells C that intersect with some satisfactory region, and assigned a satisfactory function to each of them. Next, we consider the cells $\bar{C}$ that do not contain a satisfactory function and assign them to the closest discovered satisfactory function. To do so, we use monotonicity of the angular distance and adopt Dijkstra's algorithm [20], see Appendix E for details. After this step, and assuming the existence of at least one satisfactory region, every cell in the partitioned angle space is assigned a satisfactory function. We store the cell coordinates, together with the assigned satisfactory functions, as an index that enables online answering of user queries, discussed next.

5.2 Online processing

Given an unsatisfactory function f, we need to find the cell to which f belongs, and to return its satisfactory function. This is implemented in MDONLINE (Algorithm 3) that transforms the weight vector of f to polar coordinates, performs binary search on each dimension to identify cell c to which f belongs, and return its satisfactory function $F[c]$.

Algorithm 3 MDONLINE

Input: partitioned space T, assigned functions F, dataset $\mathcal{D}$, fairness oracle O, and weight vector $\vec{w}$

Output: satisfactory weight vector $\vec{w}'$

```
1: if O(∇_{f_w}(D)) = True then return w
2: (r, Θ) = ToPolar(w)
3: for k = 1 to d − 1 do
4:     T = apply binary search and find the child to which Θ_k belongs
5: end for
6: return F[T]
```

Theorem 5. *Algorithm* MDONLINE *runs in* $O(\log N)$ *time.*

Theorem 6. *Let* f_{opt} *and* θ_{opt} *be the closest function and its angle distance to a queried function* f*. Also, let* f_{app} *and* θ_{app} *be the function and its angle distance that the algorithm* MDONLINE *returns for* f*, and* θ_r *the diameter the cells. Then,* $\theta_{app} \leq \theta_{opt} + 2\theta_r$*.*

6 SAMPLING

In this section, we describe two sampling-based approaches that improve the performance of our preprocessing and online processing: item sampling, and function sampling, respectively. A critical requirement of our system is to be efficient during online query processing, and it is fine for it to spend more time during offline preprocessing. As discussed in § 4 and § 5, the running time of proposed offline algorithms is polynomial for a fixed value of d. In addition, the arrangement tree (c.f. § 4) and the techniques of § 5 speed up preprocessing in practice. However, preprocessing can still be slow, particularly for a large number of items. We reduce preprocessing time using item sampling. We also present a negative result about function sampling, and show how it provides a practical method for on-the-fly query processing.

6.1 Item Sampling

The arrangement construction cost increases significantly for a large number of items in the dataset. On the other hand, in practice, fairness criteria often allow some degree of freedom in the ranking between the items. For example, a popular class of fairness models treat the top-k items in the ranking as a set, and if the proportion of protected group members in the set is within an acceptable range, they consider the over-all ranking to be fair. Having at least 20% minorities and at least 30% females in the top-20% of the ranking is an example of such a fairness model. We propose to use item sampling for these situations.

The main idea is that a uniform sample of the data maintains the properties of the underlying data distribution. Therefore, if a function is satisfactory for a dataset $\mathcal{D}$, it is *expected*

to be satisfactory for a uniformly sampled subset of $\mathcal{D}$. Hence, for a datasets with a large number of items, one can do preprocessing on a uniformly sampled subset to find functions that are expected to be satisfactory for each cell. We confirm the efficiency and effectiveness of this method experimentally on a dataset with over one million items in § 7.

6.2 Function sampling

Every origin-starting ray passes through one and only one point on the surface of the unit d-dimensional hyper-sphere (known as the d-sphere). This maps the universe of origin-starting rays (linear functions) to the surface of the first quadrant of the unit d-sphere. As a result, uniform sampling from this surface provides uniform samples from the function space. Using this observation, [21, 22] propose uniform sampling of the function space, by adopting the methods of [23, 24]. The idea is that, since the Normal distribution has a constant probability on the surfaces of d-spheres with common centers, taking samples based on this distribution from the weight space provides uniform samples from the function space. This idea is extended in [22] for taking unbiased samples in the θ-vicinity of a specific ray. We can use this for *on-the-fly query processing*.

In this paper, we conduct preprocessing that enables an efficient way of answering user queries. The alternative scenario is on-the-fly processing of the queries, without any preprocessing. Consider the case where the objective is to find a satisfactory function that has at most the angle distance of θ with the user-provided function, called *the region of interest* in [22]. Constructing the arrangement at query time is intractable. Instead, one can use uniform random samples for exploring the neighborhood. To do so, we take a uniform random function sample in the θ-vicinity of the input function and return it if satisfactory. Otherwise, we take another sample, and continue until a budget of s samples is exhausted. Unfortunately, the negative result is that, based on Theorem 7, function sampling cannot provide any guarantees for the discovery of an approximation solution.

Theorem 7. *For any arbitrarily small probability $p > 0$ and any arbitrarily large number s, one cannot guarantee the discovery of a satisfactory function with probability at least p, using s uniform random function samples.*

Although function sampling is efficient in drawing a function from large satisfactory regions, one cannot guarantee that all possible rankings will be discovered and, therefore, cannot ensure the *non-existence* of a satisfactory function.

On the other hand, as studied in [22], the rankings supported by small regions are unstable, and thus questionable. Following the function sampling strategy for on-the-fly query processing, we expect to find a satisfactory function, if the volume ratio of the satisfactory regions to the volume of the region of interest is more than $1/s$. Taking more samples increases the chance of hitting smaller satisfactory regions, but reduces efficiency. Given that we stop exploring as soon as a satisfactory function is discovered, as we shall show in § 7, this method is efficient in practice, for the cases in which it finds a satisfactory function. However, in other cases, it may fail to find a satisfactory function although one exists.

7 EXPERIMENTAL EVALUATION

7.1 Experimental Setup

Datasets. *COMPAS*: a dataset collected and published by ProPublica as part of their investigation into racial bias in criminal risk assessment software [25]. The dataset contains demographics, recidivism scores produced by the COMPAS software, and criminal offense information for 6,889 individuals. We used `c_days_from_compas`, `juv_other_count`, `days_b_screening_arrest`, `start`, `end`, `age`, and `priors_count` as scoring attributes. We normalized attribute values as $(val - min)/(max - min)$. For all attributes except `age`, a higher value corresponded to a higher score. In addition to the scoring attributes, we consider attributes `sex` (0:male, 1: female), `age_binary` (0: less than 35 yo, 1: more than 36 yo), `race` (0: African American, 1: Caucasian, 2: Other), and `age_bucketized` (0: less than 30 yo, 1: 31 to 40 yo, 2: more than 40 yo), as the type attributes. COMPAS is our default experimental dataset.

US Department of Transportation (DOT): the flight on-time database published by DOT is widely used by third-party websites [26]. We collected 1.3M records, for the flights conducted by 14 US carriers in the first three months of 2016. We use this to study sampling for large-scale settings, and to showcase the application of our techniques for diversity.

Hardware and platform. The experiments were performed on a Linux machine with a 2.6 GHz Core I7 CPU and 8GB memory. The algorithms were implemented using Python2.7, with scipy.optimize package for LP optimizations.

Fairness models. We evaluate the performance of our methods over two general fairness models, see § 2.

FM1, proportional representation on a single type attribute, is the default fairness model in our experiments. This model can express common proportionality constraints from the literature [3, 7, 8], including also for ranked outputs [6] and for set selection [4]. The distinguishing features of FM1 are (1) that the type attribute partitions the input dataset $\mathcal{D}$ into groups and (2) that the proportion of members of a particular group is bounded from below, from above, or both. For the COMPAS dataset, unless noted otherwise, we state FM1 over the type attribute `race` as follows: African Americans constitute about 50% of the dataset; a fairness oracle will consider a ranking to be satisfactory if at most 60% (or about 10% more than in $\mathcal{D}$) of the top-ranked 30% are African American.

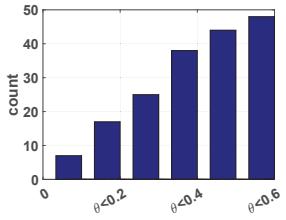


Figure 13: Angle between input/output functions

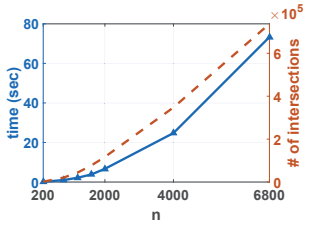


Figure 14: 2D; preprocessing time, varying n

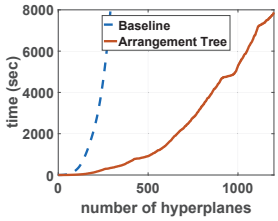


Figure 15: The advantage of arrangement tree

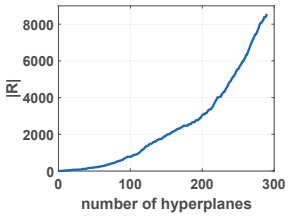


Figure 16: Incremental arrangement cost ($d = 3$)

FM2, proportional representation on multiple, possibly overlapping, type attributes, is a generalization of FM1 that can express proportionality constraints of [10]. As in [10], we bound the number of members of a group from above. For example, for COMPAS, we specify the maximum number of items among the top-ranked 30% based on sex (80% of $\mathcal{D}$ are male), race (50% are African American), and age_bucketized (42% are 30 years old or younger, 34% are between 31 and 50, and 24% are over 50). A ranking is considered satisfactory if the proportion of members of a particular demographic group is no more than 10% higher than its proportion in $\mathcal{D}$.

7.2 Validation experiments

In our first experiment, we show that our methods are effective — that they can identify scoring functions that are both satisfactory and similar to the user's query. We use COMPAS with $d = 3$ (scoring attributes c_days_from_compas, juv_other_count, start), and with fairness model FM1 on race (at most 60% African Americans among the top 30%).

We issued 100 random queries, and observed that 52 of them were satisfactory, and so no further intervention was needed. For the remaining 48 functions, we used our methods to suggest the nearest satisfactory function. Figure 13 presents a cumulative plot of the results for these 48 cases, showing the angle distance $\theta(f, f')$ between the input f and the output f' on the x-axis, and the number of queries with *at most* that distance on the y-axis.

We observe that a satisfactory function f' was found close to the input function f in all cases. Specifically, note that $\theta(f, f') < 0.6$ in all cases, and recall that $\theta \in [0, \pi/2]$, with lower values corresponding to higher similarity. (For a more intuitive measure: the value of $\theta = 0.6$ corresponds to cosine similarity of 0.82, where 1 is best, and 0 is worst). Among the 48 cases, 38 had $\theta(f, f') < 0.4$ (cosine similarity 0.92).

Next, we give an intuitive understanding of the layout of satisfactory regions in the space of ranking functions. We use COMPAS with age (lower is better) and juv_other_count (higher is better) for scoring. The intuition behind this scoring function is that individuals who are younger, and who have a higher number of juvenile offenses, are considered to be more likely to re-offend, and so may be given higher priority for particular supportive services or interventions.

Naturally, a scoring function that associates a high weight with age will include mostly members of the younger age group at top ranks. About 60% of COMPAS are 35 years old or younger. Consider a fairness oracle that uses FM1 over age_binary (with groups g_1: 35 year old or younger, and g_2: over 35 years old), and that considers a ranking satisfactory if at most 70% of the top-100 results are in g_1. Because of the correlation (by design) between one of the scoring attributes and the type attribute, there is only one satisfactory region for this problem set-up — it corresponds to the set of functions in which the weight on age is close to 0, and whose angle with the x-axis (juv_other_count) is at most 0.31.

Next, suppose that we use the same scoring attributes, but a different fairness oracle — one that applies FM1 on the attribute race, requiring that at most 60 of the top-100 are African American. This time, there exist several satisfactory regions. For any assignment of weights to the two scoring attributes, there exists a satisfactory function f' such that $\theta(f, f') < 0.11$ (cosine similarity is more than 0.99).

Finally, we use juv_other_count and c_days_from_compas for scoring, with fairness model FM2 that considers a ranking satisfactory if there are at most 90 males, at most 60 African Americans, and at most 52 persons who are 30 years old or younger at the top-100. This fairness model is stricter than in the preceding experiment (with FM1 on race), making the gaps between the satisfactory regions wider. Still, the maximum angle between f and f' was less than 0.28, which corresponds to the minimum cosine similarity of 0.96.

7.3 Performance of query answering

While preprocessing can take more time, a critical requirement of our system is to be fast when answering users' queries. In this section, we use the COMPAS dataset and evaluate the performance of 2DONLINE and MDONLINE , the 2D and MD algorithms for online query answering. We show that queries can be answered in interactive time. We use the default fairness model (i.e., at most 60% African Americans in the top-30%) and the scoring attributes in the same ordering provided in the description of COMPAS dataset.

2D. 2DONLINE does not need to access the raw data at query time. It only needs to apply binary search on the sorted

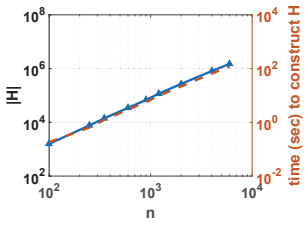


Figure 17: MD; effect of n on $|H|$ ($d = 3$)

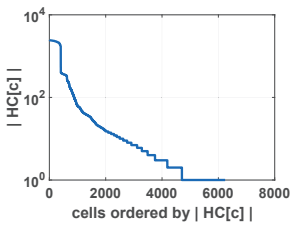


Figure 18: MD; # of hyperplanes intersecting a cell

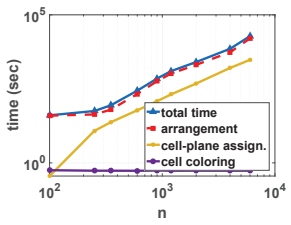


Figure 19: MD; effect of n on preprocessing time

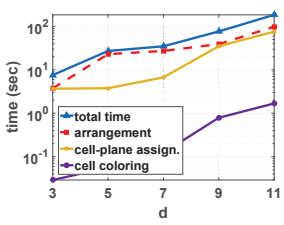


Figure 20: MD; effect of d on preprocessing time

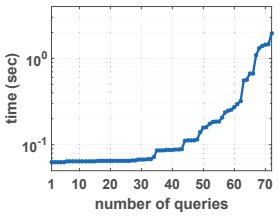


Figure 21: On-the-fly processing in $\pi/20$-vicinity

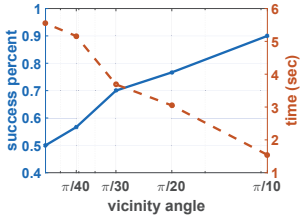


Figure 22: On-the-fly processing; varying vicinity

list of satisfactory ranges to locate the position of the input function f. In this experiment, we compare the required time for ordering the results based on the input function, averaged over 30 runs of 2DONLINE on random inputs. Confirming the theoretical $O(\log n)$ complexity of 2DONLINE v.s. the $O(n \log n)$ for the ordering, 2DONLINE only required *30 μsec* on average, while even ordering the results based on f (to check if f is satisfactory) required 25 msec to complete.

MD. In this experiment, just as for 2D, we took the average running time of 30 random queries. Upon arrival of a query function f, MDONLINE finds the cell to which f belongs in $O(\log N)$, where N is the number of cells, and returns the proper satisfactory function f'. In all experiments the running time was less than *200 μsec* whereas the time required to order the items was *25 msec*.

7.4 Performance of preprocessing

To study preprocessing performance, we again use the COMPAS dataset, with the default fairness model (at most 60% African Americans at the top 30%).

2D. We start by evaluating the efficiency of 2DRAYSWEEP, the 2D preprocessing algorithm proposed in § 3. We study the effect of n (the number of items in the dataset) on the performance of the algorithm 2DRAYSWEEP and evaluate the number of ordering exchanges and the running time of it. Figure 14 shows the experiment results for varying the number of items from 200 to 6,800. The x-axis shows the values of n (in log-scale), and the left and right y-axes show the running time of 2DRAYSWEEP and the number of ordering exchanges, respectively. Looking at the right y-axis, one can observe that the number of ordering exchanges is much smaller than the theoretical $O(n^2)$ upper-bound. For example, while the upper-bound on the number of ordering exchanges for n = 4k is 16M, the observed number in this experiment was around 400k. This is because the pairs of items in which one dominates the other do not have an ordering exchange. Also, looking at the left y-axis, one can confirm that the algorithm managed to finish the preprocessing within the reasonable time of 80 sec, even for the largest setting.

MD, the effect of using arrangement tree. In § 4, we proposed the arrangement tree data structure for constructing the arrangement of hyperplanes, in order to skip comparing a new hyperplane with all current regions. Here, as the first MD experiment, we run the algorithm SATREGIONS as the baseline and also use AT_+ for adding the hyperplanes using the arrangement tree. Figure 15 shows the incremental cost of adding hyperplanes to the arrangement when $d = 3$. While the baseline (SATREGIONS) needed 8,000 sec for adding the first 250 hyperplanes, using the arrangement tree helped save around 7,740 secs. Fixing the budget to 8,000 sec, the baseline could construct the arrangement for the first 250 hyperplanes, while using the arrangement tree allowed us to extend the construction to 1,200 hyperplanes.

Recall from §4 that the number of regions at step i is $O(i^{2(d-1)})$, and hence, adding the subsequent hyperplanes is more expensive. This is presented in Figure 16, where the y-axis shows the number of regions in the arrangement ($|\mathcal{R}|$) for different number of hyperplanes. Observe that the number of regions for the first 50 hyperplanes is less than 200; it increases to more than 5,000 regions for the hyperplanes that are added after 250^{th} iteration. As a result, while adding a hyperplane (without using the arrangement tree) at the first 50 iterations requires checking fewer than 200 regions, adding a hyperplane after iteration 250 requires checking more than 5,000 regions, and so is significantly more expensive.

MD, preprocessing. We now evaluate the algorithms proposed in § 5 for preprocessing. Recall that in § 5, we partition the angle space into N cells and assign a satisfactory function to each cell. This enables interactive query processing, since, for each query f, we simply return the satisfactory function of the cell to which f belongs. Theorem 6 provides an

upper-bound on the quality of the approximation introduced by the algorithm. Following the upper-bound for $d = 3$ and $N = 40$k, for example, the maximum distance of the approximate output to the input function is about 0.004 degrees more than the optimal distance. We could not evaluate an average approximation, as the exact baseline solution did not finish for any of the settings.

First, similar to the 2D experiments, varying n from 200 to 6,000, in Figure 17 we observe $|H|$ (the number of hyperplanes) as well as the time for constructing the hyperplanes in the angle coordinate system. Comparing this figure with Figure 14 (remember that intersections in 2D and hyperplanes in MD refer to the ordering exchanges), we observe that $|H|$ gets closer to n^2 as the number of dimensions increase. This is because, as the number of dimensions increases, the probability that one in a pair of items dominate the other decreases, and therefore $|H|$ gets closer to n^2. Also, looking at the right y-axis and the dashed orange line, and comparing it with $|H|$ (the left y-axis) confirms that the total running time is linear to the number of hyperplanes.

In the previous experiment, we observed the major effect of the number of hyperplanes on the construction time. Thus, in § 5, we limit the arrangement construction for each cell to the hyperplanes passing through it. In Figure 18, setting $n = 100$ and $d = 4$, we evaluated the number of hyperplanes passing through the cells. The x-axis in Figure 18 is the cells sorted by $|\mathcal{HC}[c]|$ (the number of hyperplanes passing through a cell c), and the y-axis shows $|\mathcal{HC}[c]|$. More than 80% of the cells have fewer than 100 hyperplanes passing through them, and even the complete arrangement construction inside them is reasonable. Also, recall from § 5 that MARKCELL stops arrangement construction as soon as a satisfactory function is identified.

Figures 19 and 20 show the required time for different steps of preprocessing, as well as the total preprocessing time. Figure 19 shows the cost for varying n, with $d = 3$ and $N = 40,000$. We note that in practice, humans tend to define rankings over a limited number of attributes on account of the cognitive burden involved. Even then, coming up with a weight vector for a limited set of attributes is challenging. Still, there are situations with more attributes, and it makes sense to study the performance of our proposals for these cases. Hence, in Figure 20 we vary d from 3 to 11.

Since the number of attributes in COMPAS and DOT is limited, we executed the experiment in Figure 20 on synthetic data, generated using the Zipfian distribution. We normalized attribute values in the range $[0, 1]$ and added a binary type attribute with values values assigned uniformly at random. Similarly to our default fairness oracle $FM1$, we consider a ranking to be fair if at most 60% of its top-30% are of type 1.

The yellow line in both figures shows the required time for identifying the hyperplanes passing through each cell. Applying CELLPLANE$_\times$ for finding the cells for each hyperplane helps skip a large portion of the cells. Still its running time increases significantly as n increases. This is because the number of ordering exchanges $|H|$ is in $O(n^2)$. Similarly, as the number of attributes increases, the chance that items dominate each other decreases and hence $|H|$ increases. Also, for a fixed N, the hyperplanes intersect with more cells. As a result, the time taken by CELLPLANE$_\times$ increases by the number of attributes. The red dashed line shows the arrangement construction cost. As expected, in most settings the majority of the time is taken by this step. Still, the optimizations proposed in § 4 and 5 result in reasonable performance of this step. First, limiting the construction of the arrangement for each cell c, to the hyperplanes passing through it, reduces the complexity of the arrangement to $|\mathcal{HC}[c]|^{d-1}$. Second, as shown in Figure 15, the arrangement tree data structure helps rule out checking the intersection of the hyperplanes with all regions. Finally, the early stop condition is effective at reducing the running time. The final step is to assign the function of the closest satisfactory cell to each unsatisfactory cell. This step uses a priority queue, and is observed in all the settings in Figures 19 and 20 to be be fast, as expected.

7.5 Performance of sampling

Item sampling for a large-scale setting. As explained in § 6.1, item sampling can be used for reducing preprocessing time for large datasets. In this experiment, we use the DOT dataset, with three scoring attributes, `departure delay`, `arrival delay`, and `taxi in`. The fairness oracle uses FM1 with airline name as the type attribute. A ranking is satisfactory if the percentage of outcomes from each of four major companies Delta Airlines (DL), American Airlines (AA), Southwest (WN), and United Airlines (UA) in the top 10% is at most 5% higher than their proportion in the dataset. We sample 1K records uniformly at random from the dataset of 1.3M records and use it for preprocessing with $N = 40$K. Preprocessing took 20 min. Next, we used the complete dataset and checked if the function assigned to the cells using the sample are in fact satisfactory. For all assigned functions the percentage of results from each of four major airlines in the top 10% was at most 5% higher than their proportion in the whole dataset — all of them were satisfactory.

Function sampling for on-the-fly query processing. In § 6.2, we discussed function sampling for our problem. Unfortunately, as stated in Theorem 7, function sampling cannot provide a guarantee on finding an approximation solution for the problem. Still, we suggest it for on-the-fly query processing in order to explore the θ-vicinity of the input function.

We choose the COMPAS dataset and set $n = 6,000$ and $d = 3$. First, we generate 100 random queries and use a budget of 500 random functions to explore the $(\pi/20)$-vicinity of

the input functions. For 27 of those, the algorithm exhausted its budget but could not find a satisfactory function, taking about 11 sec. We plotted the running time of the algorithm for the other cases in Figure 21. Around 60% of the successful queries finished in less than 0.1 sec. Those are the ones that are either satisfactory, or are in mostly satisfactory regions. There are a few cases in which the exploration was successful but it took a few seconds to find a satisfactory function. In summary, in most cases the algorithm either quickly finds a satisfactory function or it never finds one.

Next, we evaluate the effect of the width of θ-vicinity. To do so, we draw 30 random functions, vary θ from $\pi/50$ to $\pi/10$, and use a budget of 500 function samples for exploration for each input function. The results are provided in Figure 22. The left y-axis shows the percentage of queries in which the algorithm could find a satisfactory result. In general, the wider the exploration space, the higher the chance of finding a satisfactory result. This is reflected in the figure, as the success rate increased from around 50% at $\theta = \pi/50$ to around 90% at $\theta = \pi/10$. The right y-axis of the plot shows the average running time of the algorithm for each setting. The running time drops from around 5.5 sec at $\theta = \pi/50$ to less than 2 sec at $\theta = \pi/10$. The reason is that for wider vicinities, the algorithm has a higher chance of finding a satisfactory function and stopping early.

8 RELATED WORK

There is a robust body of prior work on computing preferences over datasets, divided into two major categories: (1) ranking and top-k [27] query processing for cases where the user has a scoring function in mind, and (2) finding representatives such as skyline [18, 28, 29], and its subsets such as regret minimizing sets [16, 21, 30], in the absence of a scoring function. The primary focus of this work is on efficient query processing, for which methods include threshold-based [31], view-based [32], and indexing-based [33]. Subsequent work considered efficient query processing for computing preferences over spatial [34] and noisy [35] databases. None of these consider adjusting the scoring function.

Our work is among a small handful of studies that focus on fairness in ranking [5, 6, 10]. Several recent papers focus on measuring fairness in ranked lists [5, 6], on constructing ranked lists that meet fairness criteria [10], and on fair and diverse set selection [4]. Fairness in top-k over a single binary type attribute (such as gender, ethnicity, or disability status) is studied in Zehlike et al. [6], where the goal is to ensure that the proportion of members of a protected group in every prefix of the ranking remains statistically above a given minimum. Celis et al. [10] provide a theoretical investigation of ranking with fairness constraints. In their work, fairness in a ranked list is quantified as an upper bound or a lower bound on the number of items at the top-k that belong to multiple, possibly overlapping, types. In contrast, our goal is to assist in designing fair rankers by adjusting scoring function weights.

Diversification of query results has always been important in data retrieval [36–38]. Different definitions of diversity include similarity function-based [39] and topic-based [38]. General background on diversity and a connection to fairness are provided in [36]. An advantage of the techniques proposed in our paper is that they are independent of the choice of a fairness function. In fact, one can replace the fairness oracle with any binary-output function that takes an ordering of the items as the input. This makes our techniques suitable for a general range of diversity definitions.

The techniques of this paper mainly follow the concepts in combinatorial geometry. The general background and the terms are provided in [17, 40]. In addition, [17] discussed the complexity bounds and proposes the incremental algorithm for constructing the lattice of arrangement. Arrangement of hyperplanes is also studied in [41–43]. Applications of arrangements such as motion planner in robotics are discussed by Agarwal et. al. [44].

Finally, our work in this paper is synergistic with our recent work on obtaining stable rankings [22], where we also build on combinatorial geometry techniques, and are motivated by supporting responsible decision making practices in ranked contexts.

9 FINAL REMARKS

In this paper, we studied the problem of designing fair ranking schemes. Our system computes the score of each item as a weighted linear combination of its attribute values, and assists users in choosing attribute weights that lead to fair ranked outcomes. Creating proper indexes in an offline manner enables interactive response to user queries. We carried out theoretical analysis and empirical experiments, and confirmed that our methods are both efficient and effective.

We designed techniques for a general fairness model that takes an ordering of the items as input and decides whether it meets the fairness requirements. Additional information about the fairness model may help optimize the techniques, which we will explore in our future work.

10 ACKNOWLEDGEMENTS

The work of Abolfazl Asudeh and H. V. Jagadish was supported in part by NSF grants No. 1741022 and 1250880. The work of Julia Stoyanovich was supported in part by NSF grant No. 1741047. The work of Gautam Das was supported in part by grant W911NF-15-1-0020 from the Army Research Office, NSF grant No. 1745925, and a grant from AT&T.

REFERENCES

[1] Solon Barocas and Andrew D. Selbst. Big data's disparate impact. *California Law Review*, 104, 2016.
[2] Batya Friedman and Helen Nissenbaum. Bias in computer systems. *ACM Trans. Inf. Syst.*, 14(3):330–347, 1996.
[3] Michael Feldman, Sorelle A. Friedler, John Moeller, Carlos Scheidegger, and Suresh Venkatasubramanian. Certifying and removing disparate impact. In *SIGKDD*, 2015.
[4] Julia Stoyanovich, Ke Yang, and H.V. Jagadish. Online set selection with fairness and diversity constraints. In *EDBT*, 2018.
[5] Ke Yang and Julia Stoyanovich. Measuring fairness in ranked outputs. In *SSDBM*, 2017.
[6] Meike Zehlike, Francesco Bonchi, Carlos Castillo, Sara Hajian, Mohamed Megahed, and Ricardo A. Baeza-Yates. FA*IR: A fair top-k ranking algorithm. In *CIKM*, 2017.
[7] Indre Zliobaite. Measuring discrimination in algorithmic decision making. *Data Min. Knowl. Discov.*, 31(4):1060–1089, 2017.
[8] Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard S. Zemel. Fairness through awareness. In *ITCS*, 2012.
[9] Sorelle A. Friedler, Carlos Scheidegger, and Suresh Venkatasubramanian. On the (im)possibility of fairness. *CoRR*, abs/1609.07236, 2016.
[10] L. Elisa Celis, Damian Straszak, and Nisheeth K. Vishnoi. Ranking with fairness constraints. In *ICALP*, 2018.
[11] The College Board. SAT percentile ranks, 2014.
[12] Peter Jacobs. Legacy admissions policies were originally created to keep jewish students out of elite colleges. *Business Insider*, 2013.
[13] Jerome Karabel. *The Chosen: The Hidden History of Admission and Exclusion at Harvard, Yale, and Princeton.* HMHCO, 2005.
[14] The New York City Council. Int. No. 1696-A: A Local Law in relation to automated decision systems used by agencies. https://laws.council.nyc.gov/legislation/int-1696-2017/, 2017. [Online; accessed on 28-September-2018].
[15] Abolfazl Asudeh, Nan Zhang, and Gautam Das. Query reranking as a service. *PVLDB*, 9(11):888–899, 2016.
[16] Abolfazl Asudeh, Azade Nazi, Nan Zhang, and Gautam Das. Efficient computation of regret-ratio minimizing set: A compact maxima representative. In *SIGMOD*, 2017.
[17] Herbert Edelsbrunner. *Algorithms in combinatorial geometry*, volume 10. Springer Science & Business Media, 2012.
[18] Abolfazl Asudeh, Saravanan Thirumuruganathan, Nan Zhang, and Gautam Das. Discovering the skyline of web databases. *VLDB*, 2016.
[19] Akrivi Vlachou, Christos Doulkeridis, and Yannis Kotidis. Angle-based space partitioning for efficient parallel skyline computation. In *SIGMOD*. ACM, 2008.
[20] M L Fredman and R E Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *JACM*, 34(3), 1987.
[21] Abolfazl Asudeh, Azade Nazi, Nan Zhang, Gautam Das, and H.V. Jagadish. Rrr: Rank-regret representative. In *SIGMOD*, 2019.
[22] Abolfazl Asudeh, H.V. Jagadish, Gerome Miklau, and Julia Stoyanovich. On obtaining stable rankings. *PVDLB*, 12(3):237–250, 2018.
[23] Mervin E Muller. A note on a method for generating points uniformly on n-dimensional spheres. *Communications of the ACM*, 2(4), 1959.
[24] George Marsaglia et al. Choosing a point from the surface of a sphere. *The Annals of Mathematical Statistics*, 43(2), 1972.
[25] Julia Angwin, Jeff Larson, Surya Mattu, and Lauren Kirchner. Machine bias: Risk assessments in criminal sentencing. *ProPublica*, 5/23/2016.
[26] United States Department of Transportation. Bureau of transportation statistics. https://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time. [Online; accessed 12/29/2017].
[27] I Ilyas, G Beskales, and M Soliman. A survey of top-k query processing techniques in relational database systems. *CSUR*, 40(4), 2008.
[28] F Rahman, A Asudeh, N Koudas, and G Das. Efficient computation of subspace skyline over categorical domains. In *CIKM*. ACM, 2017.
[29] S Borzsony, Donald Kossmann, and Konrad Stocker. The skyline operator. In *ICDE*, 2001.
[30] Danupon Nanongkai, Atish Das Sarma, Ashwin Lall, Richard J Lipton, and Jun Xu. Regret-minimizing representative databases. *VLDB*, 2010.
[31] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *JCSS*, 66(4), 2003.
[32] V. Hristidis and Y. Papakonstantinou. Algorithms and applications for answering ranked queries using ranked views. *VLDBJ*, 2004.
[33] Y. Chang, L. Bergman, V. Castelli, C. Li, M. Lo, and J. Smith. The onion technique: indexing for linear optimization queries. In *SIGMOD*, 2000.
[34] Gisli R Hjaltason and Hanan Samet. Ranking in spatial databases. In *SSTD*. Springer, 1995.
[35] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic ranking of database query results. In *PVLDB*, 2004.
[36] Marina Drosou, HV Jagadish, Evaggelia Pitoura, and Julia Stoyanovich. Diversity in Big Data: A review. *Big Data*, 5(2), 2017.
[37] Bert Boyce. Beyond topicality: A two stage view of relevance and the retrieval process. *Inf Process Manag*, 18(3):105–109, 1982.
[38] Rakesh Agrawal, Sreenivas Gollapudi, Alan Halverson, and Samuel Ieong. Diversifying search results. In *WSDM*, pages 5–14. ACM, 2009.
[39] Harr Chen and David R Karger. Less is more: probabilistic models for retrieving fewer relevant documents. In *SIGIR*. ACM, 2006.
[40] Mark De Berg, Otfried Cheong, Marc Van Kreveld, and Mark Overmars. *Computational Geometry: Introduction.* Springer, 2008.
[41] Peter Orlik and Hiroaki Terao. *Arrangements of hyperplanes*, volume 300. Springer Science & Business Media, 2013.
[42] Branko Grünbaum. Arrangements of hyperplanes. In *Convex Polytopes*, pages 432–454. Springer, 2003.
[43] Vadim V Schechtman and Alexander N Varchenko. Arrangements of hyperplanes and lie algebra homology. *Inventiones math.*, 1991.
[44] Pankaj K Agarwal and Micha Sharir. Arrangements and their applications. *Handbook of computational geometry*, pages 49–119, 2000.
[45] Raphael A Finkel and Jon Louis Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

A ORDERING EXCHANGES IN MD

Before one can construct satisfactory regions, we first need to compute ordering exchanges in the angle coordinate system. Algorithm 4 computes $h_{i,j}$ for a given pair of items t_i and t_j. The algorithm uses $(d-1)$ linearly independent points in the hyperplane of Equation 5, and finds the angles of the ray from the origin through each of the points, using their polar representations. To find the points, one can start with an arbitrary non-zero point on the plane and scale each dimension independently to get the other points. After this step, each row of the $(d-1) \times (d-1)$ matrix Θ shows a point in the angle coordinate system. HYPERPOLAR represents hyperplanes as $\sum_{k=1}^{d-1} h_{i,j}[k]\theta_k = 1$. Since all $(d-1)$ points in Θ fall in $h_{i,j}$, this forms a linear system of equations $\Theta \times h_{i,j} = \iota$, where ι is the unit vector of size $(d-1)$. Solving this system of equations, we get $h_{i,j} = \Theta^{-1} \times \iota$. Given that computing Θ^{-1} is the bottleneck in Algorithm 4, it is easy to see that HYPERPOLAR is in $O(d^3)$, which is $O(1)$ for a fixed d.

Algorithm 4 HYPERPOLAR
Input: items t_i and t_j
Output: ordering exchange $h_{i,j}$

$V = [t_i[k] - t_j[k]\,,\ \forall 1 \le k \le d]$ // Equation 5
$\Theta = \{\}$
$p = d - 1$ linearly independent points satisfying Equation 5
for $k = 1$ to $d - 1$ **do**
  (r, θ) = **ToPolar**($p[k]$)
  add θ to Θ
end for
// Find the hyperplane containing the points in Θ
$\iota = [1, 1, \cdots, 1]$
return $\Theta^{-1} \times \iota$

Algorithm 5 SATREGIONS
Input: dataset $\mathcal{D}$ and fairness oracle O
Output: satisfactory regions $\mathcal{R}$

$H = \{\}$
for $i = 1$ to $n - 1$ **do**
  for $j = i + 1$ to n **do**
    if t_i or t_j dominates the other **then continue**
    add HYPERPOLAR (t_i, t_j) to H
  end for
end for
$\mathcal{R} = \{\ \{(H[1], +)\}, \{(H[1], -)\}\ \}$
// add hyperplanes incrementally to the arrangement
for $h \in (H \backslash \{H[1]\})$ **do**
  $\ell_{\mathcal{R}} = |\mathcal{R}|$
  for $i = 1$ to $\ell_{\mathcal{R}}$ **do**
    if $\exists p \in h$ s.t. $\sigma_{\mathcal{R}[i]}$ **then**
      $R' = \mathcal{R}[i]$
      append $\mathcal{R}[i]$ by $(h, +)$; append R' by $(h, -)$
      add R' to $\mathcal{R}$
    end if
  end for
end for
for $R \in \mathcal{R}$ **do**
  θ = a point that σ_R is satisfied; $\vec{w}$ = **ToCartesian**$(1, \theta)$
  if $O(\nabla_{f_{\vec{w}}}(\mathcal{D}))$ = False **then** remove R from $\mathcal{R}$
end for
return $\mathcal{R}$

B ARRANGEMENT TREE

Consider a binary tree where every vertex v is associated with a hyperplane h_i, while its left and right edges refer to h_i^- and h_i^+, respectively. Every vertex of the tree corresponds to a region R that is the set of half-spaces specified by the edges from the root to it. As a result, the left (resp. right) child of v shows the regions in R that fall in h^- (resp. h^+). Figure 10 shows a sample arrangement tree for a set of 6 hyperplanes $\{h_{1,2}, h_{1,3}, h_{1,4}, h_{2,3}, h_{2,4}, h_{3,4}\}$. The leaves of the tree are the regions of the arrangement. The region R_3, for example, is the intersection of the half-spaces $\{h_{1,2}^-, h_{1,3}^+, h_{2,4}^+\}$. In this figure, consider the left child of the root. Let us assume that a new hyperplane h does not intersect with the right child of this node, i.e., it does not intersect with the region $\{h_{1,2}^-, h_{1,3}^+\}$. Then we can prune the whole subtree and skip checking the intersection of h with the regions R_3, R_6, and R_7, because all these regions are inside the region $\{h_{1,2}^-, h_{1,3}^+\}$.

C ONLINE PROCESSING BASELINE

We now present a baseline algorithm for identifying satisfactory regions in the multidimensional case.

Algorithm 6 MDBASELINE
Input: Satisfactory regions $\mathcal{R}$, dataset $\mathcal{D}$, fairness oracle O, function $f : \vec{w}$
Output: the satisfactory weight vector $\vec{w}'$

if $O(\nabla_{f_{\vec{w}}}(\mathcal{D}))$ = True **then return** $\vec{w}$
$(r, \Theta^{(i)})$ = **ToPolar**$(\vec{w})$
mindist=∞
for $R \in \mathcal{R}$ **do**
  (dist,$\Theta^{(j)}$) = the minimum $\theta_{i,j}$ such that C_R is satisfied
  if dist<mindist **then** $\Theta^o = \Theta^{(j)}$, mindist=dist
end for
return ToCartesian$(1, \Theta^o)$

D IDENTIFYING THE HYPERPLANES PASSING THROUGH EACH CELL

Given a hyperplane h and a cell c, checking if h passes through c is simple, using the "bottom-left" (bl) and "top-right" (tr) corners of the cell, i.e., the corners that have the minimum and maximum values of the cell ranges in each dimensions. Recall that HYPERPOLAR constructs the hyperplane h in the form of $\sum_{k=1}^{d-1} h[k]\theta_k = 1$. Thus, for every point p in h^-, $\sum_{k=1}^{d-1} p_k\theta_k \le 1$ while for every point p' in h^+, $\sum_{k=1}^{d-1} p'_k\theta_k \ge 1$.Therefore, h passes through c, *iff* $\sum_{k=1}^{d-1} bl[k]\theta_k \le 1$ and $\sum_{k=1}^{d-1} tr[k]\theta_k \ge 1$.

The complete pairwise check between each hyperplane and each cell takes $O(N \times |H|)$ time. Instead, we use the following observation to skip some of the operations: consider a hyperrectangle specified by its bottom-left corner bl and the top-right corner tr; also consider a hyperplane h that does not pass through this hyperrectangle. For every cell c for which its bottom-left dominates bl (for each dimension i its value is greater than or equal to $bl[i]$) and its top-right corner is dominated by tr, h does not pass through c.

As a result, for checking the cells that intersect with hyperplane h, one can start from the complete angle space, partition the space in a hierarchical manner, and prune the cells inside the hyperrectangles that do not intersect with h. We adopt the *quadtree* [45] data structure for this purpose. To do so, we develop a recursive algorithm (CELLPLANE$_\times$) that iterates over the dimensions in a round robin manner and, at every step, if h passes through the current hyperrectangle, divides it in two equi-size hyperrectangles on the current dimension. Figure 11 illustrates CELLPLANE$_\times$ for finding the cells that intersect with the drawn line h. The algorithm

prunes all cells in the bottom-right quadrant, since h does not pass through it.

Algorithm 7 MARKCELL
Input: cell c, $\mathcal{HC}$

```
1: if |HC[c]| = 0 then
2:     p = a point inside c
3:     if O(∇_fp(D)) = True then Marked[c]= p
4:     return
5: end if
6: p = a point in HC[c][1]^- ∩ c
7: if O(∇_fp(D)) = True then Marked[c]= p; return
8: p = a point in HC[c][1]^+ ∩ c
9: if O(∇_fp(D)) = True then Marked[c]= p; return
10: T =new ArrangementTree(HC[c][1])
11: for h ∈ HC[c]\HC[c][1] do
12:     if p =ATC+ (T,h,c,null) is not null then
13:         Marked[c]= p; return
14:     end if
15: end for
```

Algorithm 8 ATC+
Input: arrangement tree T, hyperplane h, cell c, constraints path to root σ

```
1: if T is null then
2:     T = new ArrangementTree(h)
3:     σ_l = σ ∪ {Σ_{k=1}^{d-1} h[k]θ_k ≤ 1}
4:     p = a point in c s.t. σ_l is satisfied
5:     if O(∇_fp(D)) = True then return p
6:     σ_r = σ ∪ {Σ_{k=1}^{d-1} h[k]θ_k ≥ 1}
7:     p = a point in c s.t. σ_r is satisfied
8:     if O(∇_fp(D)) = True then return p
9:     return
10: end if
11: σ_l = σ ∪ {Σ_{k=1}^{d-1} T.h[k]θ_k ≤ 1}
12: if h passes through σ_l then
13:     if p =ATC+ (T,h,c,σ_l) is not null then return p
14: end if
15: σ_r = σ ∪ {Σ_{k=1}^{d-1} T.h[k]θ_k ≤ 1}
16: if h passes through σ_r then
17:     if p =ATC+ (T,h,c,σ_r) is not null then return p
18: end if
```

E COLORING CELLS OUTSIDE SATISFACTORY REGIONS

After identifying the cells C that intersect with some satisfactory region, and assigned a satisfactory function to each of them, here we focus on cells $\bar{C}$ that do not contain a satisfactory function. For ease of explanation, we will represent the satisfactory function assigned to cell $c \in C$ with the color of c (see Figure 23). For each cell $c' \in \bar{C}$, our objective is to find the closest satisfactory function to the center of c', and to color c' accordingly (see Figure 24).

To do so, we implement an algorithm (CELLCOLORING) that uses monotonicity of the angular distance and adopts

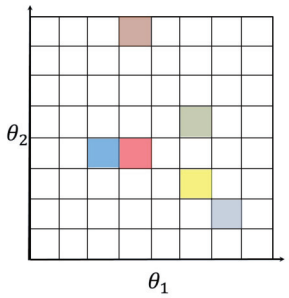


Figure 23: Satisfactory cells example

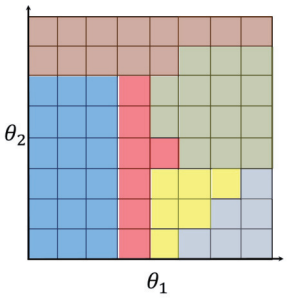


Figure 24: Coloring unsatisfactory cells in Fig. 23

Dijkstra's algorithm [20]. The algorithm initially sets the distance of the satisfactory cells to zero, and the distance of all other cells to ∞, and adds them to a priority queue Q. Then, while Q is not empty, it visits the cell c with the minimum distance, and remove it from Q. For all neighbors of c that are still not visited and their distances are more than the angular distance of their center with $F[c]$, the algorithm updates their distance and position in the queue, and sets their color to $F[c]$.

Since the number of neighbors of each cell is fixed, it is easy to see that CELLCOLORING is in $O(N \log N)$ [20].

F PROOFS

THEOREM 1.

PROOF. The proof is straightforward, following the number of ordering exchanges. Ordering the items and adding the initial ordering exchanges is in $O(n \log n)$. Since every pair of items in 2D has at most one ordering exchange, the total number of ordering exchanges is in $O(n^2)$. The algorithm sweeps over the ordering exchanges and for each of them spends $O(\log n)$ for the heap operations and Υ_n for accessing the oracle. Therefore 2DRAYSWEEP is in $O(n^2(\log n + \Upsilon_n))$. □

THEOREM 2.

PROOF. There totally are $O(n^2)$ ordering exchanges for n items. Therefore, the number of satisfactory regions in 2D is in $O(n^2)$. Applying binary search on this list is $O(\log n)$. □

THEOREM 3.

PROOF. Lines 2 to 6 of SATREGIONS construct $h_{i,j}$ for each pair of the items t_i and t_j (in the dual space). Since Algorithm 4 has a constant complexity for a fixed number of dimensions, constructing the ordering exchanges in the angle coordinate system is in $O(n^2)$. The next step of the algorithm is constructing the arrangement of hyperplanes. Using results from combinatorial geometry, the complexity of the arrangement of n^2 hyperplanes in $\mathbb{R}^{d-1}$ is $O(n^{2(d-1)})$ [17]. The bottleneck in Algorithm 5 is the construction of the arrangement: at iteration i, add the i^{th} hyperplane to the arrangement. To do so, identify the set of regions with which the current hyperplane intersects, by applying a linear scan

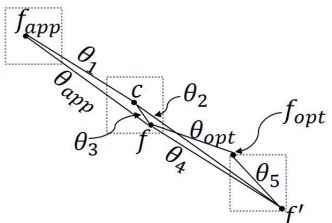


Figure 25: Illustration of θ_{app} v.s. θ_{opt}

over the set of regions to find intersections. Furthermore, for each region, Algorithm 5 solves an LP with i^2 constraints over a fixed number of variables. The number of regions at iteration i is $i^{2(d-1)}$. Thus the total cost is:

$$O(\sum_{i=1}^{n^2}(i^{2(d-1)}Lp(i^2))) \leq O(n^{2d}Lp(n^2))$$

After constructing the arrangement, the algorithm removes the unsatisfactory regions from $\mathcal{R}$. To do so, for each region, it chooses a function inside the regions, orders the items based on it, and calls the oracle to check if it is satisfactory. There are $O(n^{2(d-1)})$ regions in the arrangement and ordering the items in each region is in $O(n \log n)$. Hence, this step is in $O(n^{2d-1} \log n \, \Upsilon_n)$. The time complexity of the algorithm, therefore, is: $O\big(n^{2d-1}(n\, Lp(n^2) + \Upsilon_n \log n)\big)$ □

Theorem 4.

Proof. Given that the upper-bound on the total number of satisfactory regions, is $O(n^{2(d-1)})$, the proof is straightforward. For every satisfactory region, MDBASELINE needs to solve a non-linear programming problem of size $O(n^2)$ constraints over fixed number of variables. Thus, Algorithm 6 is in $O(n^{2(d-1)}NLp(n^2))$. □

Theorem 5.

Proof. The proof follows the fact that ordering the items based on the input function is in $O(n \log n)$ while finding its corresponding cell, using binary search is in $O(\log N)$. □

Theorem 6.

Proof. Let c_{app} and c_{opt} be the cells f_{app} and f_{opt} belong to. First, there should exists a satisfactory function f' inside c_{opt} that is assigned to it. That is because f_{opt} belongs to c_{opt} and thus its intersection with the satisfactory regions is not empty. Figure 25 illustrates such a setting. The point c in the figure shows the center of the cell that f belongs to. Since f_{app} is assigned to this cell, the angle distance between f_{app} and c (θ_1 in the figure) is less than the angle distance between f' and c (θ_2 in the figure). Let θ_3, θ_4, and θ_5 (as specified in the figure) be the angle distance between c and f, f and f', and f' and f_{opt}, respectively. Following the triangular inequality:

$$\theta_{app} \leq \theta_1 + \theta_3,\ \theta_4 \geq \theta_2 - \theta_3$$
$$\Rightarrow \theta_{app} + \theta_2 - \theta_3 \leq \theta_1 + \theta_3 + \theta_4$$
$$\Rightarrow \theta_{app} \leq \theta_4 + 2\theta_3$$

id	x	y	color
t_1	1	0	blue
t_2	0.95	$0.1\tan((p\pi)/(2s) - \epsilon)$	red
t_3	0.9	$0.1\tan((p\pi)/(2s) - \epsilon)$	red
t_4	0	1	red

Figure 26: The contradictory dataset; $\epsilon \to 0^+$

Similarly:

$$\theta_4 \leq \theta_5 + \theta_{opt}$$
$$\Rightarrow \theta_{app} \leq \theta_{opt} + \theta_5 + 2\theta_3$$

Let θ_r be the diameter of each cell. Looking at the figure, $\theta_5 \leq \theta_r$ and $\theta_3 \leq \theta_r/2$. Thus: $\theta_{app} \leq \theta_{opt} + 2\theta_r$ □

Theorem 7.

Proof. We provide the proof by contradiction. Assume there exist a number s and a probability p such that the probability of finding a satisfactory function after s samples is at least p. We construct the dataset $\mathcal{D}$ shown in Figure 26 with two attributes x and y and a binary protected attribute color and four items. We define a ranking to be fair, if with its top-2 there are one red and one blue item. Note that ranking $\mathcal{D}$ based on the x attribute ($\theta = 0$) generates the same ordering shown in the figure, in which t_1 and t_2 are the top-2. Based on our fairness criteria, this ranking is fair, as its top-2 contains one blue and one red item. Following Algorithm 1 (moving a ray from x-axis toward y-axis), t_1 and t_2 quickly exchange ordering, but the top-2 does not change and, hence, remains fair. Next, t_1 exchanges ordering with t_3 at:

$$\theta_{t_1,t_3} = \arctan(\tan(\frac{p\pi}{2s} - \epsilon)) = \frac{p\pi}{2s} - \epsilon$$

Note that at this point the ordering changes to $\langle t_2, t_3, t_1, t_4\rangle$ and is no longer fair, since both items in its top-2 are red. Also, after this point t_1 never appears in the top-2, hence, there exists no fair region after this point.

Therefore, the only satisfactory region in this dataset is $\langle 0, \frac{p\pi}{2s} - \epsilon\rangle$. The probability that a uniformly drawn sample function will land in this region is:

$$\frac{p\pi/(2s) - \epsilon}{\pi/2} < \frac{p}{s}$$

By the union bound from probability, the probability that at least one of these samples land into the satisfactory region is at most:

$$s\frac{p\pi/(2s) - \epsilon}{\pi/2} < p$$

This contradicts with the assumption that the probability of finding a satisfactory function after s samples is at least p. □