Designing Fair Ranking Schemes

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

†University of Michigan, †Drexel University, ††University of Texas at Arlington

†{asudeh, jag}@umich.edu, †stoyanovich@drexel.edu, ††gdas@uta.edu

ABSTRACT

Items from a database are often ranked based on a combination of multiple criteria. A user may have the flexibility to accept combinations that weigh these criteria differently, within limits. On the other hand, this choice of weights can greatly affect the fairness of the produced ranking. 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 vector of weights, or as a point in a multi-dimensional space. For a broad range of fairness criteria, 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. We develop user-controllable approximation that and indexing techniques that are applied during preprocessing, and support sub-second response times during the online phase. Our extensive experiments on real datasets demonstrate that our methods are able to find solutions that satisfy fairness criteria effectively and efficiently.

1. INTRODUCTION

Data-driven algorithmic decisions are commonplace today. Because of the impact these decisions have on individuals and on population groups, issues of algorithmic bias and discrimination are coming to the forefront of societal and technological discourse [7]. In the seminal work of Friedman and Nissenbaum [21] a biased computer system is 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.

A prominent source of bias in data-driven systems is the data itself, a phenomenon colloquially known as "racism in — racism out". For example, it has been shown that machine learning models trained on biased data will produce biased results, further propelling historical discrimination [16]. Naturally, the effects of biased data are not limited to machine learning scenarios, but also

impact processes that are directly designed and validated by humans. Perhaps the most immediate example of such a process is a score-based ranker. In this paper we consider the task of *designing a fair score-based ranking scheme*.

Ranking of individuals is ubiquitous, and is used, for example, to establish credit worthiness, desirability for college admissions and employment, and attractiveness as dating partners. 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 scorebased rankers compute the score of an individual as a linear combination of attribute values, with non-negative weights. Designing a ranking scheme amounts to selecting a set of weights, one for each feature, and validating the outcome on the database \mathcal{D} .

Our goal is to assist the user in designing a ranking scheme that both reflects a user's a priori notion of quality and is fair, in the sense that it mitigates *preexisting bias with respect to a protected feature* that is embodied in the data. In line with prior work [17,27, 31–33], a protected feature denotes membership of an individual in a legally-protected category, such as persons with disabilities, or under-represented groups by gender or ethnicity. Interpreting the definition of Friedman and Nissenbaum [21] for rankings, a biased outcome occurs when a ranking decision is based fully or partially on a protected feature. Discrimination occurs when this outcome is systematic and unfavorable, for example, when minority ethnicity or female gender systematically lead to placing individuals at lower ranks. To make our discussion concrete, we consider an example.

EXAMPLE 1. A college admissions officer is evaluating 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, and use these in a score-based ranking scheme.

As the first step, to make the two 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 a 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 [28].

The system will then assist the user in identifying a new scoring functions $f'(t) = 0.45 \times s + 0.55 \times g$, which meets the fairness con-

straint and is close to the original function f in terms of attribute weights, thereby reflecting the user's a priori notion of quality.

In machine learning, the common setup is to have training data for which we know the outcome (label), and then the problem is to have the system learn weights (or other model parameters) that result in an algorithm that maximizes the predicted outcome (or correctness of label). While this problem setup is appropriate for many tasks, it requires unreasonably simplistic assumptions in many others. For example, what is the outcome an admissions officer seeks to maximize in admitting students? Some outcomes are relatively easy to measure, such as GPA after admission and enrollment. But what the university presumably really cares about is long-term success: the admissions officer is looking for students who will go on to become rich or famous or successful in some other dimension that matters. This outcome is fuzzy, multi-dimensional, and hard to measure. It is also long-term — the algorithm cannot really be tuned for today based on data regarding students admitted 30 years ago. For these reasons, many practical systems have simple models with weights set by human experts, usually in a subjective manner.

Precisely because these weights are often subjectively chosen, we have an even greater fear of discrimination than just algorithmic bias. In fact, 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 achievements alone [23, 24].

In this paper, we consider this sort of reverse problem: the selection of model weights after we already know (the distribution of attribute values in) the dataset to be scored and ranked. The goal is to select weights such that desired fairness and diversity criteria are satisfied. To be certain of meeting these criteria, the weights have to be selected after we have the dataset in hand. If we know that the distribution of values in the dataset will not change too much over some window, we can go through a design process to choose model weights once using a representative sample of the data, and then just reuse the same model and weights for each dataset that follows. We may still wish to verify that we continue to meet the required criteria, and adjust our ranking function if needed. In short, the choice of ranking function is not a one-time thing. Rather, in practice, ranking functions are frequently tuned, typically with small changes.

As such, we repeatedly have a human model designer trying to tune model weights. It may be acceptable for this tuning process 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 human designer of a ranking function would want the system to support her work through interactive response times. Our goal is to meet this need, to the extent possible.

In the remainder of this paper, we will present a *query answering system* that assists the user in designing fair score-based rankers. As the first step, the system pre-processes a dataset of candidates off-line, and is then able to handle user requests in real time. The user specifies a *query* in the form of a scoring function f, which associates non-negative weights with item attributes and is used to compute items scores, and to sort the items 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. If the list of items was found to be unsatisfactory, we will suggest to the user an alternative scoring function f' that is both satisfactory and close to the query f.

The user may accept the suggested function f', or she may decide to manually adjust the query and invoke our system once again.

Numerous fairness definitions have been considered in the literature [14, 33]. 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 [14]. These two requirements represent intrinsically different world views, and accommodating both may require tradeoffs [20]. Our focus is on group fairness, which is based on the relationship between (1) membership of individuals in demographic groups and (2) their ranked outcome.

While fairness in algorithmic systems is an active area of research [33], our work is among a small handful of studies that focus on fairness in ranking [9, 31, 32]. While others considered mitigating bias in the output of a ranker [9, 32], or incorporating fairness constraints into ranked models [31], our work is the first to support the user in designing fair ranking schemes.

Our methods are general, and can accommodate a large class of group fairness constraints — including those based on asserting a minimum or a maximum number of individuals at the top-k that belong to a particular demographic group, as in Example 1 and in the work of Celis et al. [9], but going far beyond this class. In fact, our techniques treat the evaluation of fairness constraints as a black box (embodied by the fairness oracle), and support any constraint that can be evaluated over a ranked list of items. We are not limited to binary protected group membership (such as gender in Example 1, and in the work of Zehlike et al. [32]), and can accommodate fairness constraints on multiple non-overlapping population groups (such as ethnicity). Further, we support constraints that are stated over more than one sensitive attribute; for example, we can enforce constraints on gender, ethnicity and age group simultaneously.

Summary of contributions: In this paper, we assist the user in designing fair score-based ranking schemes. Towards this goal, 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 any 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 techniques to find the nearest scoring function f' that satisfies the constraint (or to state that the constraint is not satisfiable). Our contributions are as follows:

- We propose a query answering system that helps users choose ranking functions that meet fairness requirements. (§ 1)
- Carefully defining the terms, problem statement, and assumptions (§ 2), we pursue offline indexing of the data that helps in online answering of users' queries.
- We introduce the notion of ordering exchange and use a transformation of the items to dual space to identify satisfactory regions, in which fairness constraints are met. (§ 3)
- We propose the ray sweeping algorithm 2DRAYSWEEP for indexing the satisfactory regions in two-dimensional space and 2DONLINE, an exact logarithmic binary search-based algorithm that takes as input a user's query f and proposes an alternative f' in real time. (§ 3)
- For studying the linear ranking functions in fixed-size multi dimensional spaces, we introduce an angle coordinate system. We

propose HYPERPOLAR to transform the ordering exchanges in the angle coordinate system. (\S 4)

- We use "the arrangement of hyperplanes" [15, 25] and propose the polynomial time exact algorithms SATREGIONS and MD-BASELINE for identifying satisfactory regions and proposing fair scoring functions to a user in a space with arbitrary number of dimensions. (§ 4)
- We propose the arrangement tree data structure for optimizing the running time of SATREGIONS. (§ 4)
- We propose a user-controllable grid partitioning of the angle coordinate system that guarantees a maximum angle distance between every pair of points in a cell.
- We use the grid partitioning of the angle coordinate system and propose an approximate algorithm (for multi-dimensional space) that guarantees a controllable distance from the optimal solution, while enabling efficient processing of users' queries. This approximate algorithm provides opportunities for speeding up the indexing algorithms by limiting the arrangements to each cell and applying an early stopping strategy. We propose algorithms CELLPLANEx, MARKCELL, and CELLCOLORING for indexing, and an efficient online algorithm MDONLINE for answering users' queries. (§ 5)

In addition to the theoretical analyses, we conduct extensive experiments on real datasets that confirm the efficiency and effectiveness of our techniques, as described in \S 6. Related work and conclusions are discussed in \S 7 and \S 8, respectively.

2. PRELIMINARIES

Data model: We are given a dataset \mathcal{D} of n items, each with d scalar scoring attributes¹. We represent an item t as a d-long vector of scoring attributes, $\{t[1], t[2], \ldots, t[d]\}$. 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: Our focus in this paper is on the class of linear ranking functions that use a weight vector $\vec{w} = \{w_1, w_2, \dots, w_d\}$ to compute a goodness score $f_{\vec{w}}(t)^2$ of item t as $\Sigma_{j=1}^d w_j t[j]$. Without loss of generality, we assume each weight $w_j \in \vec{w} \geq 0$. The scores of items are used for ranking them. We assume that an item with a higher score outranks an item with a lower score.

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 $\vec{w} = \{w_1, w_2, ..., w_d\}$. 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 $\{1,1\}$. 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 $\vec{w} = \{w_1, w_2, \ldots, w_d\}$ induces the same ordering on the items as does its linear scaling $\vec{w'} = \{c.w_1, c.w_2, \ldots, c.w_d\}$, 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

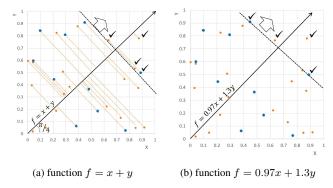


Figure 1: 2D, the effect of two similar functions on output fairness.

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. Further details for computing the angular distance between two functions are provided in Appendix A.1. 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) [4, 6]. For linear functions, the contour of an item t is the hyperplane t that is perpendicular to the ray of t and passes through t.

Fairness model: We adopt a general ranked fairness model, in which a fairness oracle \mathcal{O} takes as input an ordered list of items from \mathcal{D} , and determines whether the list meets fairness constraints: $\mathcal{O}: \operatorname{ordered}(\mathcal{D}) \to \{\top, \bot\}$. 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 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 [9,27,32]: 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 corresponding weight vector, may not satisfy the required fairness constraints. Our problem is to propose a scoring function f' with a similar weight vector as 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. The formal statement of our problem is as follows:

¹Additional non-scalar attributes are considered in the fairness model.

²To simplify notation, we use f(t) to refer to $f_{\vec{w}}(t)$.

CLOSEST SATISFACTORY FUNCTION:

Given a dataset \mathcal{D} with n items over d scalar scoring attributes, a fairness oracle $\mathcal{O}: ordered(\mathcal{D}) \to \{\top, \bot\}$, and a linear scoring function f with the weight vector $\vec{w} = \{w_1, w_2, \cdots, w_d\}$, find the function f' with the weight vector \vec{w}' such that $\mathcal{O}(OrderBy_{f'}(\mathcal{D})) = \top$ 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 dataset, and develop data structures that will be useful in the online phase. In the online phase, we will exploit these data structures to quickly find similar satisfactory weight vectors. 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 \S 3 will help us in \S 4 for developing algorithms for the general multi-dimensional case where the number of scalar scoring attributes is d > 2.

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. The problem in 2D is easier to understand, visualize, and explain, and allows us to create the foundation for the general problem, which we will address in subsequent sections. 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 $t_1\succeq t_2$.

Given the importance of the place where items t_1 and t_2 switch ordering, we define the *ordering exchange* as the ranking functions

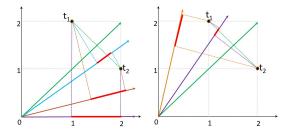


Figure 2: The ordering exchange between a pair of points.

according to which t_1 and t_2 are equally good. 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

The goal of offline processing is to identify and index the satisfactory functions in a way that allows 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 [15], where every item t is transformed into the line d(t), as follows:

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

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

Using Equation 3, the intersection of the lines $d(t_i)$ and $d(t_j)$ can be computed by solving the following system of equations:

$$\times_{\mathsf{d}(t_i),\mathsf{d}(t_j)} : \left\{ \begin{array}{l} t_i[1]x + t_i[2]y = 1 \\ t_j[1]x + t_j[2]y = 1 \end{array} \right.$$

The ordering exchange is the origin-starting ray with the angle:

$$\Rightarrow x = \left(1 - \frac{t_i[2]}{t_j[2]}\right) / \left(t_i[1] - \frac{t_j[1]t_i[2]}{t_j[2]}\right)$$

$$\Rightarrow y = \frac{1 - t_i[1]x}{t_i[2]}$$

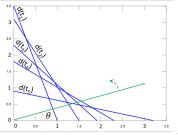
$$\Rightarrow \theta = \arctan(y/x) \tag{2}$$

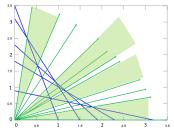
Now, we use the ordering exchanges to design the ray sweeping algorithm 2DRAYSWEEP , presented in Algorithm 1: the algorithm first computes the ordering exchange between the pairs of items that do not dominate each other using Equation 2, and adds them, in addition to the angle 0, to a list. Next, Line 9 sorts the angles in an ascending order. Then it orders the items in $\mathcal D$ based on the x-axis (angle 0) and gradually updates the ordered list (Ω) as it sweeps the

³This is because the line showing the contour of t, transforms to the intersection point of d(t) and the ray of f.

 $^{^4}t$ dominates t' if $\forall i\in[1,d],\ t[i]\geq t'[i]$ and $\exists j\in[1,d]$ such that t[j]>t'[j] . [5]

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





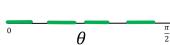


Figure 3: A 2D dataset

Algorithm 1 2DRAYSWEEP

21:

23:

24: end for

26: **return** *S*

 $\langle \Theta[j], 0 \rangle$

flag = sign

25: **if** flag = True **then** append $(S, \langle \pi/2, 1 \rangle)$

Figure 4: Dual presentation of Fig. 3

Figure 5: Satisfactory sectors

Figure 6: Satisfactory regions

```
Input: dataset \mathcal{D} and fairness oracle \mathcal{O}
Output: sorted satisfactory regions S
 1: \Theta = \{0\}
 2: for i = 1 to n-1 do
 3:
         for j = i+1 to n do
            if t_i or t_j dominates the other then continue
 4:
 5:
            \theta_{ij} = Angle of ordering exchange of t_i and t_j (Eq. 2)
            add (\theta_{ij}, t_i, t_j) to \Theta
 6:
 7:
         end for
 8: end for
 9: sort \Theta
10: \Omega = \text{sort } \{t_i \in \mathcal{D}\}\ \text{on } x\text{-axis, } i = 1
11: while i < |\Theta| and \mathcal{O}(\Omega) = False do
12:
         i = i + 1
13:
         (\theta, a, b) = \Theta[i]
         swap a and b in \Omega
14:
15: end while
16: S = [\langle \Theta[i], 0 \rangle], flag = True
17: for j = i + 1 to |\Theta| - 1 do
18:
         (\theta, a, b) = \Theta[j]
19:
         swap a and b in \Omega
20:
         sign = \mathcal{O}(\Omega)
```

ray toward the y-axis (angle $\pi/2$), by changing the order of pairs of items in their ordering exchanges. Upon finding a satisfactory sector, the algorithm continues attaching the neighboring sectors as long as those are still satisfactory, to generate a satisfactory region.

if flag = True and sign = False then append $(S, \langle \Theta[j], 1 \rangle)$

else if flag = False and sign = True then append(S,

Algorithm 1 stores the borders of the 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. One can see the third from the left satisfactory region is the union of two neighboring satisfactory sectors in Figure 5.

THEOREM 1. Algorithm 1 has time complexity $O(n^2(\log n + \mathbb{O}_n))$, where \mathbb{O}_n is the time complexity of \mathcal{O} for input of size of n.

PROOF. The proof for this theorem is straightforward, following the number of ordering exchanges. Since every pair of items in 2D has at most one ordering exchange, the total number of ordering

exchanges is in $O(n^2)$. Sorting the ordering exchanges in Line 9 is in $O(n^2 \log n)$. Sorting the items along the x-axis is in $O(n \log n)$. Then in lines 11 to 24, the algorithm gradually updates the ranked list as it moves from each sector to the next one. For each of the sectors, it calls the oracle once to check if it is satisfactory. This is in $O(n^2 \mathbb{O}_n)$. Therefore 2DRAYSWEEP is in $O(n^2 (\log n + \mathbb{O}_n))$.

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 the users' 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 a function f, applies binary search on the sorted list of satisfactory regions. If f falls within 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 $f:\{w_1,w_2\}$ **Output:** weight vector $\{w_1',w_2'\}$

```
1: (r,\theta) = (\sqrt{w_1^2 + w_2^2}, \arctan \frac{w_2}{w_1})
 2: low = 1, high = |S|
 3: while (high-low) > 1 do
       mid = (low+high)/2
       if S[mid][1] < \theta then low = mid
 5:
 6:
       else high = mid
 7:
    end while
 8:
    if S[low][2] = 0 then
       return \{w_1, w_2\} // input vector is satisfactory
11: if (\theta - S[low][1]) < (S[high][1] - \theta) then
       return \{r\cos(S[low][1]), r\sin(S[low][1])\}
14: return \{r\cos(S[\text{high}][1]), r\sin(S[\text{high}][1])\}
```

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

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

4. THE MULTI-DIMENSIONAL CASE

In general, more than two attributes may be used for ranking. We now extend the basic framework introduced in \S 3 to handle multi-dimensional cases. The challenge is that regions of interest are no longer simple planar wedges, bounded by two rays at an angle.

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 [15]. We then propose an exact online algorithm that works based on the indexed satisfactory regions.

4.1 Ordering exchange in angle coordinates

Consider function f with weight vector $\vec{w} = \{w_1, w_2, \cdots, w_d\}$. 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 :

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

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

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

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

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

For example, 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.

Before we can construct satisfactory regions, we first need to compute ordering exchanges in the angle coordinate system. Algorithm 3 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

```
Input: items t_i and t_j
Output: ordering exchange h_{i,j}

1: V = [t_i[k] - t_j[k], \forall 1 \le k \le d] // Equation 5
2: \Theta = \{\}
3: p = d - 1 linearly independent points satisfying Equation 5
4: for k = 1 to d - 1 do
```

7: end for // Find the hyperplane containing the points in Θ 8: $\iota = [1,1,\cdots,1]$

9: **return** $\Theta^{-1} \times \iota$

add θ to Θ

2. **Teturn** 0 × v

 $(r, \theta) = \mathbf{ToPolar}(p[k])$

Algorithm 3 HYPERPOLAR

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 3, it is easy to see that HYPERPOLAR is in $O(d^3)$, which is O(1) for a fixed d.

4.2 Construction of satisfactory regions

The construction of satisfactory regions relates to the arrangement [15] 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 [15]. Every hyperplane h divides the space into two half-spaces h^+ and h^- . In our problem, 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 [15], we develop an incremental algorithm for discovering the convex regions in the arrangement. Intuitively, Algorithm 4 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.

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

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 4 breaks it down into two convex regions that represent the intersections of R with half-spaces h^+ and h^- .

Having constructed the arrangement, Algorithm 4 finds, using linear programming, a point θ that satisfies σ_R , and uses θ to check if region R is satisfactory. If R is not satisfactory, it is removed from the set of satisfactory regions \mathcal{R} .

ſ	t_1	1	2	3
Γ	t_2	2	4	1
Γ	t_3	5.3	1	6
Γ	t_4	3	7.2	2

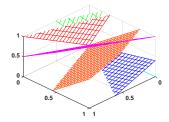


Figure 7: A 3D dataset

Figure 8: Ordering exchanges for Fig. 7

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

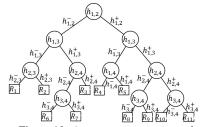


Figure 10: Arrangement tree example

```
Algorithm 4 SATREGIONS Input: dataset \mathcal{D} and fairness oracle \mathcal{O} Output: satisfactory regions \mathcal{R}
```

remove R from R

24:

25:

end if

26: **end for**

27: return \mathcal{R}

```
1: H = \{\}
     // construct ordering exchanges in angle coordinates
 2: for i = 1 to n - 1 do
 3:
        for j = i + 1 to n do
           if t_i or t_j dominates the other then continue
 4:
 5:
           add HYPERPOLAR (t_i, t_i) to H
 6:
        end for
 7: end for
 8: \mathcal{R} = \{ \{ (H[1], +) \}, \{ (H[1], -) \} \}
     // add hyperplanes incrementally to the arrangement
 9: for h \in (H \setminus \{H[1]\}) do
10:
        \ell_{\mathcal{R}} = |\mathcal{R}|
        for i=1 to \ell_{\mathcal{R}} do
11:
            if \exists p \in h \text{ s.t. } \sigma_{\mathcal{R}[i]} then
12:
13:
               R' = \mathcal{R}[i]
14:
               append \mathcal{R}[i] by (h, +)
               append R' by (h, -)
15:
               add R' to \mathcal{R}
16:
17:
            end if
18:
        end for
19: end for
     // remove the unsatisfactory regions
20: for R \in \mathcal{R} do
        \theta = a point that \sigma_B is satisfied
21:
22:
         \vec{w} = \text{ToCartesian}(1,\theta)
23:
        if \mathcal{O}(\text{OrderBy}_{f_{\vec{w}}}(\mathcal{D})) = \text{False then}
```

THEOREM 3. For a fixed number of dimensions, the time complexity of Algorithm 4 is $O(n^{2d-1}(n Lp(n^2) + \mathbb{O}_n \log n))$, where $Lp(n^2)$ is the time of solving a linear programming problem of n^2 constrains and a fixed number of variables and \mathbb{O}_n is the time complexity of \mathcal{O} for an input of size n.

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 3 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 $O(n^{2(d-1)})$ [15]. The bottleneck in Algorithm 4 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 over the set of regions to find intersections. Furthermore, for each region, Algorithm 4 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:

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

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 $O(n^{2d-1}\log n \mathbb{O}_n)$. The time complexity of the algorithm, therefore, is:

$$O(n^{2d-1}(n Lp(n^2) + \mathbb{O}_n \log n))$$

To add a new hyperplane, Algorithm 4 checks the intersection of every region with the hyperplane. But in practice most regions do not intersect with it. In the following, we define the *arrangement tree*, 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 illustrate experimentally in § 6.4.

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 $\{h_{1,2}^-, h_{1,3}^+\}$.

Algorithm 5 shows the recursive algorithm for adding a hyperplane to an arrangement, using the arrangement tree. It replaces the lines 9 to 18 in Algorithm 4.

4.3 Online processing

Thus far in this section, we studied how to preprocess the data and construct satisfactory regions in multiple dimensions. Next,

Algorithm 5 AT₊

Input: arrangement tree T, the hyperplane h, the constraints path to root σ

```
1: if T is null then

2: T = \text{new ArrangementTree}(h)

3: return

4: end if

5: \sigma_l = \sigma \cup \{\sum_{k=1}^{d-1} T.h[k]\theta_k \leq 1\}

6: \sigma_r = \sigma \cup \{\sum_{k=1}^{d-1} T.h[k]\theta_k \geq 1\}

7: if h passes through \sigma_l then \text{AT}_+(T.\text{left},h,\sigma_l)

8: if h passes through \sigma_r then \text{AT}_+(T.\text{right},h,\sigma_r)
```

given a query (a function f) and the satisfactory regions, our objective is to find the closest satisfactory function f' to f. To do so, MDBASELINE solves a non-linear programming problem for each satisfactory region to find the closest point of the region to f. It then returns the function with the minimum angle distance with f.

Algorithm 6 MDBASELINE

Input: Satisfactory regions \mathcal{R} , dataset \mathcal{D} , fairness oracle \mathcal{O} , function $f: \vec{w}$

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

```
1: if \mathcal{O}(\text{OrderBy}_{f_{\overrightarrow{w}}}(\mathcal{D})) = \text{True then}
 2:
         return \vec{w}
 3: end if
 4: (r, \Theta^{(i)}) = \mathbf{ToPolar}(\vec{w})
 5: mindist=∞
 6: for R \in \mathcal{R} do
         (\mathrm{dist},\Theta^{(j)}) = the minimum \theta_{i,j} such that C_R is satisfied //
         based on Equation 10
 8:
         if dist<mindist then
 9:
             \Theta^o = \Theta^{(j)}, mindist=dist
10:
         end if
11: end for
12: return ToCartesian(1,\Theta^o)
```

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.

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 $O(n^{2(d-1)}NLp(n^2))$.

5. APPROXIMATION

A user developing a scoring function requires an interactive response 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. We conduct the partitioning in a way that the maximum angle distance

between every pair of functions in every cell is bounded. lease see the details in Appendix A.2. In the 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, in \S 5.1, we first identify the cells that intersect with a satisfactory region, and assign the corresponding satisfactory function to each such cell. Then, in \S 5.2, 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,

$$C = \{ c \in Cells \mid \exists R \in \mathcal{R} \text{ s.t. } R \cap c \neq \emptyset \}$$
 (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. Given the potentially large values of N and size of \mathcal{R} , this is inefficient. As discussed in § 4, and experimentally shown in § 6, the complexity of the arrangement and the running time of Algorithm 4 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. On the other hand, 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.

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.

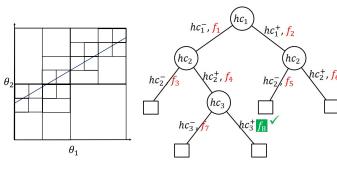
Recall that HYPERPOLAR constructs the hyperplane h in the form of $\sum_{k=1}^{d-1} h[k]\theta_k = 1$. Thus, for every point p in h^- , $\sum_{k=1}^{d-1} p_k \theta_k \leq 1$ while for every point p' in h^+ , $\sum_{k=1}^{d-1} p_k' \theta_k \geq 1$. Therefore, h passes through c, iff $\sum_{k=1}^{d-1} bl[k]\theta_k \geq 1$ and $\sum_{k=1}^{d-1} tr[k]\theta_k \geq 1$.

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* [18] data structure for this purpose. To do so, the recursive Algorithm 7 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.

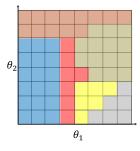
After identifying \mathcal{HC} (the sets of hyperplanes passing through the cells), for each cell $c \in Cells$, we limit the arrangement to $\mathcal{HC}[c]$. Moreover, note that in this step our goal is to find a satisfactory function inside c. This is different from our objective in



that intersect a hyperplane

Figure 11: Identifying cells Figure 12: Early stopping when construct- Figure 13: Satisfactory cells Figure 14: Coloring unsatising the arrangement of a cell

θ_1



example

factory cells in Fig. 13

Algorithm 7 CELLPLANE ×

Input: hyperplane h, Cells, low (indices of bottom-left corner), high (indices of top-right corner), turn (the dimension to divide), and list of hyperplanes for cells \mathcal{HC}

```
1: if h does not passes through
    · · · rectangle(bottom-left(low),top-right(high)) then return
2: if high[turn] = low[turn] then
      if \forall 1 \leq i \leq (d-1): low[i]=high[i] then
         add h to HC[low] and return
4:
5:
       while high[turn] = low[turn] turn = (turn + 1)mod(d-1)
6:
7: end if
8: mid = low[turn]+high[turn]/2
9: tmp= high[turn]; high[turn] = mid
10: CELLPLANE_{\times} (h, Cells, low, high, (turn+1) mod(d-1), \mathcal{HC})
11: high[turn]=tmp; low[turn] = mid+1
```

12: CELLPLANE $_{\times}$ (h, Cells, low, high, (turn+1) mod(d-1), \mathcal{HC})

Algorithm 8 MARKCELL Input: cell c, \mathcal{HC}

```
1: if |\mathcal{HC}[c]| = 0 then
         p = a point inside c
 3:
         if \mathcal{O}(\text{OrderBy}_p(\mathcal{D})) = \text{True then Marked}[c] = p
 4:
         return
 5: end if
 6: p = a point in \mathcal{HC}[c][1]^- \cap c
 7: if \mathcal{O}(\text{OrderBy}_p(\mathcal{D})) = \text{True then Marked}[c] = p; return
 8: p = a \text{ point in } \mathcal{HC}[c][1]^+ \cap c
 9: if \mathcal{O}(\text{OrderBy}_p(\mathcal{D})) = \text{True then Marked}[c] = p; return
10: T = \text{new ArrangementTree}(\mathcal{HC}[c][1])
11: for h \in \mathcal{HC}[c] \backslash \mathcal{HC}[c][1] do
         if p = ATC_+ (T,h,c,null) is not null then
12:
13:
             Marked[c] = p; return
14:
         end if
15: end for
```

SATREGIONS, where we wanted to find all satisfactory regions. This gives us the opportunity to apply a stop early strategy, as follows: at every iteration, while using the arrangement tree for the construction, check a function inside the newly added regions, and stop as soon as a satisfactory function is discovered.

Algorithm 8, MARKCELL, assigns a satisfactory function to the cells that intersect with a satisfactory region R. It calls Algorithm 9 that adds the new hyperplanes and checks if a function inside the new regions is satisfactory. Both algorithms stop as soon as they

Algorithm 9 ATC₊

Input: arrangement tree T, hyperplane h, cell c, constraints path to root σ

```
1: if T is null then
 2:
          T = \text{new ArrangementTree}(h)
          \sigma_l = \sigma \cup \{\sum_{k=1}^{d-1} h[k]\theta_k \le 1\}
 3:
          p = a \text{ point in } c \text{ s.t. } \sigma_l \text{ is satisfied}
 4:
 5:
          if \mathcal{O}(\operatorname{OrderBy}_p(\mathcal{D})) = \operatorname{True} then return p
          \sigma_r = \sigma \cup \{\sum_{k=1}^{d-1} h[k]\theta_k \ge 1\}
 7:
          p = a point in c s.t. \sigma_r is satisfied
 8:
          if \mathcal{O}(\text{OrderBy}_p(\mathcal{D})) = \text{True then return } p
 9:
          return
10: end if
11: \sigma_l = \sigma \cup \{\sum_{k=1}^{d-1} T.h[k]\theta_k \le 1\}
12: if h passes through \sigma_l then
13:
          if p = ATC_+ (T, h, c, \sigma_l) is not null then return p
14: end if
15: \sigma_r = \sigma \cup \{\sum_{k=1}^{d-1} T.h[k]\theta_k \le 1\}
16: if h passes through \sigma_r then
          if p = ATC_+(T,h,c,\sigma_r) is not null then return p
18: end if
```

find a satisfactory function and assign it to the cell.

Figure 12 illustrates how Algorithm 8 finds a satisfactory function for cell c. After adding hyperplanes hc_1 and hc_2 , since functions f_1 to f_6 are unsatisfactory (denoted by red color), Algorithm 8 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 without constructing the rest of the arrangement.

Considering $|\mathcal{HC}[c]|$ as the total number of hyperplanes passing through a cell c, the complexity their arrangement is $O(|\mathcal{HC}[c]|^{d-1})$. Thus, adopting Theorem 3 for Algorithm 8, its time complexity is $O(|\mathcal{HC}[c]|^d Lp(|\mathcal{HC}[c]|) + |\mathcal{HC}[c]|^{d-1} n \log n \mathbb{O}_n)$ for a fixed d.

Coloring cells outside satisfactory regions 5.2

So far, we identified cells C that intersect with some satisfactory region, and assigned a satisfactory function to each of them. We now focus on cells C that do not contain a satisfactory function. For ease of explanation, we will represent the satisfactory function assigned to cell $c \in \mathcal{C}$ with the color of c (see Figure 13). For each cell $c' \in \bar{\mathcal{C}}$, our objective is to find the closest satisfactory function to the center of c', and to color c' accordingly (see Figure 14).

To do so, we implement CELLCOLORING, an algorithm that uses monotonicity of the angular distance and adopts Dijkstra's algorithm [19]. 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].

Algorithm 10 CELLCOLORING

Input: Satisfactory cells C, unsatisfactory cells \bar{C} , and assigned functions to cells F

```
1: for c \in Cells do
       visited[c] = False
 2:
 3:
       if c \in \mathcal{C} then Q.add_with_priority(c, 0)
       else Q.add_with_priority(c', \infty)
 4:
 5: end for
 6: while Q is not empty do
 7:
       c = Q.\text{extract\_min}()
 8:
       visited[c] = True
       for each neighbor c' of c where visited [c] = False do
9:
10:
          alt = \theta_{F[c], center(c')}
11:
          if alt<dist[c'] then
12:
             dist[c'] = alt; F[c'] = F[c]
13:
             Q.decrease_priority(c', alt)
14:
          end if
15:
       end for
16: end while
```

Since the number of neighbors of each cell is fixed, it is easy to see that CELLCOLORING is in $O(N \log N)$ [19].

Applying CELLCOLORING completes offline preprocessing. After this step every cell in the partitioned angle space is assigned a satisfactory function⁵. We store the cell coordinates, together with the assigned satisfactory functions, as an index that enables online answering of user queries, discussed next.

5.3 Online processing

Given an unsatisfactory function f, we now need to find the cell to which f belongs, and to return the satisfactory function assigned to that cell. This is implemented by Algorithm 11.

Given a query f and the assigned functions to the cells, the algorithm transforms the weight vector of f to polar coordinates and then performs binary search on each dimension to identify the cell c to which f belongs. MDONLINE returns the satisfactory function of the cell, F[c].

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

PROOF. The proof simply 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 $O(\log N)$. \square

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 Algorithm 11 returns for f, based on the space partitioning parameter N. Then, $\theta_{app} \leq \theta_{opt}$

$$+ 4 \arcsin \left(\frac{\sqrt{d-1}}{2} \ ^{d-1} \sqrt[d-1]{\frac{\pi^{d/2}}{N2^{d-1}\Gamma(d/2)}} \right).$$

Algorithm 11 MDONLINE

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

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

```
    if O(OrderBy<sub>fw</sub>(D)) = True then
    return w̄
    end if
    (r, Θ) = ToPolar(w̄)
    for k = 1 to d − 1 do
    T = apply binary search on children of T and find the child to which Θ<sub>k</sub> belongs
```

7: end for

8: return F[T]

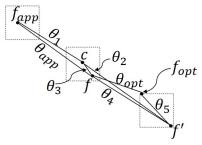


Figure 15: Illustration of θ_{app} v.s. θ_{opt}

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 15 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:

$$\theta_{app} \le \theta_1 + \theta_3, \ \theta_4 \ge \theta_2 - \theta_3$$

$$\Rightarrow \theta_{app} + \theta_2 - \theta_3 \le \theta_1 + \theta_3 + \theta_4$$

$$\Rightarrow \theta_{app} \le \theta_4 + 2\theta_3$$

Similarly:

$$\theta_4 \le \theta_5 + \theta_{opt}$$

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

Let θ_r be the diameter of each cell. Looking at the figure, $\theta_5 \le \theta_r$ and $\theta_3 \le \theta_r/2$. Thus:

$$\theta_{app} \leq \theta_{opt} + 2\theta_r$$

Following Equation 14, the diameter of the hypercube base of each cell is:

$$\eta_d = \sqrt{d-1} \sqrt[d-1]{\frac{\pi^{d/2}}{N2^{d-1}\Gamma(d/2)}}$$

Therefore, θ_r is:

$$\theta_r = 2\arcsin\left(\frac{\sqrt{d-1}}{2} \sqrt[d-1]{\frac{\pi^{d/2}}{N2^{d-1}\Gamma(d/2)}}\right)$$

⁵We assume the existence of at least one satisfactory region.

Hence

$$\theta_{app} \leq \theta_{opt} + 4 \arcsin \left(\frac{\sqrt{d-1}}{2} \sqrt[d-1]{\frac{\pi^{d/2}}{N2^{d-1}\Gamma(d/2)}} \right)$$

5.4 Sampling for large-scale settings

A critical requirement of our system is to be efficient during online query processing, and it is fine for it to spend more time in the offline preprocessing. As discussed in \S 4 and \S 5, the proposed offline algorithms are polynomial for a fixed value of d. In addition, the arrangement tree (c.f. \S 4) and the techniques of \S 5 speed up preprocessing in practice. However, preprocessing can still be slow, particularly for a large number of items. We reduce preprocessing time using sampling.

The main idea is that a uniform sample of the data maintains the underlying properties of the data distribution. Therefore, if a function is satisfactory for a dataset, it is *expected* to be satisfactory for a uniformly sampled subset. Hence, for a datasets with large numbers of items, one can do the 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 \S 6.

6. EXPERIMENTAL EVALUATION

6.1 Experimental Setup

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. We used the Python scipy.optimize ⁶ package for LP optimizations.

Datasets. All experiments are conducted on real datasets.

COMPAS: a dataset collected and published by ProPublica as part of their investigation into racial bias in criminal risk assessment software [3]. 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 the default dataset for our experiments.

US Department of Transportation (DOT): the flight on-time database published by DOT is widely used by third-party websites to identify the on-time performance of flights, routes, airports, and airlines [29]. The dataset contains 1,322,024 records, for all flights conducted by the 14 US carriers in the first three months of 2016. We use this dataset to study sampling for large-scale settings, and to showcase the application of our techniques for diversity.

Fairness models. We evaluate performance of our methods over two general fairness models, see $\S 2$.

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 [14, 17, 33], including also for ranked outputs [32] and for set selection [27]. 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.

FM2, proportional representation on multiple, possibly overlapping, type attributes, is a generalization of FM1 that can express proportionality constraints of [9]. As in [9], we bound the number of members of a group from above. For example, for COMPAS, we specify the maximum number of items among the topranked 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). In all experiments, 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 $\mathcal D$.

6.2 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 the COMPAS dataset with d=3 (scoring attributes start, 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 16 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.

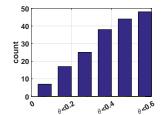
We observe that a satisfactory function f' was found close to the input function f in all cases. Specifically, note that $\theta(f,f')<0.6$ in all cases, and recall that $\theta\in[0,\pi/2]$, with lower values corresponding to higher similarity. (For a more intuitive measure: the value of $\theta=0.6$ 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).

In our next experiment, 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 <code>juv_other_count</code> (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 reoffend, 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 with the angle with the x-axis (juv_other_count) of at most 0.31.

Next, suppose that we use the same scoring attributes, but a

⁶https://docs.scipy.org/doc/scipy/reference/optimize.html



between input and output func- time, varying n tions

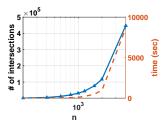


Figure 16: MD, angle distance Figure 17: 2D; preprocessing

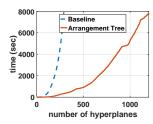


Figure 18: MD; arrangement construction cost, the advantage of using arrangement tree

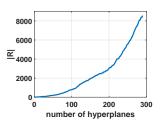


Figure 19: MD; arrangement complexity while adding the hyperplanes (d=3)

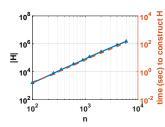
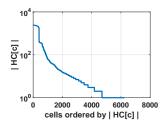
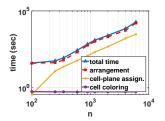


Figure 20: MD; effect of n on |H| Figure 21: MD; umber of hyper-(d = 3)



planes passing through each cell (n = 100, d = 4)



preprocessing times for different preprocessing times for different steps; d=3

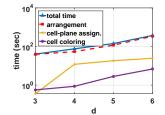


Figure 22: MD; effect of n on Figure 23: MD; effect of d on steps; n = 100

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. In fact, 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 between f and f' is always more than 0.99).

In our final validation experiment, 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.

Performance of query answering 6.3

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 two-dimensional and multidimensional 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% AA in the top-30%) and the scoring attributes in the same ordering provided in the description of COMPAS dataset.

2D. One nice property of 2DONLINE is that it does not need to access the raw data at query time. It only needs to apply binary search on the sorted 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, similarly to 2D, we took the average running time of 30 random queries, for between 3 and 6 scoring attributes (dimensions). Upon arrival of a query function f, MDON-LINE finds the cell to which f belongs in $O(\log N)$, and returns the corresponding satisfactory function f'. As a result, similar to 2DONLINE is significantly faster than even finding the ordering of the items based on f. This is confirmed in our experiments were the running time, in all cases, was less than 200 μsec whereas the time required to order the items based on f was 25 msec. Please note that the running time of MDONLINE is independent of n (the number of items in the dataset) and will perform similarly for the very large datasets.

6.4 Performance of preprocessing

In order to study the preprocessing performance, similar to § 6.3, we use COMPAS as the default dataset, the default fairness model (at most 60% African Americans at the top-ranking 30%), and the scoring attributes in the same ordering provided in the description of COMPAS dataset.

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 17 shows the experiment results for varying the number of items from 100 to 6,000. The xaxis shows the values of n (in log-scale), and the left and right y-axes show the number of ordering exchanges and the running time of 2DRAYSWEEP, respectively. Looking at the left 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 450k. This is because the pairs of items in which one dominates the other do not have a ordering exchange. Also, looking at the right y-axis, and comparing the dashed orange line (time) with the blue line (number of ordering exchanges), one can see that the orange line has a

sharper slope as it passes through the blue line. This is because the oracle is in O(n) and thus, based on Theorem 1, 2DRAYSWEEP is in $O(n^3)$.

MD, the effect of using arrangement tree. In \S 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 18 shows the incremental cost of adding hyperplanes to the arrangement when d=3. While the baseline (SATREGIONS) needed 8,000 seconds for adding the first 250 hyperplanes, using the arrangement tree helped save around 7,740 seconds. Fixing the budget to 8,000 seconds, 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 consequent hyperplanes (with SATREGIONS) is more expensive. This is presented in Figure 19, 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 \S 5 for preprocessing the data in partitioned angle space. First, similar to the 2D experiments, varying n from 200 to 6,000, in Figure 20 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 17 (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 increases. 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 for observing the benefit of using the arrangement tree, we discussed the effect of the number of hyperplanes on the complexity of the arrangement (quantified by the number of regions) and on the running time for constructing it. Thus, rather than constructing the arrangement for the complete set of hyperplanes, in § 5, we limit the arrangement construction for each cell to the hyperplanes passing through it. In Figure 21 we set the number of items to 100 and d to 4, and observe the number of hyperplanes passing through the cells. The x-axis in Figure 21 is the cells sorted by $|\mathcal{HC}[c]|$ (the number of hyperplanes passing through the cell c), and the y-axis shows $|\mathcal{HC}[c]|$ for each cell c. Looking at the figure, one can see that more than 5000, out of 6000 cells have less than 100 hyperplanes passing through them, and even constructing the complete arrangement inside them is not very expensive. We explained in § 5 that our goal is to associate a satisfactory function with each cell, allowing MARKCELL to stop early (before constructing the complete arrangement) once a satisfactory function is identified.

Figures 22 and 23 show the required time for different steps of preprocessing, as well as the total preprocessing time. Figure 22 shows the cost for varying n, with d=3 and N=40,000. In

Figure 23 we fix n = 100 and N = 40,000 and vary d. The yellow line in both figures shows the required time for identifying the hyperplanes passing through each cell. Applying CELLPLANEX 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 hyperplanes |H| is in $O(n^2)$. On the other hand, despite the complexity of the arrangement construction (c.f. Theorem 3), finding a satisfactory function for each cell that intersects with a satisfactory region (the dashed red line) may not be not very expensive and in certain cases has similar running time as CELLPLANE $_{\times}$. Different optimizations proposed in § 4 and 5 result in reasonable performance of this step. First, reducing 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 18, the arrangement tree data structure helps to rule out checking the intersection of the hyperplanes with all regions. Finally, the early stop condition is effective at reducing the running time. Still, looking at Figures 22 and 23 this step always takes the majority of the preprocessing time.

The final step is to use CELLCOLORING to assign the satisfactory function of the closest satisfactory cell to each unsatisfactory cell. Using a priority queue, this step is expected to be fast, which is observed in all the settings in Figures 22 and 23.

MD, sampling for a large-scale setting. We discussed in § 5.4 that preprocessing time can be reduced for very large datasets by conducting it over a uniform sample. 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 1,000 records uniformly at random from the dataset of 1.3M records and use it for preprocessing with N=40,000. Preprocessing took 1,276 seconds to complete. Next, we used the complete dataset and checked if the function assigned to the cells using the sample are in fact satisfactory. It turned out that 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.

7. RELATED WORK

Several recent papers focus on measuring fairness in ranked lists [31,32], on constructing ranked lists that meet fairness criteria [9], and on fair and diverse set selection [27]. Fairness in top-k over a single binary type attribute (such as gender, ethnic majority/minority, or disability status) is studied in Zehlike et al. [32], 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. [9] provide a theoretical investigation of ranking with fairness constraints. In their work, fairness in a ranked list is quantified as an upper bound on the number of items at the top-k that belong to multiple, possibly overlapping, types. In contrast, our goal is to assist the user in designing fair score-based rankers. Our framework accommodates a large class of fairness constraints. In our experiments, we focus on variants of fairness constraints similar to those in [9, 27, 32].

Diversification of query results has always been an important data retrieval topic [2, 8, 13]. Different definitions of diversity include similarity function-based [11] and topic-based [2]. General

background on diversity and a connection to fairness are provided in [13]. A nice property of the techniques proposed in this 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 provided in this paper mainly follow the concepts in combinatorial geometry. The general background and the terms are provided in [12,15]. In addition, [15] discussed the complexity bounds and proposes the incremental algorithm for constructing the lattice of arrangement. Arrangement of hyperplanes is also studies in [22,25,26]. Applications of arrangements such as motion planner in robotics are discussed by P. Agrawal et. al. [1].

8. FINAL REMARKS

In this paper, we studied the problem of designing fair ranking schemes. Considering the linear combinations of attribute values as the score of each item, our system assists users in choosing criterion weights that are fair. Creating proper indexes in an offline manner enables efficient answering of the users' queries. In addition to the theoretical analyses, empirical experiments on real datasets confirmed both efficiency and effectiveness of our proposal.

In this paper, we designed techniques for a general fairness definition that takes an ordering of the items as input and decides whether it meets the fairness requirements. Additional information about the fairness model can help optimize the techniques. For example, knowing that the fairness oracle investigates fairness only within the top-k of the ordering [32] can help in ignoring the items that do not belong to k convex layers [10], as those will never appear within the top-k. This reduces complexity of the arrangement from $n^{2(d-1)}$ to $n^{2(d-1)}_k$, where n_k is the number of items in the top k convex layers. We will explore this and other kinds of optimizations in future work. The techniques of this paper are provided for a fixed number of dimensions. We consider extending our techniques to a variable number of dimensions for future work.

9. REFERENCES

- [1] P. K. Agarwal and M. Sharir. Arrangements and their applications. *Handbook of computational geometry*, pages 49–119, 2000.
- [2] R. Agrawal, S. Gollapudi, A. Halverson, and S. Ieong. Diversifying search results. In *Proceedings of the second ACM international conference on web search and data mining*, pages 5–14. ACM, 2009.
- [3] J. Angwin, J. Larson, S. Mattu, and L. Kirchner. Machine bias: Risk assessments in criminal sentencing. *ProPublica*, May 23, 2016.
- [4] A. Asudeh, A. Nazi, N. Zhang, and G. Das. Efficient computation of regret-ratio minimizing set: A compact maxima representative. In SIGMOD, 2017.
- [5] A. Asudeh, S. Thirumuruganathan, N. Zhang, and G. Das. Discovering the skyline of web databases. *Proceedings of the VLDB Endowment*, 9(7):600–611, 2016.
- [6] A. Asudeh, N. Zhang, and G. Das. Query reranking as a service. *PVLDB*, 9(11):888–899, 2016.
- [7] S. Barocas and A. D. Selbst. Big data's disparate impact. *California Law Review*, 104, 2016.
- [8] B. Boyce. Beyond topicality: A two stage view of relevance and the retrieval process. *Information Processing & Management*, 18(3):105–109, 1982.
- [9] L. E. Celis, D. Straszak, and N. K. Vishnoi. Ranking with fairness constraints. *CoRR*, abs/1704.06840, 2017.

- [10] Y.-C. Chang, L. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The onion technique: indexing for linear optimization queries. In *ACM Sigmod Record*, volume 29, pages 391–402. ACM, 2000.
- [11] H. Chen and D. R. Karger. Less is more: probabilistic models for retrieving fewer relevant documents. In Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval, pages 429–436. ACM, 2006.
- [12] M. De Berg, O. Cheong, M. Van Kreveld, and M. Overmars. *Computational Geometry: Introduction*. Springer, 2008.
- [13] M. Drosou, H. Jagadish, E. Pitoura, and J. Stoyanovich. Diversity in Big Data: A review. *Big Data*, 5(2), 2017.
- [14] C. Dwork, M. Hardt, T. Pitassi, O. Reingold, and R. S. Zemel. Fairness through awareness. In *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, pages 214–226, 2012.
- [15] H. Edelsbrunner. Algorithms in combinatorial geometry, volume 10. Springer Science & Business Media, 2012.
- [16] D. Ensign, S. A. Friedler, S. Neville, C. E. Scheidegger, and S. Venkatasubramanian. Runaway feedback loops in predictive policing. *CoRR*, abs/1706.09847, 2017.
- [17] M. Feldman, S. A. Friedler, J. Moeller, C. Scheidegger, and S. Venkatasubramanian. Certifying and removing disparate impact. In *SIGKDD*, 2015.
- [18] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.
- [19] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal* of the ACM (JACM), 34(3):596–615, 1987.
- [20] S. A. Friedler, C. Scheidegger, and S. Venkatasubramanian. On the (im)possibility of fairness. *CoRR*, abs/1609.07236, 2016.
- [21] B. Friedman and H. Nissenbaum. Bias in computer systems. *ACM Trans. Inf. Syst.*, 14(3):330–347, 1996.
- [22] B. Grünbaum. Arrangements of hyperplanes. In *Convex Polytopes*, pages 432–454. Springer, 2003.
- [23] P. Jacobs. Legacy admissions policies were originally created to keep jewish students out of elite colleges. *Business Insider*, October 23, 2013. [Online; accessed 29-December-2017].
- [24] J. Karabel. The Chosen: The Hidden History of Admission and Exclusion at Harvard, Yale, and Princeton. Houghton Mifflin Company, 2005.
- [25] P. Orlik and H. Terao. Arrangements of hyperplanes, volume 300. Springer Science & Business Media, 2013.
- [26] V. V. Schechtman and A. N. Varchenko. Arrangements of hyperplanes and lie algebra homology. *Inventiones mathematicae*, 106(1):139–194, 1991.
- [27] J. Stoyanovich, K. Yang, and H. Jagadish. Online set selection with fairness and diversity constraints. In *EDBT*, 2018.
- [28] The College Board. SAT percentile ranks, 2014.
- [29] 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 29-December-2017].
- [30] A. Vlachou, C. Doulkeridis, and Y. Kotidis. Angle-based space partitioning for efficient parallel skyline computation.

In SIGMOD. ACM, 2008.

- [31] K. Yang and J. Stoyanovich. Measuring fairness in ranked outputs. In SSDBM, 2017.
- [32] M. Zehlike, F. Bonchi, C. Castillo, S. Hajian, M. Megahed, and R. A. Baeza-Yates. FA*IR: A fair top-k ranking algorithm. In CIKM, 2017.
- [33] I. Zliobaite. Measuring discrimination in algorithmic decision making. *Data Min. Knowl. Discov.*, 31(4):1060–1089, 2017.

APPENDIX

A. APPENDIX

A.1 Angle distance computation

As explained in \S 2, linear ranking function can be represented as rays in \mathbb{R}^d that start at the origin. These rays can be represented by d-1 angles. Consider a ray ρ that starts from the origin and passes through the point p. Let $polar(p) = \langle r, \Theta \rangle$ be the polar representation of p. First all the points p' that ρ passes through them have the polar representative $\langle r', \Theta \rangle$. Second, for a point p with the polar representative $\langle r, \Theta \rangle$, there is one and only one ray starting from the origin that passes through it. Thus, the angle vector Θ of size d-1 is enough for identifying this ray. We use cosine similarity to compute the angle distance between two rays, represented by the angle vectors $\Theta^{(i)}$ and $\Theta^{(j)}$.

Consider the point $p_i = \langle 1, \Theta^{(i)} \rangle$ (that the ray $\Theta^{(i)}$ passes through it). The cartesian coordinates of p_i are⁷:

$$p_i = \langle \sin \Theta_k^{(i)} \prod_{l=k+1}^{d-1} \cos \Theta_l^{(i)}, \forall 0 \le k < d \rangle$$
 (8)

Using the definition of cosine similarity, for the points $p_i = \langle 1, \Theta^{(i)} \rangle$ and $p_j = \langle 1, \Theta^{(j)} \rangle$:

$$\cos(\theta_{ij}) = \sum_{k=0}^{d-1} \sin \Theta_k^{(i)} \sin \Theta_k^{(j)} \prod_{l=k+1}^{d-1} (\cos \Theta_l^{(i)} \cos \Theta_l^{(j)})$$
 (9)

Thus, θ_{ij} (the angle between the rays $\Theta^{(i)}$ and $\Theta^{(j)}$) is:

$$\theta_{ij} = \arccos\left(\sum_{k=0}^{d-1} \sin\Theta_k^{(i)} \sin\Theta_k^{(j)} \prod_{l=k+1}^{d-1} (\cos\Theta_l^{(i)} \cos\Theta_l^{(j)})\right)$$
(10)

A.2 Angle space partitioning

According to Appendix A.1, the distance between two rays specified by two (d-1) dimensional angle vectors $\Theta^{(i)}$ and $\Theta^{(j)}$ is not the same as their euclidean distance. Thus, as also discussed in [30], a regular grid partitioning that equally partitions each axis into $\sqrt[d]{N}$ equal size ranges will not generate cells of equal sizes. One can verify this by looking at Figure 9, in which the cells in the bottom row have larger areas than the ones in the upper rows. Inspired by [30], we propose the angle space partitioning that partitions the space into N equal area cells. We do the partitioning using the surface of (the first quadrant of) the unit hypersphere in \mathbb{R}^d . Consider a hypercone starting from the origin, while its base is a hypercube (square in \mathbb{R}^3) on the surface of unit hypersphere. We want to partition the space into N such hypercones such that the

area of all cells are equal. The total area of the space (the area of the first quadrant of the unit hypersphere) is ⁸

$$\eta = \frac{\pi^{d/2}}{2^{d-1}\Gamma(d/2)} \tag{11}$$

where $\boldsymbol{\Gamma}$ is the gamma function. Thus, the area of each cell is

$$\eta_{\text{cell}} = \frac{\pi^{d/2}}{N2^{d-1}\Gamma(d/2)}$$
(12)

Considering the cells to be small enough, one can assume that the area of each cone on the surface of the hypersphere is equal to the area if its base. The area of a hypercube with sides of size γ is γ^{d-1} (e.g. γ^2 in \mathbb{R}^3). Assuming the area of the hypercone and its base to be equal, using Equation 12, the sides of the hyper square are of size:

$$^{d-1}\!\!\sqrt{\eta_{\text{cell}}} = \sqrt[d-1]{\frac{\pi^{d/2}}{N2^{d-1}\Gamma(d/2)}}$$
 (13)

Since the radius of the hypersphere is 1, the angle between the rays in two corners of a side are:

$$\gamma = 2\arcsin\frac{\sqrt[d-1]{\frac{\pi^{d/2}}{N2^{d-1}\Gamma(d/2)}}}{2}$$
 (14)

We use γ , as computed in Equation 14, for angle space partitioning, as follows.

Consider the axes $\theta_1, \theta_2, \dots, \theta_{d-1}$. For every axis, we maintain a vector of angles T_{θ_i} such that each element $T_{\theta_i}[j]$ of the vector contains:

- range: the borders of the row in axis θ_i .
- elements: the vector of angles for axis $T_{\theta_{i+1}}$ in row $T_{\theta_i}[j]$.

One can see the partitioning data structure as a tree of depth d-1 that its leaves are the cells; the path from the root to each leaf identifies its borders in every dimension. In order to construct the ranges, in an iterative manner, we apply Equation 10 to specify ranges of angle γ as the rows of each axis. Then, we recursively partition the rows of the axis into equal area cells. Algorithm 12 shows the pseudo code for angle space partitioning. Consider the moment where the algorithm is partitioning the i-th axis and the current point is in the form of $p_c = \langle \Theta_1, \Theta_2, \cdots, \Theta_{i-1}, \theta, 0, \cdots, 0 \rangle$. The objective is to find the next point in i-th axis such that the angle of its corresponding ray with the current point is γ . The next point is in the form of $p_n = \langle \Theta_1, \Theta_2, \cdots, \Theta_{i-1}, \theta', 0, \cdots, 0 \rangle$, where θ' is unknown. Using Equation 10, the angle between the rays of p_c and p_n can be rewritten as:

$$\cos \gamma = \cos \theta' \cos \theta \sum_{k=0}^{i-1} \sin^2 \Theta_k \prod_{l=k+1}^{i-1} (\cos^2 \Theta_l) + \sin \theta' \sin \theta$$
(15)

Let α be $\cos\theta \sum_{k=0}^{i-1} \sin^2\Theta_k \prod_{l=k+1}^{i-1} (\cos^2\Theta_l)$ and β be $\sin\theta$. Then the angle between the above equation is

$$\alpha\cos\theta'+\beta\sin\theta'=\cos\gamma$$
 Now, let us set $\delta=\arctan\frac{\beta}{\alpha}$ and $\Delta=\sqrt{\alpha^2+\beta^2}$. Thus,
$$\Delta\cos\delta\cos\theta'+\Delta\sin\delta\sin\theta'=\cos\gamma$$

$$\Rightarrow\Delta\cos(\theta'-\delta)=\cos\gamma$$

 $^{^7\}text{To}$ simplify the equation, we set $\Theta_0^{(i)}$ to $\pi/2,$ by appending it to the beginning of the vector $\Theta.$

 $[\]Rightarrow \Delta \cos(\theta' - \delta) = \cos \gamma$ $\Rightarrow \theta' = \arccos \frac{\cos \gamma}{\Lambda} + \delta$ (16)

 $^{^8} http://mathworld.wolfram.com/Hypersphere.html\\$

Algorithm 12 ANGLEPARTITIONING

Input: axis number i, angle combination for previous axes Θ , d **Output:** Partitioned space T

```
1: \theta = 0, T = \{\}, j = 1
2: while \theta < \pi/2 do
3: compute \theta', using Equation 16
4: T[j].range= (\theta, \theta')
5: if i < (d-1) then
6: \Theta[i] = \theta
7: T[j].elements=ANGLEPARTITIONING (\Theta, i+1, d)
8: end if
9: \theta = \theta', j = j + 1
10: end while
11: return leaves(T)
```

THEOREM 7. Algorithm 12 is in O(N).

PROOF. The total number of cells is N. Looking at the recursion tree, every leaf of the tree (every cell) has the level d. Therefore, for a constant value of d, the cost of generating each cell is constant. Therefore, Algorithm 12 is in O(N). \square