Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

N J I T
New Jersey's Science &
Technology University

The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.

ABSTRACT

DIVERSIFICATION AND FAIRNESS IN TOP-K RANKING ALGORITHMS

by
Mahsa Asadi

Given a user query, the typical user interfaces, such as search engines and recommender systems, only allow a small number of results to be returned to the user. Hence, figuring out what would be the top-k results is an important task in information retrieval, as it helps to ensure that the most relevant results are presented to the user. There exists an extensive body of research that studies how to score the records and return top-k to the user. Moreover, there exists an extensive set of criteria that researchers identify to present the user with top-k results, and result diversification is one of them. Diversifying the top-k result ensures that the returned result set is relevant as well as representative of the entire set of answers to the user query, and it is highly relevant in the context of search, recommendation, and data exploration. The goal of this dissertation is two-fold: the first goal is to focus on adapting existing popular diversification algorithms and studying how to expedite them without losing the accuracy of the answers. This work studies the scalability challenges of expediting the running time of existing diversification algorithms by designing a generic framework that produces the same results as the original algorithms, yet it is significantly faster in running time. This proposed approach handles scenarios where data change over a period of time and studies how to adapt the framework to accommodate data changes. The second aspect of the work studies how the existing top-k algorithms could lead to inequitable exposure of records that are equivalent qualitatively. This scenario is highly important for long-tail data where there exists a long tail of records that have similar utility, but the existing top-k algorithm only shows one of the top-ks, and the rest are never returned to the user.

Both of these problems are studied analytically, and their hardness is studied. The contributions of this dissertation lie in (a) formalizing principal problems and studying them analytically. (b) designing scalable algorithms with theoretical guarantees, and (c) evaluating the efficacy and scalability of the designed solutions by comparing them with the state-of-the-art solutions over large-scale datasets.

DIVERSIFICATION AND FAIRNESS IN TOP-K RANKING ALGORITHMS

by
Mahsa Asadi

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology and
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

August 2023

Copyright © 2023 by Mahsa Asadi

ALL RIGHTS RESERVED

APPROVAL PAGE

DIVERSIFICATION AND FAIRNESS IN TOP-K RANKING ALGORITHMS

Mahsa Asadi

Dr. Senjuti Basu Roy, Dissertation Advisor — Date
Associate Professor of Computer Science, NJIT

Dr. Zhi Wei, Committee Member — Date
Professor of Computer Science, NJIT

Dr. Yiannis Koutis, Committee Member — Date
Associate Professor of Computer Science, NJIT

Dr. Dimitrios Theodoratos, Committee Member — Date
Associate Professor of Computer Science, NJIT

Dr. Sihem Amer-Yahia, Committee Member — Date
Research Director at Centre National de Recherche Scientifique, Grenoble, France

BIOGRAPHICAL SKETCH

Author: Mahsa Asadi

Degree: Doctor of Philosophy

Date: August 2023

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2023
- Master of Engineering in Computer Software,
  Isfahan University of Technology, Isfahan, Iran, 2018
- Bachelor of Engineering in Computer Software,
  Iran University of Science and Technology, Tehran, Iran, 2015

Major: Computer Science

Presentations and Publications:

Md Mouinul Islam, Mahsa Asadi, Sihem Amer-Yahia, and Senjuti Basu Roy, "A generic framework for efficient computation of top-k diverse results" *The International Journal on Very Large Data Bases (VLDB 2022).*

Mahsa Asadi, Md Mouinul Islam, and Senjuti Basu Roy, "Making Top-k Algorithms Equitable for Long Tail Data" *Submitted to Very Large Data Bases Conference (VLDB 2023).*

To my beloved sister Sarina, whose love and memory will always remain in my heart.

ACKNOWLEDGMENTS

First of all, I would like to gratitude my sincerely appreciate to my advisor, Dr. Senjuti Basu Roy, for her professional and instructively guidance, for her patience, motivation, and support. I could not have imagined having a better advisor and mentor for my Ph.D. study.

I would like to thank the rest of my dissertation committee: Drs. Sihem Amer-Yahia, Zhi wei, Yiannis Koutis, and Dimitri Theodoratos. I am truly honored to have them as my committee members. Each of the members of my Dissertation Committee has provided me extensive personal and professional guidance and taught me a great deal about both scientific research and life in general.

I would like to extend my sincere thanks to the Department of Computer Science and to the National Science Foundation. It would be impossible for me to complete my Ph.D. degree without their generous support.

I would also like to thank all my collaborators who are part of this dissertation: Dr. Sihem Amer-Yahia and Mr. Md Mouinul Islam. I also wish to thank the Big Data Analytics Lab (BDAL) and my lab members, Dr. Dong Wei, Dr. Mohammadreza Esfandiari (Payam), Mrs. Sepideh Nikookar, Mr. Md Mouinul Islam, Mr. Md Rakibul Hasan, and Mr. Sohrab Namazinia for their support, help, and research collaboration. From the bottom of my heart I would like to say big thank you for all the friends and peers I have met at NJIT, specially my friend Dr. Raina Samuel for the thoughts we exchanged, the discussions, and the good days we had.

Last but not least, I would like to express my deepest love to my family who have always been encouraging and supporting me to achieve my dream. My husband Hossein, my beloved parents, Fereshteh, and Mohammad, my sisters Roza, Roksana and my brother Amirhossein and my beloved younger sister Sarina. Also I would like

to thank my dearest old friends: Mehrnaz, Zahra, and Setareh for always being there for me.

TABLE OF CONTENTS

Chapter **Page**

1 INTRODUCTION 1
1.1 Overview 1
1.1.1 Background and Motivations 3
1.1.2 Contributions 4
2 RELATED WORK 7
2.1 Diversification in Top-k 7
2.1.1 Content based algorithms 7
2.1.2 Comparison with existing indexes 10
2.2 Fairness in Top-k 12
2.2.1 Group fairness 12
2.2.2 Individual fairness 13
2.2.3 Top-k algorithms 14
3 ACCESS PRIMITIVE FOR TOP-K DIVERSITY COMPUTATION 15
3.1 Introduction 15
3.2 Background and Approach 19
3.2.1 Motivating example and problem definition 19
3.2.2 Approach 21
3.3 MMR Query Processing with **DivGetBatch()** 25
3.3.1 MMR Algorithm 25
3.3.2 **Aug-MMR** Algorithm 26
3.4 GMM Query Processing with **DivGetBatch()** 34
3.4.1 GMM Algorithm 35
3.4.2 **Aug-GMM** Algorithm 35
3.5 $SWAP$ Query Processing with **DivGetBatch()** 39
3.5.1 $SWAP$ Algorithm 40

TABLE OF CONTENTS
(Continued)

Chapter **Page**

3.5.2 **Aug-SWAP** Algorithm 41

3.6 **I-tree** 49

3.6.1 Index construction 50

3.6.2 Index maintenance 52

3.7 Experimental Evaluation 54

3.7.1 Baselines 57

3.7.2 Summary of results 61

3.7.3 Quality analysis 63

3.7.4 Scalability analysis 64

3.8 Conclusion 73

4 TOP-K DIVERSIFICATION CONSIDERING FAIRNESS 74

4.1 Introduction 74

4.1.1 Demographic parity 75

4.1.2 Equalized odds 75

4.1.3 Unawareness 76

4.1.4 Individual fairness 76

4.1.5 Counterfactual fairness 77

4.1.6 Proportionate fairness 77

4.2 Data Model and Problem Definition 80

4.2.1 Running example 80

4.2.2 Data model 81

4.2.3 Problem definition and hardness 83

4.3 Exact Algorithms 86

4.3.1 Algorithm for **θ-Equiv-top-k-Sets** 86

4.3.2 Algorithm for **MaxMinFair** 95

4.4 Approximation Algorithms 96

TABLE OF CONTENTS
(Continued)

Chapter **Page**

4.4.1 Algorithm **RWalkTop-k-θ** 97

4.4.2 Algorithm **ARWalkTop-k-θ** 100

4.5 Experimental Evaluations 101

4.5.1 Goal 1: Fairness inside LambdaRank 105

4.5.2 Goal 2: MMSP complements Group-fairness 105

4.5.3 Algorithms for **θ-Equiv-top-k-Sets** 106

4.5.4 Algorithms for **MaxMinFair** 111

4.5.5 Summary of results 112

4.6 Conclusion 112

5 SUMMARY AND FUTURE WORK 114

5.1 Summary 114

5.2 Open and ongoing problems 115

REFERENCES 122

LIST OF TABLES

Table **Page**

3.1 Technical Results for Running Time Analysis w.r.t. $|CandR|$ 18

3.2 Technical Results for Running Time Analysis w.r.t. C, m, l 19

3.3 Notations & Interpretations 23

3.4 Similarity Matrix for Records 26

3.5 First Two Iterations of DivGetBatch() in Aug-MMR 29

3.6 Dataset Statistics 55

3.7 Aug-MMR vs MMR Running Time (s) on **MakeBlobs** with $l = 2$, $m = 6$ 56

3.8 $|CandR|$ Percentage Returned by DivGetBatch() on MovieLens 65

3.9 $|CandR|$ Percentage Returned by DivGetBatch() Using Different Index Structures for Aug-MMR on MakeBlobs 65

3.10 Pruning Percentage by DivGetBatch() Using Different Index Structures for Aug-MMR on MakeBlobs 66

3.11 Number of Access Percentage for Aug-MMR and SPP on MakeBlobs . . 66

3.12 Index Comparisons 67

3.13 **Aug-MMR** vs MMR Running Time on MakeBlobs 100k Records . . . 67

3.14 Aug-MMR vs MMR on Movielens Non-metric Data 67

3.15 I-tree Maintenance on MakeBlobs 10k Records 68

3.16 I-tree Maintenance Algorithm GrMn vs Construction from Scratch Algorithm NonIncrMn Running Time on MakeBlobs 10k Records 70

4.1 Table of Notations 85

4.2 Records with Sorted Relevance (Example 4.2.1) 85

4.3 Sorted Diversity List Based on Example 4.2.1 85

4.4 WRMSD Scores of All Set of Sets, Each with Three Movies 85

4.5 Dataset Statistics 101

LIST OF FIGURES

Figure **Page**

3.1 Proposed computational framework. 22
3.2 **I-tree**. 52
3.3 Aug-MMR vs MMR scalability. 54
3.4 Aug-MMR vs MMR varying parameters. 55
3.5 Aug-GMM vs GMM scalability. 57
3.6 **Aug-GMM** vs *GMM* performance varying parameters. 58
3.7 Aug-SWAP vs SWAP scalability. 59
3.8 Aug-SWAP vs SWAP varying parameters. 60
3.9 I-tree construction time. 62
3.10 I-tree maintenance time varying $|Y|$. 63
3.11 Index Construction and Query Processing time for tree baselines and **I-tree**. 64
3.12 Aug-MMR vs MMR running time on UCI Gas data. 69
4.1 Viewership distribution of top-1000 IMDB movies. 79
4.2 A complete lattice based on Example 4.2.1. 93
4.3 θ-Equiv-top-k-MMSP inside LambdaRank *Read source: [20].* 101
4.4 θ-Equiv-top-k-MMSP complements group-fairness. 102
4.5 Recall of RWalkTop-k-θ varying θ. 103
4.6 Recall of ARWalkTop-k-θ varying θ. 103
4.7 Record pruning percentage OptTop-k-θ. 103
4.8 RWalkTop-k-θ vs ARWalkTop-k-θ vs OptTop-k-θ scalability by varying dataset size N. 106
4.9 RWalkTop-k-θ vs ARWalkTop-k-θ vs OptTop-k-θ scalability by varying k. 107
4.10 RWalkTop-k-θ vs ARWalkTop-k-θ vs OptTop-k-θ scalability by varying θ. 108
4.11 **RWalkTop-k-**θ scalability for different utility functions. 109
4.12 MaxMinFair approximation factor and scalability. 110

CHAPTER 1

INTRODUCTION

1.1 Overview

Given a user query, the underlying process typically retrieves a large set of candidate results that could potentially be relevant to the query. However, the traditional user interfaces can only accommodate a small number of results and not all of those that the underlying engine returns. There is a need to present the user with a smaller set of results, which are known as top-k results [72]. To do this, the system typically scores each candidate based on various criteria such as relevance, diversity, newness, serendipity, etc. The scores are then used to rank the candidates and select the top-k with the highest scores [72]. A substantial amount of research has been conducted to explore the modification and creation of new ranking functions that allow for the combination of multiple criteria in order to provide results to the user [1, 2, 4, 21, 22, 33, 59, 71, 79, 81, 82]. Among these criteria is diversification, which is a significant area of interest that we seek to tackle.

Result diversification in ranking is an important way of deciding the appropriate top-k to return to the user. Result diversification ensures that the returned top-k results are relevant to the user query and represent the entire set of answers that could be returned to the user. Although this existing research has been studying more about designing effective ranking functions, there does not seem to be much work studying computational frameworks which can expedite query processing and answering faster while not changing the results of the original answer. One of our major contributions in this dissertation is to study this problem. In other words, We investigate how one can design a generic framework that can make the process of these different ranking functions, especially those that involve diversification from the computational

standpoint, further efficient without compromising the quality of their answers. Our first studied problem is to propose an access primitive and integrate it inside popular diversity algorithms to save running time while preserving the same output result. Instead of comparing the diversity scores of each pair of records separately (which could be computationally expensive for large sets), the approach is to compute an aggregate diversity score for a group of records and compare those scores pairwise. By using aggregate diversity scores, the number of pairwise comparisons needed to compute diversity scores for a set of records can be reduced, which can improve the algorithm's efficiency.

We also realize the existing top-k algorithms have a severe shortcoming while dealing with long tail data. In other words, the data contains many records which have similar utility. The existing algorithms are static, and they only return a fixed top-k to the user query, making the process inequitable to the records with similar utility and they never get the chance to be returned to the user [11, 19, 31]. This leads to inequitable exposure and brings in the second dimension of our studying of the process that we investigate, which we refer to as the process of equitable selection of the records. Our second studied problem is to argue that given a query, there are multiple top-k sets that are equivalent in utility, and we create a probability distribution over those sets. Hence, instead of creating one top-k set as a result, we are creating multiple top-k sets and giving a chance to other equivalent records having similar utility to appear in the result set rather than only giving the opportunity to the fixed specific records. The goal is to ensure that the selection probabilities of the records in these sets are as uniform as possible, thereby promoting fairness and reducing the potential for bias or discrimination. This proposed notion is rooted in the maxmin fairness theory that maximizes the minimum fairness [41].

1.1.1 Background and Motivations

The diversification algorithms can be either interchangeable or incremental greedy algorithms. Interchangeable algorithms first select k relevant records and then exchange selected records with remaining records to increase the overall diversity. An example is the $SWAP$ algorithm [95]. Incremental greedy algorithms iteratively build the top-k set by selecting the best record at each round. Some of the representative examples are Maximal Marginal Relevance, or MMR [23], Greedy Max-Min or GMM [44], Max-Sum [43] etc. Although these algorithms are incremental greedy, they are different in their objective function. For example, MMR computes an objective score based on two parameters: relevance to the query and diversity with other records. The algorithm repeats this computation k times to produce top-k. GMM finds the k most diverse records by selecting the maximum of minimum distances between undiscovered records and previously selected ones at each iteration, while $Max-Sum$ finds the k most diverse records by selecting the maximum of the sum of the distances between undiscovered records and previously selected ones at each iteration.

The original implementation of these representative algorithms, such as GMM, MMR, and SWAP that are iterative in nature, do not make any assumptions on the nature of the diversity functions. Indeed these algorithms decide to update the top-k set by making a greedy choice based on the current top-k set and the remaining records that are not yet in the top-k. These representative algorithms go through the time-consuming step of pairwise diversity computation of records between and across these two sets even to make a single update in the top-k set. Indeed, for a large database, this repetitive computation is expensive. Hence, the gap between our work and previous works is to reduce that computation without making any explicit assumptions about the diversity function, that is, considering diversity functions to be fully arbitrary or even non-metric. By using aggregate diversity scores, the number of pairwise comparisons needed to compute diversity scores for a set of records

can be reduced significantly, improving the algorithm's efficiency and making them faster. Moreover, unlike other existing baselines, our index can be used in non-metric functions, considers the records atomic, and has 90% pruning of the original dataset and searching through only a small fraction of the original dataset.

Our research also focuses on the issue of unequal exposure of datasets that exhibit a long-tail phenomenon, where only a few popular records receive significant exposure and are always returned in response to user queries. However, there are many other niche records that may be just as useful but are not shown as results. This can lead to the rich-gets-richer phenomenon, where those who already have advantages are more likely to gain even more over time. We use the example of the 1000 top-rated movies on IMDB [53], which follow a long-tail distribution regarding the number of views they receive. There is a long tail of movies with almost equal IMDB ratings, but they have a lower average number of user votes, making them get less exposure and remain uncovered. This unequal exposure can occur in various domains, including economics, social networks, and access to information [88].

The gap between our second work and previous existing algorithms is most of the previous work considers a fixed top-k as a result set [19,35,62,67], while we argue that it may be beneficial to generate all (or many) top-k sets that are equivalent in utility and create a probability distribution over those sets. We state that many top-k sets have a utility score close to each other, but they never get a chance to be presented to the users.

1.1.2 Contributions

Our proposed research could be broadly categorized in two parts:

- First, we redesign three existing popular diversification solutions, namely MMR [23], GMM [44], and SWAP [95]. We propose a framework consisting of a unified indexed data structure and build an index offline where it assumes the records to be atomic and the diversity function to be arbitrary. We partition the original

data into a height-balanced tree data structure and create indexes given to the augmented algorithms as the input instead of the original data. Then, we design an access primitive over the proposed indexing structure in the online phase. It is being used inside our proposed algorithms, leading to the fast computation of identical top-k diversified results. We provide theoretical proofs for the quality of the results and computation cost analysis, which verify the efficiency of our algorithms. This primitive is guaranteed to produce identical top-k results as of the original diversity algorithms. The fundamental idea is to make the comparison go over a group of records, as opposed to record pairs, thereby accelerating the computation. We also evaluate the performance of our algorithms compared to the original ones over large-scale datasets, which guarantees the efficiency of our framework. Unlike other existing baselines, our index can be used in non-metric functions and outperforms with 90% pruning of the original dataset. We use large real-world datasets, one large publicly available synthetic dataset to show that the augmented algorithms return results identical to their originals, while ensuring between a 3× to 24× speedup on large datasets. We study the effects of different parameters empirically and provide guidance for appropriate design choice. We empirically present exhaustive results to pre-process and maintain I-tree. Our empirical results corroborate our theoretical analyses. Considering that most of the works that studied the diversity in recommendation and ranking are over static data, and there are few works that studied diversity over dynamic data, we study data management challenges of maintaining our framework and data structure over "dynamic data" and design operations such as batch and individual insertions, deletions, and updates into our data structure. It is worth mentioning that considering the huge volume of the data being indexed into our framework, it is not practically efficient to redo the creation of our data structure over and over again once new data is inserted into our framework. Thus, it is needed to efficiently design insertion and deletion operations into our framework. We formally define the Batch-Insertion problem and solve it using a linear programming model. We provide experimental evaluations of how effective our procedure is.

- Second, we study how inequitable exposure can propagate in existing top-k algorithms and how to circumvent those challenges. We formalize our inequitable exposure problem in two steps: 1) generating equivalent top-k sets and 2) computing probability distribution over these equivalent top-k sets. We present an exact solution for producing equivalent top-k sets. We propose an item lattice data structure that allows efficient computation of the possible size-k sets and incremental updates of their score bounds by reusing previously calculated scores. Since the possible size-k set of sets over N records could be represented as a hierarchically ordered lattice containing $\binom{N}{k}$ nodes, we produce some of these nodes on the go, instead of discovering them from scratch one by one. Hence, we design two approximate algorithms, namely random walk and adaptive random walk, which are highly scalable. Then we evaluate the efficiency of our algorithms by doing experimental analysis.

- Finally, we describe a set of ongoing and open problems and present an overview of future work.

CHAPTER 2

RELATED WORK

2.1 Diversification in Top-k

Result diversification remains to be an active research topic with extensive applications in recommendation and search [1,2,4,21,33,61,65,71,72,78–80,82,83]. Instead of simply presenting the k most relevant or popular items, result diversification aims to provide a variety of items that cover different aspects, preferences, or dimensions of the search query or user's interests.

2.1.1 Content based algorithms

Content-based algorithms, which are our primary focus here, are of two kinds: *Interchange algorithms* first select k relevant records and then exchange selected records with remaining records to increase the overall diversity (*SWAP* [95] is an example). *Incremental greedy algorithms* iteratively build the top-k set by selecting the best record at each round. Notable examples of this latter kind are Maximal Marginal Relevance (*MMR*) method [23], Greedy Max-Min (*GMM*) [44], Max-Sum [43], which objective is to maximize the sum of the relevance and dissimilarity of the selected set, IA-SELECT [5], which maximizes the probability that the average user will find some useful information among the search results , and dLSH [1] which uses GMM to make LSH diversity aware.

SWAP [95] is a greedy algorithm that produces top-k results based on a given query Q and a tunable parameter that controls how much relevance could at most drop between any two records in the top-k results. The algorithm starts by sorting the records w.r.t. relevance and initializing the top-k result set S with the k-records with the highest relevance score with Q. It finds a *candidate record* from the current top-k set that has the smallest diversity contribution based on Equation (3.15). Indeed, in

each iteration, it attempts to swap one record from R \ S with the candidate record. It starts scanning the remaining sorted relevance list from the top. In every iteration, it attempts to swap one record from the current top-k set with another from sorted R if the latter record has a higher contribution to diversity while ensuring the threshold of relevance drop. The algorithm terminates when the relevance drop is below the threshold, or R is fully scanned.

$$Divcont(r_i, S) = \sum_{r_j \in S} Div(r_i, r_j). \tag{2.1}$$

Maximal Marginal Relevance (MMR**)** algorithm is a seminal work on result diversification [23]. MMR is based on Marginal Relvance (MR) score (Equation 3.1) that it maximizes in each iteration. Given a query, MR introduces a λ coefficient to strike a balance between the relevance score, computed between the records and the query, and the diversity score, computed between the records themselves.

MMR is greedy in nature that grows the size of the top-k set by adding records one by one in the top-k set by considering the relevance of the record and diversity with the previously selected records, using the formula below:

$$MMR(r) \leftarrow argmax_{r \in R \setminus S} MR(r),$$

$$MR(r) \leftarrow \lambda sim(r, Q) - (1 - \lambda) max_{r_j \in S} sim(r, r_j), \tag{2.2}$$

where Q is the query, S is the previously selected items, R is the remaining records in the dataset, r is a candidate record from R, and r_j is another record from S. λ is a tunable parameter. The time-consuming part of the algorithm lies in computing the MR score for each $r \in \{R \setminus S\}$ and returning the one with the highest MR score.

The MMR algorithm takes $\mathcal{O}(|R| \times |S|)$, when we add one new record to set S, demonstrating that it has an order of $N \times k$. The algorithm repeats k times and produces top-k results.

GMM [44] tries to find a subset of k most diverse records among N records by maximizing the minimum pairwise distance. *GMM* does not require any external query. Based on the original design, the first two records in the result set S are provided in constant time by an *oracle*. Then, the algorithm iteratively goes through all records in R and finds a record whose minimum *diversity* (maximum similarity) with the previously selected records is the largest (smallest). It continues until $|S|=k$. The objective function is:

$$GMM(r) \leftarrow argmax_{r \in R \setminus S} min_{r_j \in S} Div(r, r_j), \tag{2.3}$$

where $Div(r, r_j)$ is the diversity score between record r and r_j.

SPP [38] is a bounded diversification algorithm that produces same result as MMR while minimizing the number of accessed records. In [29], Drosou et al. introduce both greedy and interchange algorithms for the diversity over continuous data. Drosou and Pitoura [30] propose greedy algorithms for considering diversity over dynamic data by presenting *Insert* and *Delete* operations over the cover tree indexing structure. They also exploit the *GMM* algorithm for returning diversified top-k results. Drosou et al. [28] propose greedy algorithms for diversity over a representative subset of objects, *DisC*, which is a mapping of the original data. They also present a degree of diversification, radius r, instead of k size results. Their proposed algorithms exploit the *M-tree* [25] indexing structure.

From a different perspective, one can categorize diversification algorithms into three groups: record-level, feature-level, and category-level. In record-level algorithms (*MMR*, *GMM*, and *SWAP*), the input is the distance value between records regardless

of which record feature is more important. The score value is calculated based on an objective function that calculates distances/diversity. The inputs of feature-level algorithms are record features. Examples are DivGen and GenFilt [7]. The feature with the highest score is obtained from all records based on a ranking, and the goal is to skip some features and prune records having low scoring features. In the category-level algorithms, records are grouped into multiple categories. Such algorithms apply some constraints that will return no more than one or two records from the same category [3, 97].

2.1.2 Comparison with existing indexes

Compared to our proposed **I-tree**, existing indexing techniques are vector space based methods where coordinate information of the records are used to create data structures to answer a large spectrum of distance queries, where distance may be based on Euclidean, cosine similarity, general L_p norms, and so on. Popular solutions in low to moderate dimensional space include *K-B-D-tree* [74], *kd-tree* [16], *R-tree* [46], *R^*-tree* [15], *SS-tree* [85] or more recent *X-tree* [17], *UB-tree* [14], *SR-tree* [55]. All these methods use the domain object feature vectors to measure the distance between objects and create a similarity index. As opposed to that, we consider the records to be atomic (and not a collection of vectors), and the diversity function could be metric or not. Therefore, these methods do not extend to solve our problem.

There exists other popular tree data structures like *Cover-tree* [18], *Ball-tree* [58] and *M-tree* [25] that are used for nearest neighbor search. Unlike our **I-tree**, these trees can only be used for metric distance functions.

KD-tree [16]:KD-tree is a multidimensional Binary Search Tree. The tree is created by bisecting each dimension and finding the median. *KD-tree* can perform searches in multidimensional space for efficient nearest neighbor search.

Ball-tree [58]: Ball-tree is a binary tree in which every node defines a D-dimensional hypersphere or ball, containing a subset of the points to be searched. Each node in the tree defines the smallest ball that contains all data points in its subtree. This gives rise to the useful property that for a given test point t outside the ball, the distance to any point in a ball B in the tree is greater than or equal to the distance from t to the surface of the ball. *Ball-tree* only supports binary splits.

M-Tree [25]: M-tree is similar to *Ball-tree*, but supports multiple splits. Every node n and leaf lf residing in a particular node N is at most distance r from N, and every node n and leaf lf with node parent N keeps the distance from it. It also has the similar property of *Ball-tree*, which is for a given test point t outside the node, the distance to any point in a node in the tree is greater than or equal to the distance from t to the surface of the node.

Cover-Tree [18]: The tree is a series of levels arranged in a hierarchical order, where the highest level includes the root point and the lowest level includes all the points in the metric space. Each level, denoted by C, corresponds to a specific integer value i that decreases as one moves down the tree. The cover tree has three significant characteristics at every level C. (a) Nesting: $C_i \subset C_{i-1}$. This implies that once a point p appears in C_i, then every lower level in the tree has a node associated with p. (b) Covering tree: for every $p \in C_{i-1}$, there exists a $q \in C_i$ that $d(p, q) < 2^i$, and the node in level i associated with q is a parent of the node in level $i - 1$ associated with p. (c) Separation: for all distinct $p, q \in C_i$, $d(p, q) > 2^i$. Hence, we have fewer records at the higher levels and more records as we go down. The records that are part of the same node have a certain distance satisfied. The records that are part of the same node are actually farther from each other than the records that are apart from each other in two consecutive levels.

In summary, we present an access primitive **DivGetBatch()** *that leverages a precomputed data structure* **I-tree** *to integrate MMR, GMM, and SWAP to expedite*

their processing time. The design of our primitive is independent of features and categories and is applicable with any distance measure, making it generic and useful. We study MMR, GMM, and SWAP, since we believe these are notable choices in the existing diversity literature space, and many more recent works adapt these algorithms [1, 12, 28–30, 50, 66, 73, 86, 87, 93].

2.2 Fairness in Top-k

2.2.1 Group fairness

Most approaches to algorithmic fairness interpret fairness as lack of discrimination [39] seeking *that an algorithm should not discriminate against its input entities based on attributes that are not relevant to the task at hand*. Such attributes are called protected, or sensitive, and often include among others gender, religion, age, sexual orientation and race. So far, most work on defining, detecting and removing unfairness has focused on classification algorithms [96, 99] used in decision making. This is the notion of *group fairness*, which has been a major focus on many recent works [10, 36, 42, 52, 60, 70, 76, 84, 91, 98, 100]. A good survey on this topic focusing on search and recommendation applications can be found in [69]. We can explain Group fairness in top-k in the form of constraints on the fraction of records from some protected groups that should be included in the top-k set for any relevant k. It ensures that the proportion of protected candidates in the top-k set is proportionate to the original data distribution. Elisa Celis in [24] expresses fairness requirements by specifying an upper bound and a lower bound on the number of items with attribute A that are allowed to appear in the top-k positions of the ranking. Baruah [13] presents a new notion of group fairness named proportionate fairness or P-fairness, which is proportionate representation of every group based on a protected attribute in every position of the ranked top-k. For example If gender is the protected attribute with 50% representation of male and female,then p-fairness implies 1 male and 1

female in the top-2 items. Stoyanovich et al. [91] propose comparing the distributions of protected and non-protected candidates (for instance, using KL-divergence) on different prefixes of the list (e.g., top-10,top-20,top-30) and then averaging these differences in a discounted manner. The discount used is logarithmic, similarly to Normalized Discounted Cumulative Gain (NDCG, a popular measure used in Information Retrieval).

2.2.2 Individual fairness

Individual fairness, on the other hand, as proposed by Dwork et al [31], intends to ensure "similar individuals are treated similarly". Dwork et al. explain that a classifier is individually fair if the distance between probability distributions mapped by the classifier is not greater than the actual distance between the records [31]. Biega et al. propose measures that identify unfairness at the level of individual subjects considering position bias in ranking [19]. Mahabadi et al. study the individual fairness in k-clustering. Their goal is to develop a clustering algorithm of the records so that all records are treated (approximately) equally [62]. Patro et al. [67] investigate the fair allocation problem and study individual fairness in two-sided platforms consisting of producers and customers on opposite sides. Fish et al. study individual fairness in social network [35] to maximize the minimum probability of receiving the information for poorly connected users. *It has been recognized that group fairness alone has its deficiencies* [37]. In two independent efforts, Flanigan et. al. [36] and Garcia-Soriano et. al. [41] study how to enable equitable selection probability of the records under group fairness constraints and propose maxmin-fair distributions of ranking. Zemel et al. develop a learning algorithm for fair classification that ensures both group fairness and individual fairness [99]. [11] studies individual fairness in similarity search to ensure points within distance r from the given query have the same probability to be returned.

2.2.3 Top-k algorithms

Given a user query, a top-k result contains k records that have the highest scores [72]. Scores are computed based on relevance, diversity, newness, serendipity, etc. Designing effective scoring functions as well as efficient algorithms [1, 2] lend to numerous applications in recommendation and search [4, 21, 22, 33, 59, 81, 83] and is an active area of research.

*θ-**Equiv-top-k-MMSP** is motivated by recent existing works [11, 36, 41], yet it is unique - we study existing top-k algorithms and redesign them to address a fairness concern that is prevalent in long tail data.*

CHAPTER 3

ACCESS PRIMITIVE FOR TOP-K DIVERSITY COMPUTATION

3.1 Introduction

Diversity has a wide variety of applications in search, recommendation [1,2,33,71,79,82] and data exploration. The goal of diversification algorithms is to return results that are relevant as well as cover user intent. In the data management community, returning top-k diverse results of a query has been extensively studied, and there exists many seminal works [23,44,95] that propose objective functions and efficient algorithms to achieve a trade-off between relevance and diversity.

The original implementation of many representative algorithms, such as, GMM [44], MMR [44], SWAP [95] that do not make any assumptions on the nature of the diversity functions are iterative in nature and make the decision of updating the top-k set by making a greedy choice based on the current top-k set and the remaining records that are not yet in top-k. These representative algorithms go through the cumbersome step of pairwise diversity computation of records between and across these two sets even to make a single update in the top-k set. Indeed, for a large database containing N records, this repetitive computation is expensive $\mathcal{O}(N)$, since typically $k << N$. We are also aware of a handful of existing works [38,64] that are specifically designed on geometric space and avoid this repetitive computation. However, to the best of our knowledge, most of the existing works assume this expensive computation to be necessary, when diversity is designed for arbitrary non-metric functions or even studied in general metric space. Contrarily, our effort here is to reduce that computation without making any explicit assumptions about the diversity function, that is, considering diversity functions to be fully arbitrary or even non-metric.

Our first contribution lies in identifying one major computational bottleneck in existing popular diversification algorithms and how to accelerate that process. We identify the basic ingredients of developing **DivGetBatch()** as an access primitive such that it remains agnostic to any specific underlying diversity or distance computation function. This primitive is also guaranteed to produce identical top-k results as of the original diversity algorithms. The fundamental idea is to make the comparison go over a group of records, as opposed to record pairs, thereby accelerating the computation. In other words, the large number of N records are to be grouped in a small number of C nodes and some higher level diversity aggregates are to be maintained between the nodes. Towards that, we develop a generic computation framework that builds an index **I-tree** offline and maintains two other auxiliary data-structures (*MinsimMatrixNode* and *MaxsimMatrixNode*) that are highly generic in nature and suitable to handle updates. Indeed, the design of **I-tree** is rather simple and may appear to share resemblance with existing indexing techniques (Section 3.7 contains detailed discussion and empirical evaluation towards that). Our primary contribution lies in proposing a simple enough indexing technique that could be easily designed using off-the-shelf popular record partitioning algorithms, such as, K-Means [47], but study how to make it generic enough to work on a variety of diversification algorithms over arbitrary diversification functions. In fact, existing popular indexing techniques, such as *K-B-D-tree* [74], *kd-tree* [16], M-Tree [25], Ball-Tree [58], *R-tree* [46] assume that coordinate information of the records are available and used to create data structures to answer a large spectrum of distance queries, where distance may be based on Euclidean, cosine similarity, or general L_p norms. ***However,* I-tree *assumes the records to be atomic and the diversity function to be arbitrary.***

Our second contribution is to develop query processing algorithms for *MMR*, *GMM*, and *SWAP* [23, 44, 95] using **DivGetBatch() (Sections 3.3, 3.4, 3.5)**.

Fundamentally, we have redesigned the original algorithms to run over pairs of groups of records as opposed to pairs of records to save up processing time. **We make theoretical claims and proofs on the exactness and the running time of the augmented algorithms in expectation (assuming uniform data and query distributions) and in the worst case**. As an example, we prove that augmented $SWAP$ (**Aug-SWAP**) takes $\mathcal{O}(N/C * k * \log k + N)$ time in expectation compared to $\mathcal{O}(N * k * \log k)$ time of the original algorithm. It is easy to notice that augmented $SWAP$ is guaranteed to run faster than the original algorithm, as $Max(N/C * k * \log k, N)$ (C is the number of groups) is smaller than $N * k * \log k$. The summary of the complexity results are presented in Tables 3.1 and 3.2.

Our third contribution is developing principled solutions for creating and maintaining **I-tree (Section 3.6)**. **I-tree** is a complete m-ary tree [26] with height l. There exists many ways to build **I-tree** (e.g., hierarchical graph partitioning or clustering could be used). We identify that the main computational bottleneck of **I-tree** under batch updates lies in updating *MinsimMatrixNode* and *MaxsimMatrixNode*. Therefore, we formalize the index maintenance problem as an optimization problem, with the goal of minimizing the number of updates in these data structures. We present an integer programming-based exact solution **OPTMn** for that, and a greedy heuristic **GrMn** that is highly scalable in nature.

Our final contribution is experimental **(Section 3.7)**. We use large real-world datasets, one large publicly available synthetic dataset to show that the augmented algorithms return results identical to their originals, while ensuring between a 3× to 24× speedup on large datasets. We study the effects of different parameters empirically and provide guidance for appropriate design choice. We empirically present exhaustive results to pre-process and maintain **I-tree**. Our empirical results corroborate our theoretical analyses.

Table 3.1 Technical Results for Running Time Analysis w.r.t. $|CandR|$

Algorithm	**Variant**	**Expected time w.r.t $|CandR|$**
MMR	Original	$\mathcal{O}(N * k^2)$
	Augmented	$\mathcal{O}(C * k^2 + N + \sum_{i=1}^{k} |CandR_i| * k)$
GMM	Original	$\mathcal{O}(N * k)$
	Augmented	$\mathcal{O}(C * k + \sum_{i=1}^{k} |CandR_i|)$
$SWAP$	Original	$\mathcal{O}(N * k * \log k)$
	Augmented	$\mathcal{O}(N + \sum_{i=1}^{N} \frac{|CandR_i|}{N} * (C + k * \log k))$

Moreover, we compare the proposed index **I-tree** with a set of existing indexing structure, such as, M-Tree [25], KD-Tree [16], and Ball-Tree [58]. These latter trees are primarily designed for the Euclidean space. Our experimental results unanimously selects **I-tree** as the winner. The augmented algorithms implemented using **I-tree** is at least 18× faster in query processing and as much as 170× faster for certain configuration. **I-tree** achieves more than 1.5× speedup during the index construction and at times it is more than 20× faster w.r.t. the baselines.

To summarize, we make the following contributions:

- We develop **DivGetBatch()**, an access primitive and show how to integrate it inside popular diversity algorithms to save up running time (Sections 3.3, 3.4, 3.5). We present in depth theoretical analyses of the augmented algorithms.
- We propose a computational framework to support **DivGetBatch()**(Section 3.6. The framework consists of a pre-computed index **I-tree** and a query processing step. We also present non-trivial solutions to maintain **I-tree** under dynamic updates.
- We run an extensive experimentation that demonstrates the effectiveness of building and maintaining **I-tree** and **DivGetBatch()**, and corroborates our theoretical claims (Section 3.7).

Table 3.2 Technical Results for Running Time Analysis w.r.t. C, m, l

Algorithm	**Variant**	**Expected time w.r.t** C	**Expected time w.r.t** m **and** l
MMR	Original	$\mathcal{O}(N * k^2)$	$\mathcal{O}(N * k^2)$
	Augmented	$\mathcal{O}((N/C + C) * k^2 + N)$	$\mathcal{O}((N/m^l + m^l) * k^2 + N)$
GMM	Original	$\mathcal{O}(N * k)$	$\mathcal{O}(N * k)$
	Augmented	$\mathcal{O}(N/C + C) * k)$	$\mathcal{O}(N/m^l + m^l) * k)$
$SWAP$	Original	$\mathcal{O}(N * k * \log k)$	$\mathcal{O}(N * k * \log k)$
	Augmented	$\mathcal{O}(N/C * k * \log k + N)$	$\mathcal{O}(N/m^l * k * \log k + N)$

Index	Activity	Time	Space	Time	Space
I-tree	Construction	$\mathcal{O}(N * C^2 * t + N^2)$	$\mathcal{O}(C^2)$	$\mathcal{O}(N * m^{2l} * t + N^2)$	$\mathcal{O}(m^{2l})$
	Maintenance	$\mathcal{O}(N * \|Y\|)$	$\mathcal{O}(C^2)$	$\mathcal{O}(N * \|Y\|)$	$\mathcal{O}(m^{2l})$

3.2 Background and Approach

This section is organized in two parts. In Subsection 3.2.1, we present the background of the studied problem and define it. In Subsection 3.2.2, we present the fundamental ideas of our approach.

3.2.1 Motivating example and problem definition

The basic principle of existing diversification algorithms, such as *MMR*, *GMM*, and *SWAP* is either to incrementally build a top-k set of diverse results or to greedily replace records in a top-k list to find the most diverse ones. In both cases, the leading cost directly depends on the number of pairwise record comparisons. Imagine a toy database D containing $N = 10$ records. Since the records are considered atomic, Table 3.4 shows a record-record similarity matrix, $simMatrixRecord$, normalized between [0-1] for our example. Diversity between r_i, r_j is simply $1 - sim(r_i, r_j)$. Given a query Q, in order to produce $k = 2$ results, an algorithm such as MMR [23] first assigns all 10 records in D to a potential candidate set R. Then it iterates over all 10 records once to find the best record in terms of MR score (based on diversity and relevance), and adds that to the result set S and discards that from R. It repeats

the same process once more to produce the resulting set $S = \{r_{10}, r_8\}$. In particular, there is a repeated pairwise computation of the following kind:

```
While  k ≤ 2 :
        rec ← R[1]
         For    i = 2; i <= |R|; i++
             if    MR(Q, R[i], S) ≥ MR(Q, rec, S)
              rec ← R[i]
         EndFor
        S ← S ⋃ rec, R ← R − rec
        k ← k + 1
EndWhile
```

Problem Definition 1. *Develop an access primitive* **DivGetBatch()** *and integrate it inside existing popular diversity algorithms.***DivGetBatch()** *satisfies the following three criteria:*

- *It guarantees identical top-k results as that of the original algorithms.*
- *It is generic, i.e., it works for any diversity functions - diversity being metric or not. A function is metric if it satisfies three properties: identity, symmetry, and triangle inequality.*
- *When integrated inside existing algorithms, it accelerates the computation and returns the results faster.*

The proposed primitive simplifies the aforementioned implementation as follows - instead of iterating over the entire R set (which is $\mathcal{O}(N)$), it returns potentially a much smaller set of records $CandR$, from which the result set S would be updated.

$\mathbf{CandR} \leftarrow \mathbf{DivGetBatch}(\mathbf{R}, \mathbf{Q}, \mathbf{S})$

While $k \leq 2:$

$rec \leftarrow Max(MR(CandR, Q, S))$

$S \leftarrow S \bigcup rec, CandR \leftarrow CandR - rec$

$k \leftarrow k + 1$

EndWhile

3.2.2 Approach

DivGetBatch() is designed by developing a computational framework, described in Figure 3.1. The basic idea is to store "higher level aggregates"', such as *minimum and maximum diversity scores of a group of records instead of keeping individual pairwise diversity scores between the records.* We formally define the minimum and maximum diversity scores as *bounds* in later sections. As an example, if the same set of records are grouped in three nodes, as shown inside the indexing box of Figure 3.1 and the maximum and minimum diversity scores are preserved between them, $node_2$ and $node_3$ can be discarded in the first iteration of processing of *MMR* pruning 6 out of the 10 records and returning only $\{r_1, r_2, r_4, r_{10}\}$ in R. This indeed leads to a significant speedup.

Offline vs. Online.

In this work, we assume that both data and query follow uniform distributions. A keen reader may notice that to accelerate diversity computation using **I-tree**, one has to "group" records and maintain some higher level aggregates between them. Grouping a large database of N records is time-consuming, as that would require partitioning them based on pairwise diversity. Indeed, this process of grouping must happen once and offline.

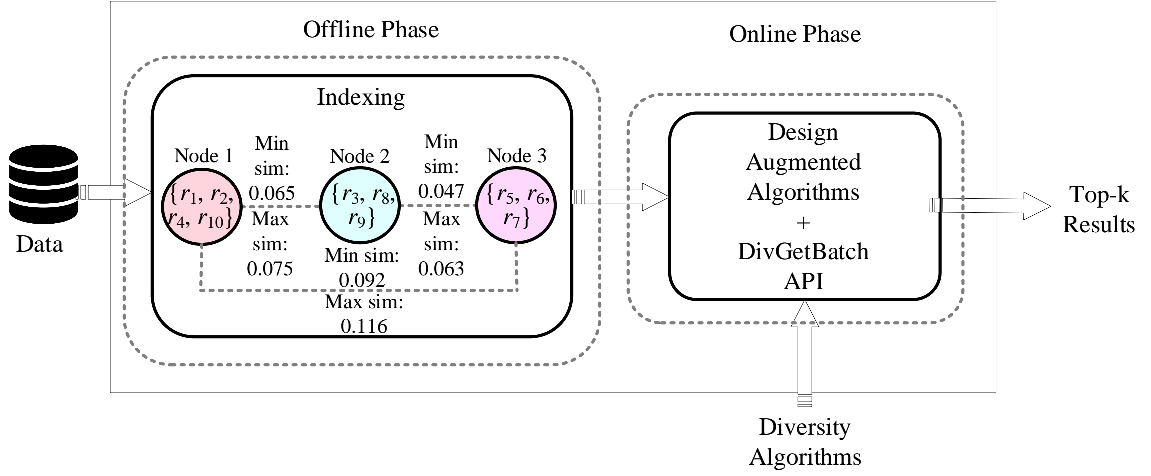


Figure 3.1 Proposed computational framework.

Precisely because of this, we resort to pre-process the records to group them and develop index **I-tree**, and use that later for processing diversity queries. This is the offline computation of the proposed framework.

Just like **DivGetBatch()**, **I-tree** is a general purpose complete tree like structure and could be designed in more than one way. It needs to satisfy three properties.

- **I-tree** has m arity and l height or levels (user inputs).
- Two highly important auxiliary data structures maintain similarity bounds between the nodes in **I-tree**: *MinsimMatrixNode* and *MaxsimMatrixNode* for maintaining minimum and maximum similarity bounds [1].
- For three nodes n, n', and n_j in **I-tree**, if n is a parent of n', and n_j is part of a different subtree and at the same level as n, the following relationship holds: **Min** $sim(n, n') \geq$ **Min** $sim(n, n_j)$, and **Max** $sim(n, n') \geq$ **Max** $sim(n, n_j)$, (basically nodes that are part of the same subtree have higher min and max similarity bounds compared to the nodes that are not).

The indexing algorithm *BuildTree* (Algorithm 5) partitions (refer to the Subroutine *Partition*) the records. It also maintains additional data structures that contain similarity scores between nodes for efficient query processing. An example of

[1]Diversity between a pair of records is simply $1 -$ *similarity* between them.

Table 3.3 Notations & Interpretations

Notations	
D	Database containing N records
S	Result set
Z	Set of nodes that contain S
R	Remaining records in the dataset
Q	Query
k	Number of records in resulting set
m, l	Arity & Total number of levels in the **I-tree**
C	Number of nodes in the **I-tree**
$CandR$	Candidate record set returned by API
Y	A batch of new records to be updated in **I-tree**

a two-level index tree is shown in Figure. 3.2. At the first level, *BuildTree* creates a root node containing all N records and m children of the root node. From the point of abstraction, it is not important at this stage to describe how the data is partitioned. Basically, the goal is to keep similar records together while separating non-similar ones. There are multiple off-the-shelf techniques such as clustering and graph partitioning to carry out this task.

In our implementation, we use the popular k-means algorithm [47] for partitioning. The algorithm repeats the partitioning procedure until it reaches l levels. We refer to Section 3.6 for further details.

Next, we present the generic recipe of using **DivGetBatch()** online or during the query processing time.

Generic Online Algorithm using DivGetBatch() The inputs to **DivGetBatch()** is **I-tree**, query Q, current candidate set of answers S, remaining records R, as well as the algorithm specific objective function f. The output is

Algorithm 1 Generic **DivGetBatch()** API

1: **Inputs:** **I-tree**, S, R, Q, f
2: **Outputs:** *CandR*: remaining eligible set of records for next iteration
3: **for** $y = 1$ **to** l **do**
4:     **for** n **in** **I-tree** *[y].nodes* **do**
5:         uB, $lB \leftarrow$ **Calculate-Bounds**(**I-tree**, n, y, f, S, Q, R)
6:         $uBs \leftarrow \bigcup uB$, $lBs \leftarrow \bigcup lB$
7:     **end for**
8:     $M \leftarrow$ **Skip-Nodes**(**I-tree**, y, uBs, lBs)
9:     $V \leftarrow \{$ **I-tree** $[y].nodes - M\}$
10: **end for**
11: $CandR = \{r \mid r \in n, n \in V\}$
12: **return** $CandR$

$CandR$, a set of candidate records that cannot be eliminated and require further processing by the original algorithm. **DivGetBatch()** explores **I-tree** level by level during query time and exploits two of its higher-level constructs: a. **Calculate-Bounds:** it computes similarity bounds [2] between Q and the nodes in **I-tree** based on a specific algorithm and objective function f. In particular, it computes an upper and a lower bound of diversity scores of the node. The goal is to decide if it is beneficial to go inside the node and explore the subtree under it. b. **Skip-Nodes:** based on the previous decision, the algorithm either skips the node and its entire subtree or explores the node.

Algorithm 1 shows the pseudo-code of the **DivGetBatch()** API.

[2]Please note diversity could be easily calculated from similarity bounds.

3.3 MMR Query Processing with DivGetBatch()

The first algorithm we study is MMR [23] algorithm. We describe the original version of the algorithm and our augmented version and provide theoretical analysis on how our augmented version outperforms the original one.

3.3.1 MMR Algorithm

Maximal Marginal Relevance (MMR) algorithm is a seminal work on result diversification [23]. MMR is based on Marginal Relvance (MR) score (Equation 3.1) that it maximizes in each iteration. Given a query, MR introduces a λ coefficient to strike a balance between the relevance score, computed between the records and the query, and the diversity score, computed between the records themselves.

MMR is greedy in nature that grows the size of the top-k set by adding records one by one in the top-k set by considering the relevance of the record and diversity with the previously selected records, using the formula below:

$$MMR(r) \leftarrow argmax_{r \in R \setminus S} MR(r),$$

$$MR(r) \leftarrow \lambda sim(r, Q) - (1 - \lambda) max_{r_j \in S} sim(r, r_j), \tag{3.1}$$

where Q is the query, S is the previously selected items, R is the remaining records in the dataset, r is a candidate record from R, and r_j is another record from S. λ is a tunable parameter. The time-consuming part of the algorithm lies in computing the MR score for each $r \in \{R \setminus S\}$ and returning the one with the highest MR score.

The MMR algorithm takes $\mathcal{O}(|R| \times |S|)$, when we add one new record to set S, demonstrating that it has an order of $N \times k$. The algorithm repeats k times and produces top-k results.

Table 3.4 Similarity Matrix for Records

	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}	Q
r_1	1.000	0.979	0.065	0.989	0.105	0.110	0.092	0.066	0.068	0.969	0.187
r_2	0.979	1.000	0.070	0.992	0.107	0.112	0.092	0.071	0.074	0.999	0.190
r_3	0.065	0.070	1.000	0.068	0.057	0.061	0.048	0.982	0.986	0.071	0.052
r_4	0.989	0.992	0.068	1.000	0.111	0.116	0.096	0.069	0.072	0.986	0.180
r_5	0.105	0.107	0.057	0.111	1.000	0.976	0.880	0.055	0.058	0.106	0.039
r_6	0.110	0.112	0.061	0.116	0.976	1.000	0.783	0.059	0.063	0.112	0.041
r_7	0.092	0.092	0.048	0.096	0.880	0.783	1.000	0.047	0.049	0.092	0.036
r_8	0.066	0.071	0.982	0.069	0.055	0.059	0.047	1.000	0.986	0.072	0.054
r_9	0.068	0.074	0.986	0.072	0.058	0.063	0.049	0.986	1.000	0.075	0.054
r_{10}	0.969	0.999	0.071	0.986	0.106	0.112	0.092	0.072	0.075	1.000	0.191

3.3.2 Aug-MMR Algorithm

Aug-MMR algorithm is designed to circumvent this aforementioned time consuming computation by leveraging **DivGetBatch()**. The general idea is to return a small subset of records, as opposed to all $|R|$ records (which is $\mathcal{O}(N)$) in each iteration, thereby saving computation. The rest of the algorithm is identical to its original version and is presented in Algorithm 2.

We now describe subroutine 2, how **DivGetBatch()** exactly works in **Aug-MMR**. Inputs to **DivGetBatch()** are **I-tree**, S, R, Q, and f (i.e., objective function of MMR). The output is $CandR$, the candidate set of records for which MR scores are to be computed to retain the best record. Based on Algorithm 1, we now describe the specifics of two higher-level constructs for **Aug-MMR**.

Calculate-Bounds: This function leverages *MinsimMatrix-Node* and *MaxsimMatrixNode* to calculate lower ($lBMR$) and upper bounds ($uBMR$), respectively. The bounds essentially represent the score of a node based on f (Equation 3.1) and mathematically can be expressed as follows:

$$lBMR_{node} \leftarrow \lambda\mathbf{Min}\ sim(node, Q) - max_{node' \in Z}(1-\lambda)\mathbf{Max}\ sim(node, node'), \quad (3.2)$$

Algorithm 2 Aug-MMR

Inputs: **I-tree**, D, MMR, Q, k

Outputs: S: final top-k result set.

1: $R \leftarrow D$, $S = \phi$

2: **for** $t = 1$ **to** k **do**

3: $CandR \leftarrow$ **DivGetBatch**(**I-tree**, R, S, Q, MMR)

4: $S = \{S \bigcup MMR(r)_{r \in CandR}\}$

5: **end for**

6: **return** S

$$uBMR_{node} \leftarrow \lambda \mathbf{Max}\ sim(node, Q) - min_{node' \in Z}(1-\lambda)\mathbf{Min}\ sim(node, node'), \quad (3.3)$$

where Z is the set of nodes that contain S,
Min $sim(node, Q)$ and **Max** $sim(node, Q)$ are the minimum and the maximum similarity between any records in *node* and Q, respectively, and **Min** $sim(node, node')$ and **Max** $sim(node, node')$ are the minimum and the maximum similarity between any two records in *node* and *node*$'$, respectively. Since $lBMR$ is the smallest score of *node*, it is calculated by taking the minimum of *sim* score in the first part of the equation and subtracting that from the maximum of *sim* score in the second part. Contrarily, $uBMR$ refers to the maximum MR score of *node* (Equation 3.3) and can be calculated by reversing the min and max of the (Equation 3.2).

Skip-Nodes: The argument of node skipping is simple - if the $uBMR$ score of a node is not larger than the $lBMR$ of another node, then the former node and its entire subtree could be pruned. The records from the remaining nodes form the

$CandR$ set.

$$CandR \leftarrow \{N - \{r \in \mathbf{I} - \mathbf{tree}.n \mid uBMR_n < \max_{\forall n'}(lBMR_{n'})\}\} \tag{3.4}$$

this is done by finding the maximum value of $lBMR_{n'}$ of all nodes and then discard ones with $uBMR$ less than it. **Running Example:** A step by step calculation of **DivGetBatch()** is shown in Table 3.5. The maximum and minimum similarity between $node_1$ and query Q is 0.180 and 0.191. In first iteration of CALCULATE-BOUNDS, lower bound of MR of $node_1$ which is $lBMR_{node_1} = 0.8 * 0.180 - (1 - 0.8) * 0 = 0.144$, and upper bound of MR of $node_1$, $uBMR_{node_1} = 0.8 * 0.191 - (1 - 0.8) * 0 = 0.153$. Similarly, $lBMR_{node_2}$, $uBMR_{node_2}$, $lBMR_{node_3}$, and $uBMR_{node_3}$ are -0.047, 0.044, 0.029, and 0.033, respectively. In SKIP-NODES, the maximum of all $lBMR$s is found 0.144 which is $lBMR_{node_1}$.

$uBMR_{node_2}$ and $uBMR_{node_3}$ are smaller than
$lBMR_{node_1}$. Therefore, $node_2$ and $node_3$ are discarded from further calculation in iteration 1. Records of $node_1$ $\{r_1, r_2, r_4, r_{10}\}$ are returned by **DivGetBatch()** to **Aug-MMR** algorithm. **Aug-MMR** performs calculation similar to original MMR on $\{r_1, r_2, r_4, r_{10}\}$ which results in $S = \{r_{10}\}$. Likewise, the maximum and minimum similarity between $node_1$ and $node_1$ are 0.969 and 1.0. In the second iteration of CALCULATE-BOUNDS, $lBMR_{node_1} = 0.8 * 0.180 - (1 - 0.8) * 0.969 = -0.050$ and $uBMR_{node_1} = 0.8 * 0.191 - (1 - 0.8) * 1.0 = -0.047$. Similarity, $lBMR_{node_2}$, $uBMR_{node_2}$, $lBMR_{node_3}$, and $uBMR_{node_3}$ are 0.028, 0.029, 0.010, and 0.009, respectively. In SKIP-NODES, the maximum of all $lBMR$s is $lBMR_{node_2} = 0.028$. $uBMR_{node_1}$ and $uBMR_{node_3}$ are smaller than $lBMR_{node_2}$. Thus, $node_1$ and $node_3$ are discarded from further calculation in iteration 2. Records of $node_2$ $\{r_3, r_8, r_9\}$ are returned by **DivGetBatch()** to **Aug-MMR** algorithm. **Aug-MMR** performs calculation similar to original MMR on $\{r_3, r_8, r_9\}$ which results in $S = \{r_{10}, r_8\}$

Table 3.5 First Two Iterations of DivGetBatch() in Aug-MMR

Functions	Nodes	Bounds	Iteration 1	Iteration 2
Calculate-Bounds	$node_1$	$lBMR$	$0.8 * 0.180 - (1 - 0.8) * 0 = 0.144$	-0.050
		$uBMR$	$0.8 * 0.191 - (1 - 0.8) * 0 = 0.153$	-0.047
	$node_2$	$lBMR$	$0.8 * 0.0191 - (1 - 0.8) * 0 = 0.0152$	0.028
		$uBMR$	$0.8 * 0.054 - (1 - 0.8) * 0 = 0.044$	0.029
	$node_3$	$lBMR$	$0.8 * 0.036 - (1 - 0.8) * 0 = 0.029$	0.010
		$uBMR$	$0.8 * 0.041 - (1 - 0.8) * 0 = 0.033$	0.009
Skip-Nodes			$lBMR$ array: 0.144, 0.041, 0.029 $uBMR$ array: 0.153, 0.044, 0.033 $node_2$, $node_3$ are skipped. $CandR = \{r_1, r_2, r_4, r_{10}\}$. $MMR(r_1, r_2, r_4, r_{10}) \leftarrow r_{10}$ Number of records discarded is 6	$lBMR$ array: -0.050, 0.028, 0.010 $uBMR$ Array: -0.047, 0.029 , 0.009 $node_1$, $node_3$ are skipped. $CandR = \{r_3, r_8, r_9\}$ $MMR(r_3, r_8, r_9) \leftarrow r_8$ top-2 set $= \{r_{10}, r_8\}$

Aug-MMR algorithm proofs

Claim 1. Aug-MMR *returns identical top-k results as that of original MMR.*

Proof. The proof is constructed using one helper lemma and one observation: Lemma 8 proves that **DivGetBatch()** never prunes a record that is part of the original top-k; Observation 1 shows that once the control comes back from **DivGetBatch()**, **Aug-MMR** works exactly as the original MMR in each iteration. Combining these lemma and observation, **Aug-MMR** returns identical top-k results as that of the original MMR. □

Lemma 1. DivGetBatch() *never prunes a record that is part of the original top-k.*

Proof. As part of this proof, we first prove that Skip-Nodes *never discards* the record which has the highest MR score in that iteration.

Recall Property 1 and note that for every two nodes n and n' in the same subtree, if n is a parent of n', then n contains all records in n', thereby having larger

$uBMR$ and $lBMR$ values. Therefore, if a node n is skipped, any child of n is also safe to be skipped.

We use helper Lemma 7 to prove that the actual MR score of any record in a node $node$ is bounded between $uBMR_{node}$ and $lBMR_{node}$. Let us assume, the next desired record $r_d \in node_d$ produces maximum MR value among all $R \setminus S$ records. MR_{r_d} is greater than $minMR_{node}$ for $\forall node$. Using Equation 3.6:

$$\begin{aligned} MR_{r_d} &\geq max_{node \in \mathbf{I-tree}[l].nodes} minMR_{node} \\ &\geq max_{node \in \mathbf{I-tree}[l].nodes}(lBMR_{node}), \end{aligned}$$

Using Equation 3.6, $MR_{r_d} = MaxMR_{node_d} \leq$
$uBMR_{node_d}$. As a result,

$$\begin{aligned} uBMR_{node_d} &\geq MR_{r_d} \\ &\geq max_{node \in \mathbf{I-tree}[l].nodes}(lBMR_{node}). \end{aligned} \tag{3.5}$$

According to Equations (3.5) and (3.4), $node_d$ will not be discarded, and all records inside $node_d$ including r_d will be returned by **DivGetBatch()** or send to the next level for further processing. This logic extends for all the iterations. Therefore, **DivGetBatch()** never prunes a record that is part of the original top-k. □

Lemma 2. *MR score of any record $r \in node$ (say MR_r) is bounded by upper and lower bound $uBMR_{node}$ and $lBMR_{node}$, respectively. That is,*

$$lBMR_{node} \leq MR_{r \in node} \leq uBMR_{node}. \tag{3.6}$$

Proof. We will first prove that maximum relevance value (say MR_{rmax}) of any record (say $r_{max} \in node$) is less than equal to $uBMR_{node}$. Where, MR_{rmax} can be expressed as:

$$MR_{rmax} = \lambda sim(r_{max}, Q) - (1 - \lambda) max_{r_j \in S} sim(r_{max}, r_j)]. \tag{3.7}$$

First part of the Equation (3.7) is always less than equals to first part of the Equation (3.3). That is:

$$\lambda sim(r_{max}, Q) \leq \lambda max_{r_i \in node} sim(r_i, Q) \\ = \lambda \mathbf{Max}\ sim(node, Q), \tag{3.8}$$

Next, we show that second part of the Equation (3.7) is always greater than second part of the Equation (3.3).

Let us assume; $r_w \in S$ produces max value for the second part of Equation (3.7). That second part can be rewritten as $(1-\lambda)sim(r_{max_{node}}, r_w)$. Let us assume, $r_w \in node_w$ where $node_w \in Z$. For any $node' \in Z$, we can write:

$$(1-\lambda)sim(r_{max}, r_w) \geq (1-\lambda)min_{r_i \in node, r_j \in node'} \\ sim(r_i, r_j) \\ \geq min_{node' \in Z}(1-\lambda)\mathbf{Min}\ sim(node, node'), \tag{3.9}$$

From these two inequalities (3.8) and (3.9), we can conclude $MR_{rmax} \leq uBMR_{node}$ or, $MR_{r \in node} \leq uBMR_{node}$.

Similarly, the lower bound $lBMR_{node}$ can be shown as follows: $lBMR_{node} \leq minMR_{node}$.

Thus, any record in *node* is certain to have MR value in between $uBMR_{node}$ and $lBMR_{node}$.

□

Observation 1. *Once the control comes back from* **DivGetBatch()**, **Aug-MMR** *works exactly as the original* MMR *in each iteration.*

Aug-MMR has identical MR score calculation and MMR selection as that of the original MMR.

Claim 2. **Aug-MMR** *requires* $\mathcal{O}((N/C + C) * k^2 + N)$ *time in expectation.*

Proof. In the original MMR algorithm, each iteration for finding one record takes $\mathcal{O}(N * k)$ times. For k iterations, the overall running time is therefore $\mathcal{O}(N * k^2)$. The running time of **Aug-MMR** does not need to go over all N records in each iteration. Instead, it relies on **DivGetBatch()** to obtain a smaller set $CandR$ records.

Part 1. Running time of the API: A single iteration of **DivGetBatch()** needs to go over all the nodes in **I-tree** and takes $\mathcal{O}(C * k)$ time. **DivGetBatch()** has to compute two subroutines:
Calculate-Bound and *Skip-Nodes*. To compute these two functions, it takes $\mathcal{O}(N)$ time. Therefore, the overall running time is $\mathcal{O}(C * k^2 + N)$, where C is the total number of nodes.

Part 2. Running time of the rest of computation: The rest of the computation depends on the size of $CandR$. Let us assume, **DivGetBatch()** returns $|CandR_i|$ records in the i-th iteration. Accordingly, we have:

$$T_{\mathbf{Aug-MMR}} = \mathcal{O}(C * k^2 + N + \sum_{i=1}^{k} |CandR_i| * k).$$

The expected case analysis basically delves deeper into the analysis of **Part 2** and studies the expected running time considering different size of $CandR_i$ and its corresponding probability.

Let us assume, in iteration i, the $|CandR_i|$ records touch x number of nodes in **I-tree**. Indeed, x_i is the number of nodes with $|CandR_i|$ records in **I-tree**, that the augmented algorithms have to access during the query processing. Let us also assume node n_i contains v_i records. We start the proof assuming there is only one level in **I-tree** (i.e., $l = 1$), and then generalize it later on. If $l = 1$, the expected running time of Part 2 calculation of **Aug-MMR** in the i-th iteration is:

$$E = \mathcal{O}(\sum_{i=1}^{C} prob(x_i) \times \text{computation cost}_{\mathbf{Aug-MMR}}(x_i)).$$

Now, probability of returning x nodes = $\binom{C}{x}$ * probability of x nodes getting selected * probability of $(C-x)$ nodes not getting selected.

We assume that both data and query follow uniform distributions, thereby, each node has an equal probability of getting selected or skipped. The probability of choosing a node is $1/C$. Therefore, the probability of not getting selected is $(1-1/C)$.

The size of the returned record set, i.e., $|CandR|$, if $x = i$ nodes are accessed:

$$
\begin{aligned}
|CandR|_i = {} & (1/C)^i * (1-1/C)^{C-i} * [(v_1 + v_2 + ... + v_i) \\
& + (v_1 + v_3 + ... + v_{i+1}) + (v_2 + v_3 + ... + v_{i+1}) \\
& + (v_3 + v_4 + ... + v_{i+2}) + \ldots] \\
& = (1/C)^i * (1-1/C)^{C-i} * \binom{C-1}{i-1} \\
& * (v_1 + v_2 + \ldots + v_C) \\
& = (1/C)^i * (1-1/C)^{C-i} * \binom{C-1}{i-1} * N.
\end{aligned}
$$

Therefore, the overall expected cost of Part 2 is:

$$
\begin{aligned}
|CandR| = {} & N * \sum_{i=1}^{C} (1/C)^i * (1-1/C)^{C-i} * \binom{C-1}{i-1} \\
& = N * (1/C)/(1-1/C) * \sum_{i=1}^{C} (1/C)^{i-1} * \\
& (1-1/C)^{C-(i-1)} * \binom{C-1}{i-1}. \\
& Let\ j = i-1: \\
& = N * (1/C)/(1-1/C) * \sum_{j=0}^{C-1} (1/C)^j * \\
& (1-1/C)^{C-j} * \binom{C-1}{j} \\
& = N * (1/C)/(1-1/C) * (1-1/C) *
\end{aligned}
$$

$$\sum_{j=0}^{C-1}(1/C)^j * (1-1/C)^{(C-1)-j} * \binom{C-1}{j}$$
$$= N * (1/C)/(1-1/C)*$$
$$(1-1/C) * (1/C + 1 - 1/C)^{C-1} = N/C.$$

Expected running time of **Aug-MMR** algorithm considering both Part 1 and Part 2 computation is:

$E_{\mathbf{Aug-MMR}} = \mathcal{O}((N/C + C) * k^2 + N).$

Now consider the case when $l > 1$. Probability of selecting a node in first level is $1/m$, given m is the arity of **I-tree**. Probability of selecting a node in second level = probability of selecting that node out of m node in that branch * probability of selecting it's parent = $1/m^2$. Similarly, Probability of selecting a node at leaf node is $1/m^l = 1/C$. Thus, in the general case, when $l > 1$, expected running time of **Aug-MMR** is $\mathcal{O}((N/C + C) * k^2 + N)$, which is same as before.

□

Worst-case Aug-MMR . In the worst-case, all N records are returned by **DivGetBatch()** in each iteration, which makes $\sum_{i=1}^{k}|CandR_i| = N * k$. Thus, the worst-case running time is $\mathcal{O}((N + C) * k^2)$.

3.4 *GMM* Query Processing with DivGetBatch()

The second algorithm we study is GMM algorithm. We describe the original version of the algorithm and our augmented version and similar to the previous section. We also provide proofs on how our augmented version outperforms the original one.

3.4.1 GMM Algorithm

The next algorithm we study is GMM [44] that tries to find a subset of k most diverse records among N records by maximizing the minimum pairwise distance. GMM does not require any external query. Based on the original design, the first two records in the result set S are provided in constant time by an *oracle*. Then, the algorithm iteratively goes through all records in R and finds a record whose minimum *diversity* (maximum similarity) with the previously selected records is the largest (smallest). It continues until $|S|=k$. The objective function is:

$$GMM(r) \leftarrow argmax_{r \in R \setminus S} min_{r_j \in S} Div(r, r_j), \tag{3.10}$$

where $Div(r, r_j)$ is the diversity score between record r and r_j. A keen reader may notice that GMM uses diversity (Div) in the objective function, whereas, in our study, we store similarity between records. Unless specified otherwise, $Div = 1 - sim$. The two similarity matrices, one that captures the similarity between every pair of records, and the other that captures that of between nodes, could be used to calculate Div.

3.4.2 Aug-GMM Algorithm

Aug-GMM leverages the **DivGetBatch()** API to reduce the number of records to iterate on. Algorithm 3 describes the pseudo-code, where the **DivGetBatch()** returns a small subset of records $CandR$ which later on is fed to the original GMM algorithm to get the $nextBest$ record.

Calculate-Bounds: This function keeps track of the upper and lower bounds of scores between nodes ($uBGMM$ and $lBGMM$, respectively) using the same principles as that of the original GMM objective function (Equation 3.10).

$$lBGMM_{node} \leftarrow min_{node' \in Z} \min Div(node, node'), \tag{3.11}$$

$$uBGMM_{node} \leftarrow min_{node' \in Z} \max Div(node, node'), \tag{3.12}$$

where Z is the set of nodes containing S, $\min Div(node,$
$node')$ and $\max Div(node, node')$ are the minimum and the maximum diversity scores between any two records in $node$ and $node'$, respectively. In Equation 3.11, minimum of the minimum diversity over all nodes in Z ensures the lower bound of GMM, such that all records in $node$ will have equal or greater value than $lBGMM_{node}$. Conversely, in Equation 3.12, minimum of the maximum diversity over all nodes in Z ensures the upper bounds, such that all records in $node$ will have equal or lower GMM value than $uBGMM_{node}$.

Skip-Nodes : This function is identical to SKIP-NODES of MMR in principle. The skip-rationale of **Aug-GMM** is:

$$CandR \leftarrow \{N - \{r \in \mathbf{I} - \mathbf{tree}.n \mid uBGMM_n < \max_{\forall n'}(lGMM_{n'})\}\} \tag{3.13}$$

Running Example: Let us assume $k = 3$ and the first two records of S are arbitrarily chosen as r_1 and r_3. Initially, $S = \{r_1, r_3\}$. From Figure 3.1, r_1 and r_3 are inside $node_1$ and $node_2$, respectively. Hence, $Z = \{node_1, node_2\}$. Node-Node diversity $Div(node, node')$ can be calculated using Div = 1 - Sim. $Div(node_3, node_1)$ = (0.884, 0.908) and $Div(node_3, node_2)$ = (0.937, 0.9530). By using Equations (3.11) and (3.12), $lBGMM_{node_3}$ = 0.884 (as min of min div) and $uBGMM_{node_3}$ = 0.908 (as min of max div). Similarly, $lBGMM_{node_1}$, $uBGMM$
$_{node_1}$, $lBGMM_{node_2}$, and $uBGMM_{node_2}$ are 0, 0.031, 0, and 0.018. $lBGMM_{node_3}$ (0.884) is greater than $uBGMM$

$_{node_1}$ (0.031) and $uBGMM_{node_2}$ (0.018). Using Equation 3.13, $node_1$ and $node_2$ can be discarded. Obtaining records from $node_3$, $candR = \{r_5, r_6, r_7\}$ is returned from **DivGetBatch()**. Finally, GMM$(r_5, r_6, r_7) = r_5$ is called and the result set $S = \{r_1, r_3, r_5\}$ is achieved.

Aug-GMM algorithm proofs

Claim 3. Aug-GMM *returns identical top-k results as that of original GMM.*

Proof. Akin to MMR proof, this proof is also constructed using one helper lemma and one observation: Lemma 3 proves that **DivGetBatch()** never prunes a record that is part of the original top-k; Observation 2 shows that in each iteration, once the control comes back from **DivGetBatch()**, **Aug-GMM** works exactly as the original GMM. Combining these lemma and observation, **Aug-GMM** returns identical top-k results as that of the original GMM. □

Lemma 3. DivGetBatch() *never prunes a record that is part of the original top-k.*

Proof. As part of this proof, we first prove that SKIP-NODES *never discards* the record which has the highest GMM score in that iteration.

We use helper Lemma 4 to prove that the actual GMM score of any record in a node *node* is bounded between $uBGMM_{node}$ and $lBGMM_{node}$. The rest of the proof is identical to Lemma 8 of **Aug-MMR**. □

Lemma 4. *GMM score of any record $r \in node$ (say GMM_r) is bounded by upper and lower bound*

$uBGMM_{node}$ and $lBGMM_{node}$, respectively. That is,

$$lBGMM_{node} \leq GMM_{r \in node} \leq uBGMM_{node}.$$

Proof. Let us first consider $uBGMM_{node}$, by assuming

$F(node, r_j) = max_{r_i \in node} Div(r_i, r_j)$, it can be re-written as:

$$uBGMM_{node} \leftarrow min_{node' \in Z}[max_{r_j \in node'} F(node, r_j)], \tag{3.14}$$

Let us assume, maximum GMM value produced by any record in *node* is $maxGMM_{node}$. According to Equation 3.10, $maxGMM_{node}$ is expressed as follows:

$$\begin{aligned} maxGMM_{node} &= max_{r_i \in node}[min_{r_j \in S} Div(r_i, r_j)], \\ &= min_{r_j \in S}[max_{r_i \in node} Div(r_i, r_j)], \\ &= min_{r_j \in S} F(node, r_j), \\ &\leq min_{node' \in Z}[max_{r_j \in node'} F(node, r_j)], \\ &= uBGMM_{node}, \text{[using equation 3.14].} \end{aligned}$$

similarly, it can be proved that, $minGMM_{node} \geq lBGMM_{node}$.

□

Observation 2. *Once the control comes back from* **DivGetBatch()***,* **Aug-GMM** *works exactly as the original GMM in each iteration.*

Aug-GMM does exactly same calculation as the original GMM does on a set of records as a result it will produce the same record as GMM does in a single iteration.

Claim 4. Aug-GMM *requires $\mathcal{O}(N/C + C) * k)$ time in expectation.*

Proof. In the GMM algorithm, each iteration for finding one record takes $\mathcal{O}(N)$ times. For k iteration, the overall running time is $\mathcal{O}(N * k)$. Similar to **Aug-MMR**, **Aug-GMM** does not need to go over all N records in each iteration, instead relies on **DivGetBatch()** to obtain a smaller set $CandR$ records.

Part 1. Running time of the API: A single iteration of **DivGetBatch()** needs to go over all the nodes in **I-tree** and takes $\mathcal{O}(C)$ time. **DivGetBatch()** has to

compute two subroutines:

Calculate-Bound() and *Skip-Nodes*(). To compute these two functions, it takes $\mathcal{O}(C)$ time. Therefore, the overall running time is $\mathcal{O}(C * k)$, where C is the total number of nodes.

Part 2. Running time of the rest of computation: Similar to **Aug-MMR**, The rest of the computation depends on the size of $CandR$. Let us assume, **DivGetBatch()** returns $|CandR_i|$ records in the i-th iteration. Hence, we have:

$$T_{\mathbf{Aug-GMM}} = \mathcal{O}(C * k + \sum_{i=1}^{k} |CandR_i|).$$

The expected case analysis basically delves deeper into the analysis of **Part 2** and studies the expected running time considering different size of $CandR_i$ and its corresponding probability. By performing similar calculation as that of **Aug-MMR** as shown before, the expected cost of **Aug-GMM** is:

$$E_{\mathbf{Aug-GMM}} = \mathcal{O}((N/C + C) * k).$$

□

Worst-case Aug-GMM . In the worst-case, all N records are returned by **DivGetBatch()** in each iteration, which makes $\sum_{i=1}^{k} |CandR_i| = N * k$. Then the worst-case running time is: $\mathcal{O}((N + C) * k)$.

3.5 *SWAP* Query Processing with DivGetBatch()

The last algorithm we study is *SWAP* [95]. We describe the original version and our proposed augmented version. Similar to the previous sections, we provide theoretical analysis.

Algorithm 3 Aug-GMM

Inputs: **I-tree**, D, GMM, k

Output: S: final top-k result set

1: $S \leftarrow$ two records selected by an oracle
2: $R \leftarrow \{D - S\}$
3: **for** $t = 1$ **to** $k - 2$ **do**
4: $\quad CandR \leftarrow$ **DivGetBatch**(**I-tree**, R, S, GMM)
5: $\quad S = \{S \bigcup GMM(r)_{r \in CandR}\}$
6: **end for**
7: **return** S

3.5.1 *SWAP* Algorithm

SWAP [95] is a greedy algorithm that produces top-k results based on a given query Q and a tunable parameter that controls how much relevance could at most drop between any two records in the top-k results. The algorithm starts by sorting the records w.r.t. relevance and initializing the top-k result set S with the k-records with the highest relevance score with Q. It finds a *candidate record* from the current top-k set that has the smallest diversity contribution based on Equation (3.15). Indeed, in each iteration, it attempts to swap one record from R \ S with the candidate record. It starts scanning the remaining sorted relevance list from the top. In every iteration, it attempts to swap one record from the current top-k set with another from sorted R if the latter record has a higher contribution to diversity while ensuring the threshold of relevance drop. The algorithm terminates when the relevance drop is below the threshold, or R is fully scanned.

$$Divcont(r_i, S) = \sum_{r_j \in S} Div(r_i, r_j). \tag{3.15}$$

3.5.2 Aug-SWAP Algorithm

Aug-SWAP is identical to the $SWAP$, i.e., it scans the sorted relevance list R, after initializing the top-k set S. It calls the **DivGetBatch()** API to retrieve a smaller set of candidate records $CandR$. These $CandR$ records are eligible to be considered during the next swap. If a record in R is not in $CandR$, then it is skipped. The rest of the process is identical to the original $SWAP$ algorithm. Algorithm 4 contains the pseudo-code.

Calculate-Bounds: Once the records are sorted w.r.t. relevance score, the diversity computation becomes query independent, and only between the records. This function calculates the upper and lower bounds of diversity contribution of nodes by leveraging
$MinsimMatrixNode$ and $MaxsimMatrixNode$ considering the set of nodes Z that contains S, as below:

$$uBSWAP_{node} \leftarrow \sum_{node' \in Z} \max Div(node, node'), \tag{3.16}$$

$$lBSWAP_{node} \leftarrow \sum_{node' \in Z} \min Div(node, node'), \tag{3.17}$$

where $\max Div(node, node')$ and $\min Div(node, node')$ are the max and the min diversity between $node$ and $node'$. Naturally, the maximum (minimum) diversity is the maximum (minimum) of node diversities between $node$ and the nodes in Z.

Skip-Nodes: This function will then check if
$uBSWAP_{node}$ is less than the diversity contribution of the candidate record (3.18); If the condition is true, it will prune the node and the entire subtree under it. In such a case, none of the records inside this node are eligible for swap because they will not increase the overall diversity of S. **DivGetBatch()** returns the records for

all non-pruned nodes:

$$CandR \leftarrow \{N - \{r \in \mathbf{I} - \mathbf{tree}.n \mid uBSWAP_n < \\ min_{r_i \in S} \sum_{r_j \in S} Div(r_i, r_j)\}\} \tag{3.18}$$

Running Example: Lets say, $k = 2$, $UB = 0.9$, sorted $R = \{r_8, r_7, r_2, r_1, r_4, r_9, r_3, r_6, r_{10}\}$, and initial top-2 records selected as S=$\{r_8, r_7\}$. Using Equation 3.15, $Divcont(r_7, S)$ = 0.953 and the *candidate* is r_7. From Figure 3.1, $Z = \{node_2, node_3\}$. Using Equations (3.16), (3.17), and Figure 3.1, if Div = 1 - sim, we have:
$uBSWAP_{node_1} = maxDiv(node_1, node_2) = 0.935$,
$lBSWAP_{node_1} = minDiv(node_1, node_2) = 0.925$.
Then, Equation 3.18 is applied and $node_1$ is discarded, $node_2$, $node_3$ are returned by **DivGetBatch()**, and $CandR = \{r_3, r_9, r_5, r_6\}$. Next record in the sorted list is r_2, which is not in $CandR$. As a result, r_2 will be skipped.

Aug-SWAP algorithm proofs

Claim 5. **Aug-SWAP** *returns identical top-k results as that of original* SWAP*.*

Proof. This proof is constructed using one helper lemma and one observation. Lemma 5 proves that **DivGetBatch()** does not skip a record that has a higher diversity contribution than that of the candidate record. Observation 3 shows that once all records returned in $CandR$, **Aug-SWAP** is identical to $SWAP$. Combining these lemma and observation, **Aug-SWAP** returns identical top-k results as that of the original $SWAP$.

□

Lemma 5. **DivGetBatch()** *never prunes a record that is part of the original top-k.*

Algorithm 4 Aug-SWAP

Inputs: **I-tree**, D, UB, k, $SWAP$

Output: S: final top-k result set.

1: $R \leftarrow$ Sort D on score;
2: $S \leftarrow$TOPKITEMS(R, k)
3: $candidate \leftarrow argmin_{r_i \in S} Equation$ 3.15
4: $CandR \leftarrow R$
5: $pos \leftarrow k + 1$
6: **while** $candidate.score$ - $R[pos].score < UB$ **do**
7: **if** $R[pos]$ **in** $CandR$ **then**
8: **if** $Divcont(R[pos], S) > Divcont(candidate, S)$ **then**
9: $S \leftarrow \{S - candidate \bigcup R[pos]\}$
10: $CandR \leftarrow$ **DivGetBatch**(**I-tree**, R, S, Q, $SWAP$)
11: $candidate \leftarrow argmin_{r_i \in S} Equation$ 3.15
12: **end if**
13: **end if**
14: pos++
15: **end while**
16: **return** S

Proof. As part of this proof, we first prove that in each iteration SKIP-NODES *never discards* a record which has the higher diversity contribution than that of the candidate record. Let us assume, $r_{cand} \in S$ has lowest diversity contribution in S.

$$Divcont(r_{cand}, S) = min_{r_i \in S} \sum_{r_j \in S} Div(r_i, r_j)\}$$
$$= min_{r_i \in S} Divcont(r_i, S).$$

We use helper Lemma 6 to prove that the actual $DivCont$ score of any record in a node $node$ is bounded between $uBSWAP_{node}$ and $lBSWAP_{node}$. Let us assume, $r_d \in node_d$ is a record inside $node$, therefore,

$$\begin{aligned} uBSWAP_{node_d} &\geq Divcont(r_d, S) \\ &\geq Divcont(r_{cand}, S) \\ &= min_{r_i \in S} \sum_{r_j \in S} Div(r_i, r_j), \end{aligned}$$

as a result,

$$uBSWAP_{node_d} \geq min_{r_i \in S} \sum_{r_j \in S} Div(r_i, r_j). \tag{3.19}$$

From Equations (3.18) and (3.19), it is evident that $node_d$ containing r_d will not be skipped by SKIP-NODES. This logic extends to all the iterations SKIP-NODES calls. Hence the proof. □

Lemma 6. *Divcont score of any record $r \in node$ is bounded by upper and lower bound $uBSWAP_{node}$ and $lBSWAP_{node}$ respectively. That is,*

$$lBSWAP_{node} \leq Divcont(r, S)_{r \in node} \leq uBSWAP_{node}. \tag{3.20}$$

Proof. By replacing the value of $\max Div(node, node')$, the upper bound can be written as:

$$uBSWAP_{node} \leftarrow \sum_{node' \in Z} max_{r_i \in node, r_j \in node'} Div(r_i, r_j). \tag{3.21}$$

For any record $r \in node$ and $r_j \in S$, $r_j \in node_d$ and $node_j \in Z$,

$$Div(r, r_j) \leq max_{r_i \in node} Div(r_i, r_j),$$

Or,

$$\sum_{r_j \in S} Div(r, r_j) \leq \sum_{node' \in Z} max_{r_i \in node, r_j \in node'} Div(r_i, r_j),$$

As a result, $Divcont(r, S) \leq uBSWAP_{node}$. similarly, we can prove: $Divcont(r, S) \geq lBSWAP_{node}$.

□

Observation 3. *Once the control comes back from* **DivGetBatch()***,* **Aug-SWAP** *works exactly as the original SWAP does in each iteration.*

Aug-SWAP performs identical calculation of *SWAP* on the records that are not pruned by **DivGetBatch()**.

Claim 6. **Aug-SWAP** *requires* $\mathcal{O}(N/C * k * \log k + N)$ *time in expectation.*

Proof. In the original *SWAP* algorithm, each iteration to select a new record to be swapped with the candidate record takes $\mathcal{O}(k * \log k)$ time. Therefore, for going over all records in R, it takes $\mathcal{O}(N * k * \log k)$. **Aug-SWAP** does not need to perform $\mathcal{O}(N * k * \log k)$, instead relies on **DivGetBatch()** to obtain a smaller set $CandR$ records.

Part 1. Running time of the API: A single iteration of **DivGetBatch()** needs to go over all the nodes in **I-tree**. **DivGetBatch()** has to compute two subroutines: *Calculate-Bound* and *Skip-Nodes*. By updating only the most recent swapped records and using dynamic programming, the two subroutines' overall running time is $\mathcal{O}(C)$, where C is the total number of nodes. However, how many times the API gets called depends on the number of times the swap condition gets satisfied (recall lines 8-10 in **Aug-SWAP** algorithm).

Part 2. Running time of the rest of computation: The other major computation of this algorithm is the running time of a record be swapped, which is $\mathcal{O}(k * \log k)$ and *Divcont* running time in the Algorithm 5 line 8, which is $\mathcal{O}(k)$. How many times *Divcont* gets executed depends on Line 7 in the **Aug-SWAP** algorithm is satisfied. The number of times *SWAP* gets executed depends on swap condition, which is Line 8 in the **Aug-SWAP** algorithm. Finally, the entire R needs to be exhausted (as long

as the bound drop threshold is satisfied), which takes $\mathcal{O}(N)$ time. As a result, we have:

$$\begin{aligned} T_{\mathbf{Aug-SWAP}} = \mathcal{O}(&Number\, of\, times\, swap\, is\, satisfied \\ &* \mathbf{DivGetBatch}()\, runtime + \\ &Number\, of\, times\, swap\, is \\ &satisfied * SWAP\, runtime + \\ &number\, of\, times\, line\, 7\, is\, satisfied * \\ &Divcont\, runtime + N). \end{aligned}$$

By considering running time of single *Divcont*, $SWAP$, and **DivGetBatch**() call, overall running time of **Aug-SWAP** becomes:

$$\begin{aligned} T_{\mathbf{Aug-SWAP}} &= \mathcal{O}(Number\, of\, times\, swap\, is\, satisfied \\ &* C + Number\, of\, times\, swap\, is\, satisfied \\ &* k * \log k + number\, of\, times\, line\, 7 \\ &is\, satisfied * k + N). \\ &= \mathcal{O}(\sum_{i=1}^{N}[probability\, of\ \ swap\, satisfied \\ &* C + probability\, of\ \ swap\, satisfied \\ &* k * \log k + probability\, of number\, of \\ ×\, line\, 7\, is\, satisfied * k] + N) \end{aligned}$$

Expected size of $CandR$ is $\sum_{i=1}^{N} \frac{|CandR_i|}{N}$. Probability of line 7 satisfied = probability that $R[pos]$ is in $CandR$ = $\frac{\sum_{i=1}^{N} \frac{|CandR_i|}{N}}{N}$. Without further information,

the probability of a record getting swapped is 1/2 (same as not getting swapped). Probability of $SWAP = 1/2 *$line 7 is satisfied $= 1/2 * \frac{\sum_{i=1}^{N} \frac{|CandR_i|}{N}}{N}$. Expected running time (cost) of **Aug-SWAP** is:

$$
\begin{aligned}
E_{\mathbf{Aug-SWAP}} &= \sum_{i=1}^{N} [1/2 * \frac{\sum_{i=1}^{N} \frac{|CandR_i|}{N}}{N} * (C + k * \log k) \\
&+ \frac{\sum_{i=1}^{N} \frac{|CandR_i|}{N}}{N} * k] + N \\
&= 1/2 * \sum_{i=1}^{N} \frac{|CandR_i|}{N} * (C + k * \log k) \\
&+ \sum_{i=1}^{N} \frac{|CandR_i|}{N} * k + N \\
&= \mathcal{O}(\sum_{i=1}^{N} \frac{|CandR_i|}{N} * (C + k * \log k) + N)
\end{aligned}
$$

First, we study the Part 2 computation having two costs associated with it, cost of *Divcont* and cost that of $SWAP$. Based on Line 7 of Algorithm 5, if $CandR$ is large, it is likely to have $R[pos]$ inside it. In fact, if $CandR$ contains all R records, $R[pos]$ will always be there. For the purpose of illustration, let us assume, in the i-th iteration, $|CandR_i|$ records touch x number of nodes in **I-tree** and node n_i contains v_i records. Therefore, the probability that $R[pos]$ is in $CandR_i = \frac{\sum_{q=1}^{x} v_q}{N}$.

The expected running time of $SWAP$ in terms of C is: $\binom{C}{x}$ * probability of x nodes getting selected * probability of $(C-x)$ nodes not getting selected * probability of $R[pos]$ is in $CandR_i$ * probability of swap * cost of swap.

The probability of $x = i$ and $R[pos]$ is in $CandR_i$ is:

$$
\begin{aligned}
&= (1/C)^i * (1 - 1/C)^{C-i} * [(v_1/N + v_2/N + \dots + v_i/N) \\
&+ (v_1/N + v_3/N + \dots + v_i/N) + \dots \\
&+ (v_{C-i}/N + \dots + v_C/N)] \\
&= (1/C)^i * (1 - 1/C)^{C-i} * \binom{C-1}{i-1} * (\frac{v_1 + v_2 + \dots + v_c}{N}).
\end{aligned}
$$

$$= (1/C)^i * (1 - 1/C)^{C-i} * \binom{C-1}{i-1}.$$

Therefore, the expected running time (cost) of $SWAP$ is,

$$E_{SWAP} = 1/2 * N * k * \log k * \sum_{i=1}^{C} (1/C)^i * (1 - 1/C)^{C-i} * \binom{C-1}{i-1} = 1/2 * N/C * k * \log k.$$

Expected running cost of $Divcont$ is $\binom{C}{x}$ * probability of x nodes getting selected * probability of $(C - x)$ nodes not getting selected * probability of $R[pos]$ is in $CandR_i$ * cost of $Divcont$. Therefore, the expected running time (cost) of $Divcont$ is:

$$E_{Divcont} = N * k * \sum_{i=1}^{C} (1/C)^i * (1 - 1/C)^{C-i} * \binom{C-1}{i-1}$$
$$= N/C * k.$$

The expected cost of Part 2 becomes:

$$E_{Part_2} = 1/2 * N/C * k * \log k + N/C * k.$$

The expected running time (cost) of Part 1 is = $\binom{C}{x}$ * probability of x nodes getting selected * probability of $(C - x)$ nodes not getting selected * probability of $R[pos]$ is in $CandR_i$ * probability of swap * cost of **DivGetBatch()**. Using similar calculation as above, expected cost of part 1 is:

$$E_{part_1} = 1/2 * N * \sum_{i=1}^{C} (1/C)^i * (1 - 1/C)^{C-i} * \binom{C-1}{i-1} * C = N/2.$$

Expected running time of **Aug-SWAP** algorithm considering both Part 1 and Part 2 computation is:

$$E_{\mathbf{Aug-SWAP}} = 1/2 * N/C * k * \log k + N/C * k + N/2$$
$$+N = \mathcal{O}(N/C * k * \log k + N)$$

Now consider the case when $l > 1$ for **Aug-SWAP**. Probability of selecting a node in first level is $1/m$, given m is the arity of **I-tree**. Probability of selecting a node in second level = probability of selecting that node out of m node in that branch * probability of selecting it's parent = $1/m^2$. Similarly, Probability of selecting a node at leaf node is $1/m^l = 1/C$. As the records are only returned from leaf nodes, the expected probability that $R[pos]$ is in $CandR_i$ does not change for $l > 1$. The running time of **DivGetBatch()** $= \mathcal{O}(m^l) = \mathcal{O}(C)$ also stays same . The rest of the computation does not directly depend on l. As a result, expected running time of **Aug-SWAP** for $l > 1$ is same as before.

□

Worst-case Aug-SWAP. In the worst-case, none of the records are skipped, so the number of swap is $\mathcal{O}(N)$. Therefore, the worst-case running time is: $\mathcal{O}(N*C*k*\log k)$.

Our technical results are summarized in Tables 3.1 and 3.2.

3.6 I-tree

The index is a hierarchical complete tree-like structure [56] that partitions D into multiple groups of records. Each node in **I-tree** consists of a group of similar records. The index structure maintains a higher level aggregate similarity between nodes [3]. **I-tree** is applicable not only to the studied three algorithms, but also to any content-based algorithm that is either based on replacing items in the top-k or building the top-k in an incremental fashion.

[3]Diversity between a pair of records is simply $1 - \textit{similarity}$ between them.

Algorithm 5 Indexing Algorithm *BuildTree(node)*

Inputs: database D of N records, m: arity of the tree, l: number of levels,

Outputs: I-tree, *simMatrixNode*: node-node similarity matrix, *recordMap*: a mapping of all records and their belonging node id for each level.

1: $rootnode \leftarrow N$ records, $y = 0$
2: $nodelist[y] \leftarrow rootnode$
3: **while** $y \leq l$ **do**
4:     **for** *node* **in** $nodelist[y]$ **do**
5:         $cnodes \leftarrow Partition(node, m)$
6:         **I-tree** $[y][node]$.ADDCHILD(cnodes)
7:         $w \leftarrow \bigcup cnodes$
8:         *recordMap[y][r]* $\leftarrow$ *node id* containing record r in y
9:     **end for**
10:     *MinsimMatrixNode[y][i][j]* $\leftarrow$ Use Equation 3.23
11:     *MaxsimMatrixNode[y][i][j]* $\leftarrow$ Use Equation 3.24
12:     $nodelist[y] \leftarrow w$
13:     $y \leftarrow y + 1$;
14: **end while**

3.6.1 Index construction

The input to the indexing step is a $N \times N$ matrix, named *simMatrixRecord*. It represents the similarity scores between every pair of records, r_i and r_j, in the database and two additional parameters, l and m, which are the number of levels and arity of the tree, respectively. The output is a complete m-ary tree with l levels, referred to as **I-tree**.

The indexing algorithm *BuildTree* (Algorithm 5) partitions (refer to the Subroutine *Partition*) the records. It also maintains additional data structures that contain similarity scores between nodes for efficient query processing. An example of

a two-level index tree is shown in Figure. 3.2. At the first level, *BuildTree* creates a root node containing all N records and m children of the root node. From the point of abstraction, it is not important at this stage to describe how the data is partitioned. Basically, the goal is to keep similar records together while separating non-similar ones. There are multiple off-the-shelf techniques such as clustering and graph partitioning to carry out this task.

In our implementation, we use the popular k-means algorithm [47] for partitioning. The algorithm repeats the partitioning procedure until it reaches l levels. Therefore, **I-tree** contains a total of C nodes such that:

$$C = \sum_{i=0}^{l} m^i = \frac{m^{l+1} - 1}{m - 1} = \mathcal{O}(m^l) \quad (3.22)$$

Inside **I-tree**, additional data structures are maintained:

a. A *recordMap* of size $N \times l$ that maps the id of a record with the id of its node in each level from $1 \dots l$. b. *MinsimMatrixNode* and *MaxsimMatrixNode* that contain inter-node minimum and maximum similarities between any two nodes in the same level, respectively. Particularly, for two nodes n and n' in level y, *MinsimMatrixNode* and *MaxsimMatrixNode* contain:

$$MinsimMatrixNode[i, j] = Min_{r \in i, r' \in j} sim(r, r'), \quad (3.23)$$

$$MaxsimMatrixNode[i, j] = Max_{r \in i, r' \in j} sim(r, r'), \quad (3.24)$$

where, $r \in n, r' \in n'$. Figure 3.1 contains these scores for 3 nodes of our running example.

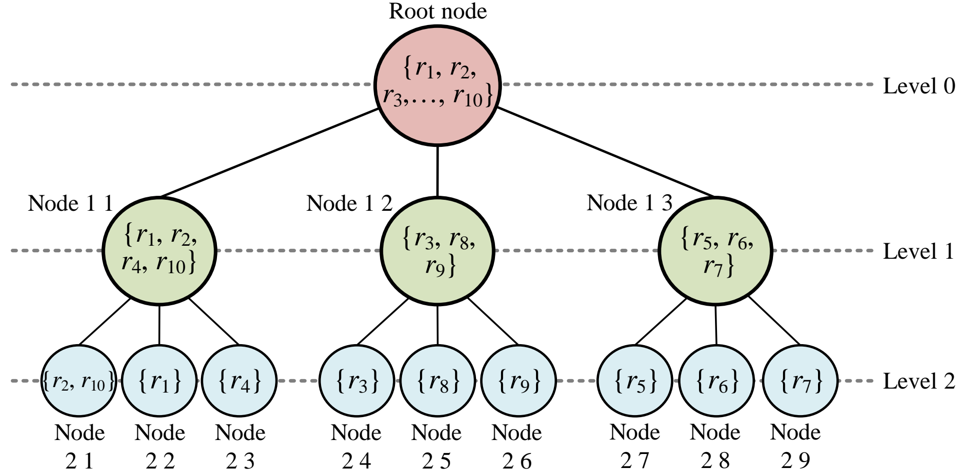


Figure 3.2 I-tree.

3.6.2 Index maintenance

Even for a single insertion or deletion, **I-tree** requires the following two activities: a. insertion/deletion of that record from/into **I-tree**; b. updating *MinsimMatrixNode* and *MaxsimMatrixNode*, if these insertion/deletion require updating the minimum and maximum similarity scores between nodes. One can easily see that (a) could be achieved in a constant time when l =1 and $\mathcal{O}(l)$ when l greater than 1. However, a single insertion/deletion may require as many as $2 \times (C - 1)$ updates in these two matrices.

Batch Update We study how to maintain **I-tree** considering both insertions and deletions.

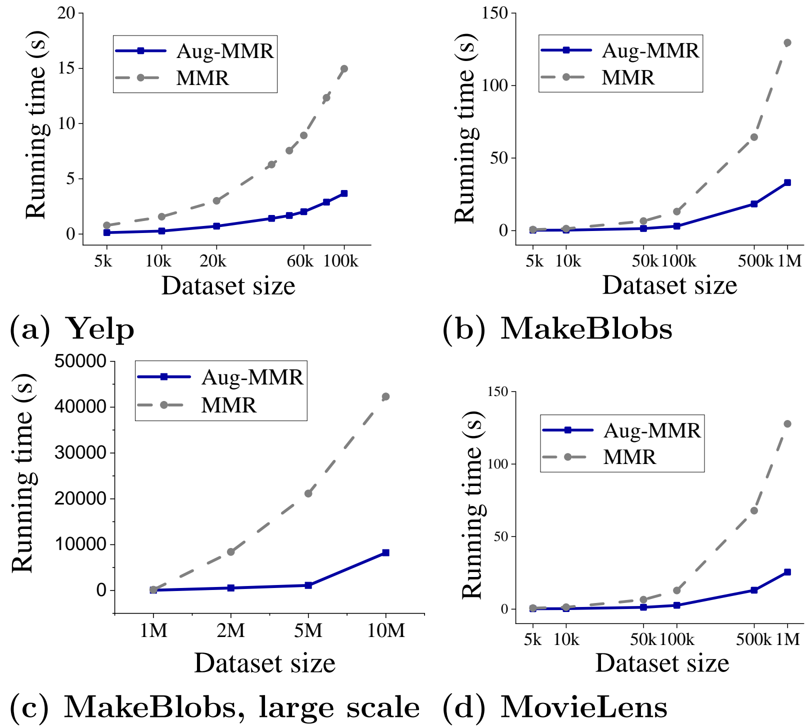
Batch Deletion. Let us assume a batch of R records are to be deleted from **I-tree**. The process deletes these R records one by one and then checks how many entries in *MinsimMatrixNode* and *MaxsimMatrixNode* need update (if the deleted records contribute to these aggregate values, then that require updates in those two matrices, else not). The overall process takes $\mathcal{O}(|Y| \times C \times N)$ time.

Batch Insertion. This problem is more complicated. If the records are inserted arbitrarily inside **I-tree**, then, each insertion may potentially cause a total of $2 \times (C -$

1) updates in the *MinsimMatrixNode* and *MaxsimMatrixNode* data structures. This is *the leading computational cost of batch insertion*. Moreover, when a batch of records are inserted, it is possible to have multiple records to get inserted inside the same node, and that should not be double-counted in the process. Finally, one needs to insert the records to those nodes, such that the aggregates stored in *MinsimMatrixNode* and *MaxsimMatrixNode* remain "tight" to enable effective pruning. These nuances are explored in formalizing the batch insertion problem.

Problem Definition 2. (Batch Insert.) *Let* $Minsim$ $MatrixNode[i,j]$ *(similarly* $MaxsimMatrixNode[i,j]$*) denote the value after* $|Y|$ *insertions at the* $[i,j]$*-th entry at the* MinsimMatrixNode *(similarly* $MaxsimMatrix$ $Node$ *matrix). Let* $Minsim_{ij}$ *and* $Maxsim_{ij}$ *be two binary variables, such that which* $Minsim_{ij} = 1$ *(similarly* $Maxsim_{ij}$*) , if it requires an update after insertions,* 0 *otherwise. Our goal is to insert a batch of records* Y *such that, it minimizes* $\sum_{i,j} Minsim_{ij} + \sum_{i,j}$ $Maxsim_{ij}$*, i.e., the total number of updates in these two matrices.*

Algorithms. We present an integer programming-based solution **OPTMn** for solving the batch insert problem. While **OPTMn** indeed produces the optimal solution, due to its exponential nature, it does not scale to a very large dataset considering a large number of insertions. As an alternative, we present **GrMn** a greedy heuristic algorithm which makes greedy choices and indirectly attempts to minimize the number of updates in *MinsimMatrixNode* and *MaxsimMatrixNode* matrices. The idea is to make a greedy decision by inserting each of the incoming records to that node which it is closest to (based on the underlying similarity measure) and then check if that insertion requires any updates in *MinsimMatrixNode* and *MaxsimMatrixNode* matrices. The running time of this algorithm is $\mathcal{O}(|Y| \times N)$.



(a) Yelp **(b) MakeBlobs**

(c) MakeBlobs, large scale **(d) MovieLens**

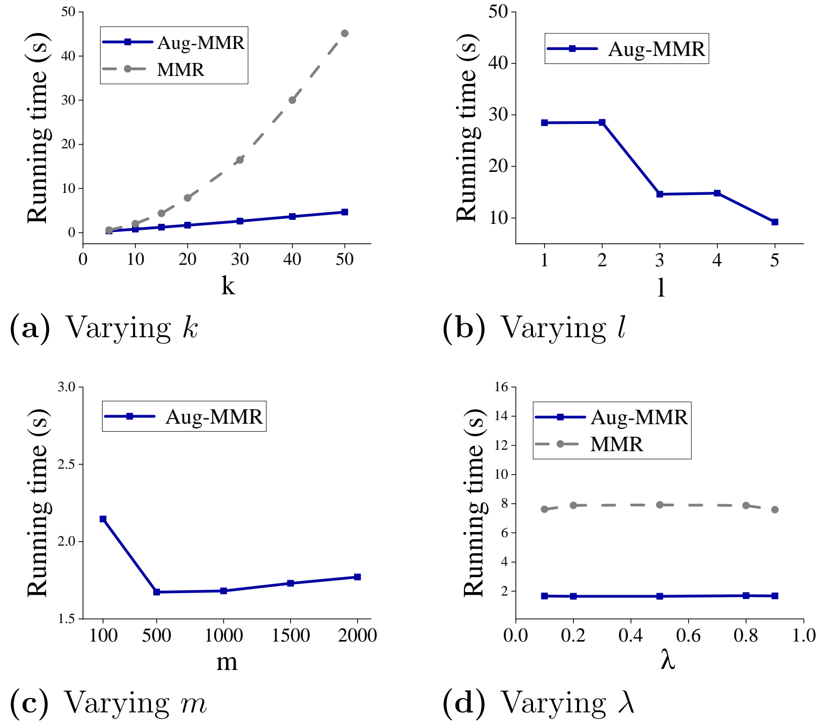
Figure 3.3 Aug-MMR vs MMR scalability.

3.7 Experimental Evaluation

Our experimental evaluations have three primary goals. First, we analyze if the augmented algorithms return identical results to their original counterparts using multiple large-scale datasets. Second, we examine the efficiency and scalability of the augmented algorithms and compare them with multiple baselines. Finally, we empirically study the cost of building and maintaining **I-tree**. For brevity, we present a subset of results that are representative.

Experimental setup. All algorithms are implemented in Python 3.8. All experiments are conducted on a cluster server OSL machine with 32GB RAM memory, OS: Scientific Linux release 7.8 (Nitrogen), CPU: Intel(R) Xeon(R) CPU E3-1245 v6 @ 3.70GHz. Obtained results are the average of three separate runs. [4]

[4]The code and data could be found at https://github.com/MouinulIslamNJIT/divGetBatch, Retrieved on 4/7/2023



(a) Varying k (b) Varying l

(c) Varying m (d) Varying λ

Figure 3.4 Aug-MMR vs MMR varying parameters.

Table 3.6 Dataset Statistics

Dataset	Size	#Total features	#Features used	Dataset type
Yelp	112,686	12	3	Real
MovieLens	1,000,209	3	2	Real
MovieLens non-metric	8,453	3	2	Real
UCI Gas dataset	13,911	128	128	Real
MakeBlobs	10,000,000	varied	20	Synthetic

Table 3.7 Aug-MMR vs MMR Running Time (s) on **MakeBlobs** with $l = 2$, $m = 6$

	Dataset Size			
Algorithm	**5k**	**10k**	**50k**	**10k**
Aug-MMR	4.33	8.69	43.57	306.11
MMR	19.77	40.16	197.28	1206.90

Diversity and Similarity. We use normalized Euclidean distance (dist) as diversity to validate our designed solutions in the geometric space, Cosine similarity [47] in general metric space. For non-metric distance, we use Movielens datasets and quantify the diversity between a pair of movies as the number of users who have rated either of these two movies but not both. We additionally use an arbitrary diversity function generated synthetically on Makeblobs dataset, such that it does not satisfy triangle inequality. Thus, diversity values are atomic for the last two cases, and are not derived from the feature vectors. For all these cases, $sim = 1 - dist$.

Query selection. In our experiments, queries are chosen randomly.

Performance Measures. We measure precision@k [47] for qualitative analysis. Efficiency of the proposed method is demonstrated with $|CandR|/N \times 100$, pruning $= 1 - |CandR|/N \times 100$, as well as by presenting the running times of the algorithms in seconds and computing *speedup* as follows:

$$speedup = \frac{T_{original-algorithm}}{T_{augmented-algorithm}} \tag{3.25}$$

where T denotes running time in seconds. Finally, we present time to build **I-tree** and the space required for that.

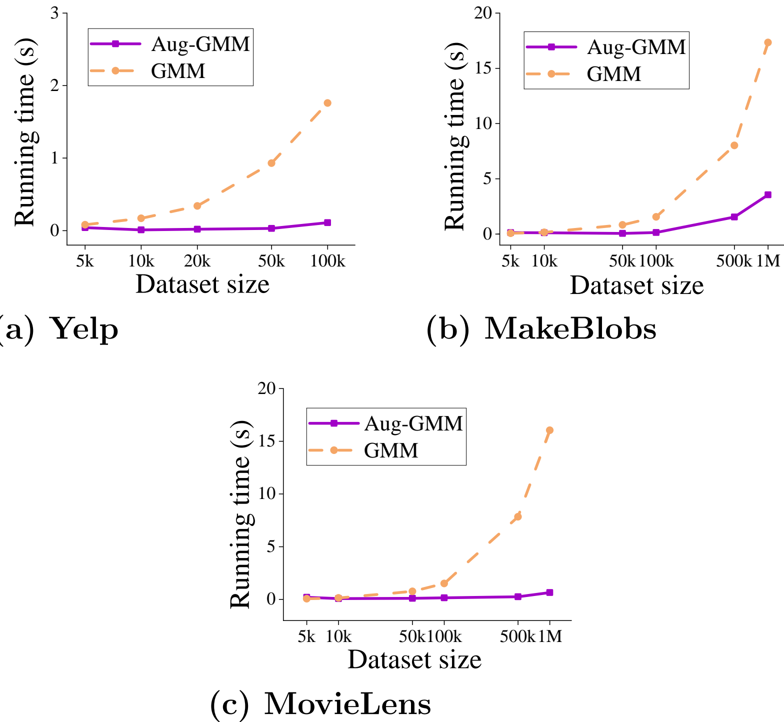


Figure 3.5 Aug-GMM vs GMM scalability.

Datasets. Experiments are conducted on five datasets, four real and one publicly available synthetic data. For real datasets, we use **Yelp**[5], **UCI Gas** dataset [6] that is high dimensional, **MovieLens** 1M records, and **MovieLens non-metric** dataset[7]. For synthetic data, we use **MakeBlobs** from the sklearn package.[8] An overview of the datasets is given in Table 4.5.

3.7.1 Baselines

In this section, we introduce diversity-based algorithms and index structure baselines that we compare to our proposed solutions.

[5] `https://www.yelp.com/dataset/documentation/main`, Retrieved on 4/7/2023
[6] `https://archive.ics.uci.edu/ml/datasets/gas+sensor+array+drift+dataset`, Retrieved on 4/7/2023
[7] `https://grouplens.org/datasets/movielens/`,Retrieved on 4/7/2023
[8] `https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html`,Retrieved on 4/7/2023

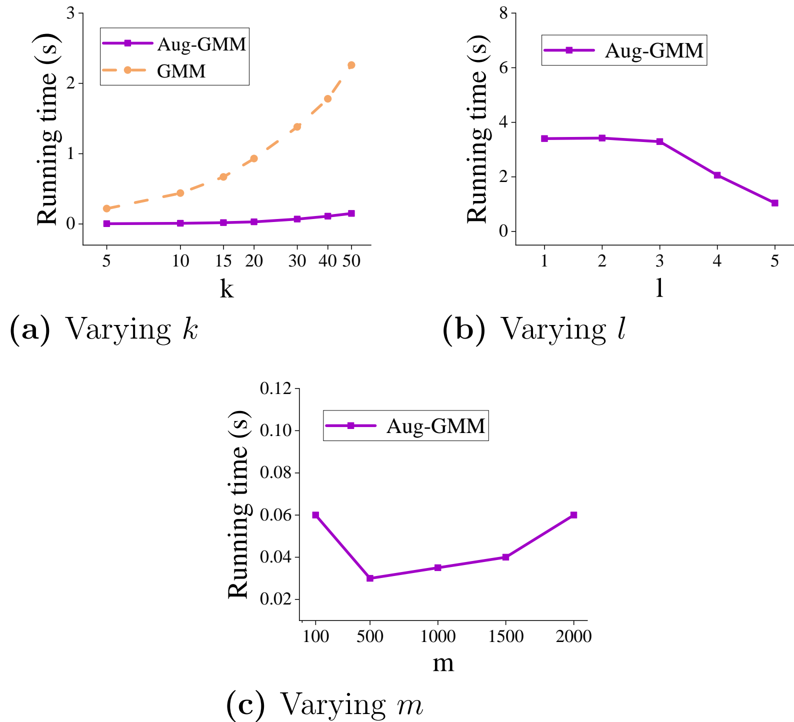


Figure 3.6 Aug-GMM vs *GMM* performance varying parameters.

Diversity Baselines For diversity-based methods, three representative algorithms are implemented.

MMR [23]: computes an objective score based on two parameters: relevance to the query and diversity with other records. As shown in Equation (3.1), they are combined in a linear expression with a λ coefficient. The algorithm repeats this computation k times to produce top-k.

GMM [44]: finds the k most diverse records by selecting the maximum of minimum distances between undiscovered records and previously selected ones at each iteration (Equation 3.10). Like MMR, it also iteratively builds the top-k set.

SWAP [95]: This greedy algorithm first finds the initial top-k records, then greedily interchanges records that are part of the current top-k with the ones that are remaining, if the *swap* improves diversity contribution (Equation 3.15).

SPP [38]: Space Partitioning and Probing (SPP in short) is an algorithm that minimizes the number of accessed objects while finding exactly the same result as

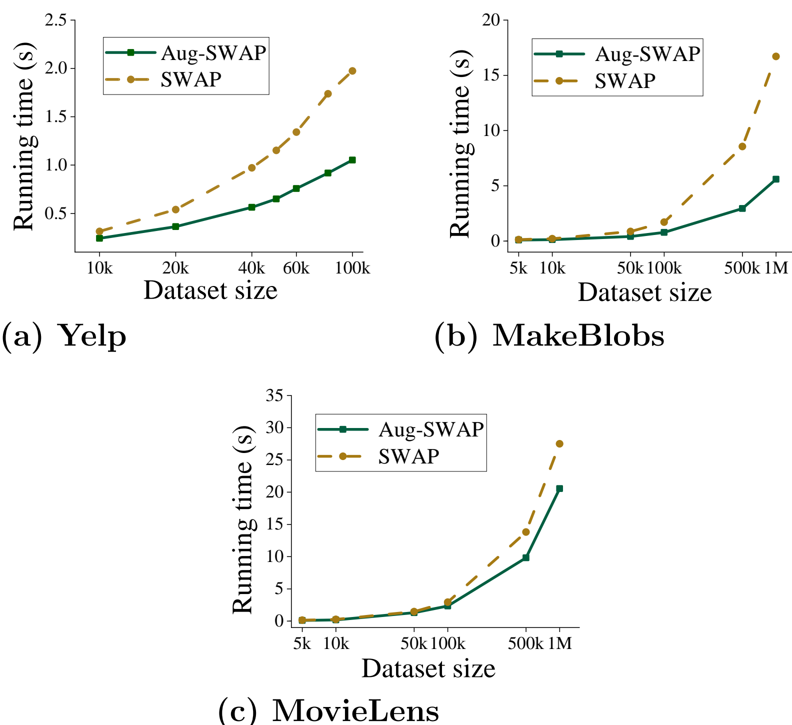


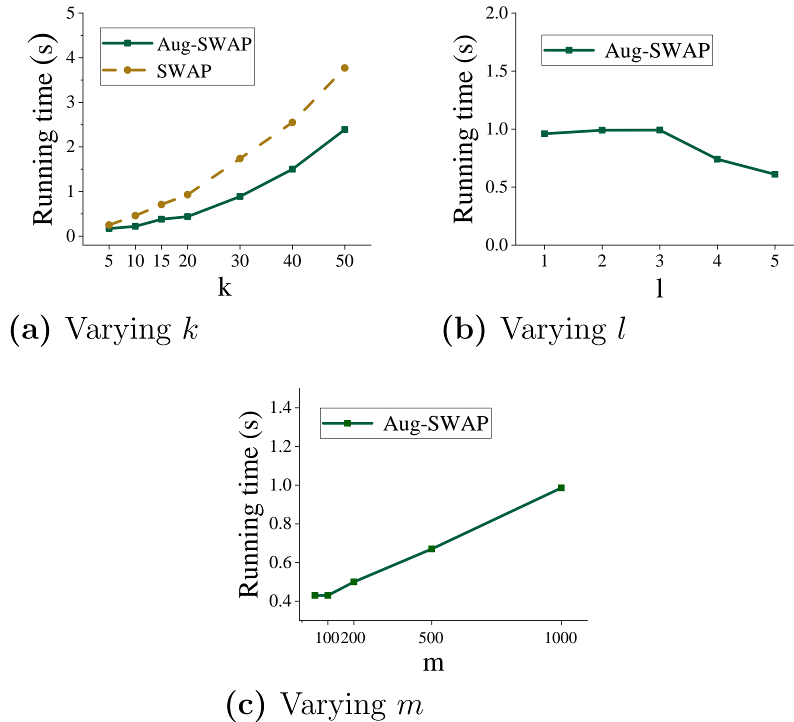
Figure 3.7 Aug-SWAP vs SWAP scalability.

MMR. *SPP* belongs to a family of algorithms that rely only on score-based and distance-based access methods, and does not require retrieving all the relevant objects. SPP is designed only for the geometric space.

Index Structure Baselines We implement three additional baselines to compare against **I-tree**. These indexing techniques are limited to metric space, and can not be applied on arbitrary diversity function not satisfying triangular inequality.

KD-tree [16]:KD-tree is a multidimensional Binary Search Tree. The tree is created by bisecting each dimension and finding the median. *KD-tree* can perform searches in multidimensional space for efficient nearest neighbor search.

Ball-tree [58]: Ball-tree is a binary tree in which every node defines a D-dimensional hypersphere or ball, containing a subset of the points to be searched. Each node in the tree defines the smallest ball that contains all data points in its



(a) Varying k (b) Varying l

(c) Varying m

Figure 3.8 Aug-SWAP vs SWAP varying parameters.

subtree. This gives rise to the useful property that for a given test point t outside the ball, the distance to any point in a ball B in the tree is greater than or equal to the distance from t to the surface of the ball. *Ball-tree* only supports binary splits.

The arity of the tree in both *KD-tree* and *Ball-tree* is fixed to 2.

M-Tree [25]: M-tree is similar to *Ball-tree*, but supports multiple splits. Every node n and leaf lf residing in a particular node N is at most distance r from N, and every node n and leaf lf with node parent N keeps the distance from it. It also has the similar property of *Ball-tree*, which is for a given test point t outside the node, the distance to any point in a node in the tree is greater than or equal to the distance from t to the surface of the node.

We are incorporating Node-Node distance matrix to these baseline tree index structures so that they can be used for **I-tree** API.

Cover-Tree [18]: Another popular indexing structure is cover tree which is used to enable efficient nearest neighbor search in metric space. To be able to work with

DivGetBatch(), the indexing technique must work in a fashion that the parent nodes of the index structure (in this case a tree) covers the records that are present in their sub-tree. This allows us to effectively maintain the inter-diversity bounds across the nodes and when a node gets pruned, all its children also does. Contrarily, in a cover tree, only the leaf nodes together contain and cover all the records and no other intermediate/ higher level nodes does. Therefore, it is not obvious how to adapt this indexing technique and integrate it inside our proposed access primitive.

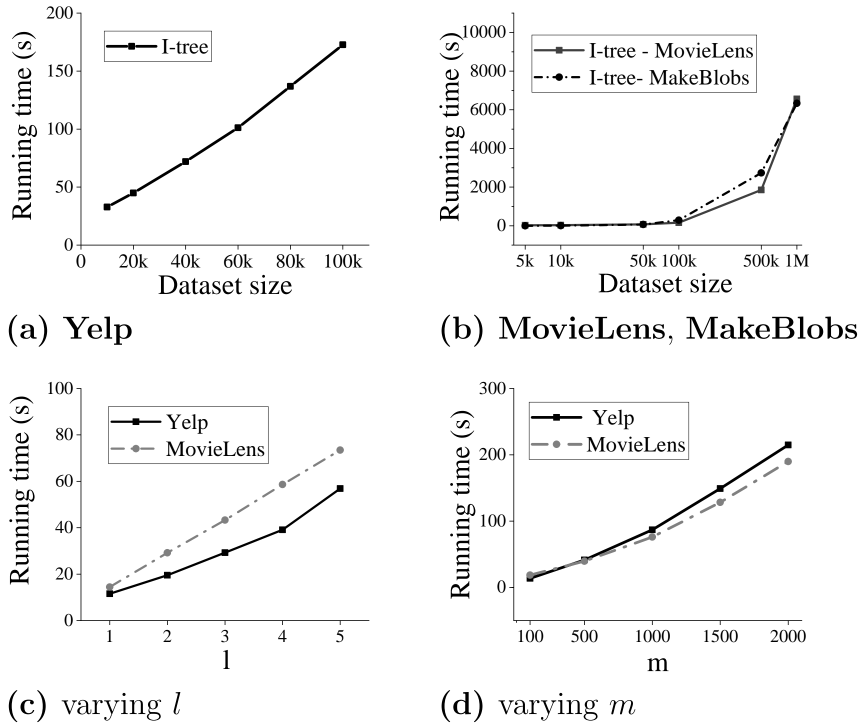
Index Maintenance Baselines **OPTMn** and **GrMn** are compared with two baselines.

NonIncrMn *Algorithm:* In **NonIncrMn**, **I-tree** is built from scratch after every $|Y|$ insertions. **NonOlMn** *Algorithm:* This algorithm makes a local decision to insert each record based on Problem 2, without accounting for overlapping updates inside the same node in **I-tree**.

3.7.2 Summary of results

Our first set of experiments verify that our results from all three augmented algorithms are *identical* to their original counterparts. We measure precision@k [47] for different k, and our empirical results obtain 100% precision score.

Our next set of experimental results demonstrate that the running time of the augmented algorithms are consistent with our theoretical analyses. We achieve a 19× and 24× speedup for **Aug-MMR** and **Aug-GMM**, on **Makeblobs** 10M and **MovieLens** 1M data, respectively. We achieve a 3× speedup for **Aug-SWAP** on **MakeBlobs** 1M dataset. These results corroborate that our proposed framework is suitable to scale on large datasets. We also show that **I-tree** works on any arbitrary distance functions while other baselines are designed for only metric distance functions. We have conducted experimental analysis on two different non-metric distance functions (one obtained from the real data), these experimental



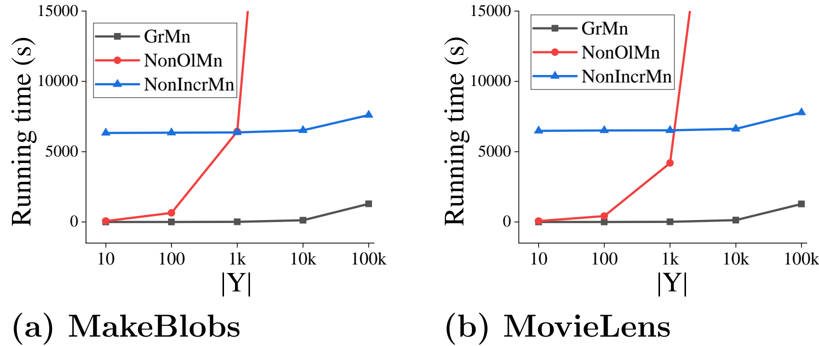
(a) Yelp **(b) MovieLens**, **MakeBlobs**

(c) varying l **(d)** varying m

Figure 3.9 I-tree construction time.

results demonstrate that **Aug-MMR** attains 82% pruning compared to the baseline solutions, resulting in about 2.7 times speed up on an average. On the other hand, the results obtained from high dimensional UCI Gas dataset demonstrate that the proposed framework is still effective even in higher dimension, as **Aug-MMR** attains about 1.7 speed up on an average.

Figures 3.11 demonstrate the index construction and the query processing time trade-off of **I-tree** and we compare that with our implemented baseline indexes, KD-tree, Ball-Tree, M-Tree. These results convincingly demonstrate that **I-tree** enables the fastest query processing time, while requiring comparable index construction time. The results demonstrate that **I-tree** is always more than 18$\times$ faster in query processing and as much as 170$\times$ faster for certain configurations. For preprocessing, it is always more than 1.5$\times$ faster and at times it is more than 20$\times$ faster. We also present $|CandR|$ percentage and pruning percentage of **I-tree**



(a) MakeBlobs **(b) MovieLens**

Figure 3.10 I-tree maintenance time varying $|Y|$.

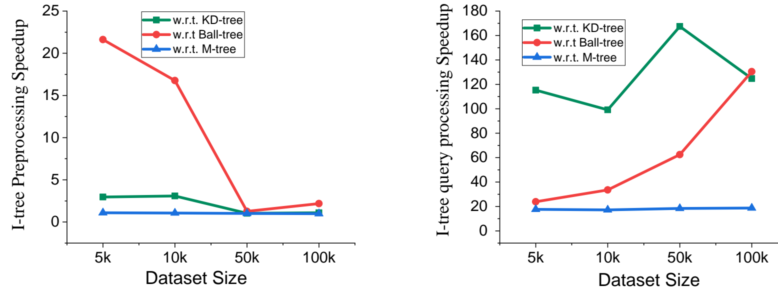
compared to other index baselines in Tables 3.9 and 3.10 which shows that **I-tree** outperforms all baselines with having 90% pruning.

The results convincingly demonstrate that **I-tree** is lightweight to compute and space efficient (for the largest dataset, it takes 109 minutes to build the index, which is acceptable because it is done offline and only once). Finally, we demonstrate that our proposed solution **OPTMn** is an ideal choice for incremental index maintenance, while the greedy heuristic **GrMn** is highly scalable while being not too inferior from the optimal solution **OPTMn** qualitatively. **GrMn** takes 22 minutes to insert $100k$ data into 1M dataset, while building **I-tree** from scratch is unrealistic as **NonIncrMn** takes 2 hours.

3.7.3 Quality analysis

The goal of these experiments is to empirically validate if the augmented algorithms produce the same results as their original counterparts. Additionally, we present how effective **DivGetBatch()** is in pruning records by presenting the size of $CandR$.

We have calculated precision@k while varying k from 10 to 50, considering the original and augmented algorithms. We obtain the precision@k equal to 100% always.



(a) I-tree Index Preprocessing speedup w.r.t baselines

(b) I-tree Query Processing speedup w.r.t baselines

Figure 3.11 Index Construction and Query Processing time for tree baselines and **I-tree**.

3.7.4 Scalability analysis

We run two types of scalability experiments. (i) demonstrate the efficacy of the augmented diversification algorithms and compare them appropriately with the baselines; (ii) demonstrate the efficacy of the indexing technique - present index construction and maintenance time, and compare them appropriately with the baselines. Additionally, we also present the memory requirements of **I-tree**. We analyze these effects by increasing dataset size and other pertinent parameters.

Augmented Diversification Algorithms We first vary dataset size, then additional parameters that impact the query processing time. To demonstrate efficacy, we present two things. (1) The percentage of remaining records returned by **DivGetBatch()**, which is which is $|CandR|/N \times 100$ and pruning $(1 - |CandR|/N \times$ 100. (II) Query processing time in seconds.

Effectiveness in Pruning. In Table 3.8, we present the number of remaining records returned by **DivGetBatch()**, which is $|CandR|$ using **MovieLens** dataset. We can observe that there is a remarkable reduction compared to the original dataset. For example, **Aug-MMR** returns only 814 records. The biggest number is for **Aug-SWAP** with 66513 records, but still returning only 6% of the records.

Table 3.8 $|CandR|$ Percentage Returned by DivGetBatch() on MovieLens

	Dataset Size					
Algorithm	**5k**	**10k**	**50k**	**100k**	**500k**	**1M**
Aug-MMR	13%	5.21%	0.56%	0.09%	0.08%	0.08%
Aug-GMM	59.96%	15.48%	4.16%	2.67%	0.31%	0.4%
Aug-SWAP	14.96%	28.11%	10.07%	48.74%	9.27%	0.66%

Table 3.9 $|CandR|$ Percentage Returned by DivGetBatch() Using Different Index Structures for Aug-MMR on MakeBlobs

	Dataset Size			
Algorithm	**5k**	**10k**	**50k**	**100k**
I-tree	10%	10%	10%	10%
KD-tree	96.72%	96.72%	96.87%	97.34%
Ball-tree	96.7%	95.62%	96.56%	96.56%
M-tree	97.92%	97.19%	98.32%	98.07%

Table 3.9 and Table 3.10 show $|CandR|$ and pruning percentage returned by **DivGetBatch()** for **Aug-MMR** algorithm using different index structures and MakeBlobs dataset. We can see that by fixing $C = 32$, KD-*tree*, *Ball-tree*, and *M-tree* pruning are below 5%, while **I-tree** pruning considerably outperforms all baseline which is 90%.

Effectiveness in Number of Accesses. In order to perform a fair comparison between our augmented algorithms and SPP, we compare the number of I/O accesses SPP does and present that number for **Aug-MMR** (SPP is designed to optimize that access). We calculate the number of accesses in **DivGetBatch()** by counting the distinct records present in $CandR$ in k rounds. The results are presented in Table 3.11. We can see that **Aug-MMR** has less number of access. For example on 100k data, **I-tree** has 2799 number of access while SPP has 26521 number of access.

Table 3.10 Pruning Percentage by DivGetBatch() Using Different Index Structures for Aug-MMR on MakeBlobs

	Dataset Size			
Algorithm	**5k**	**10k**	**50k**	**100k**
I-tree	90%	90%	90%	90%
KD-tree	3.3%	3.3%	3.1%	2.6%
Ball-tree	3.3%	4.3%	3.4%	3.4%
M-tree	2%	2.8%	1.6%	1.9%

Table 3.11 Number of Access Percentage for Aug-MMR and SPP on MakeBlobs

	Dataset Size			
Algorithm	**5k**	**10k**	**50k**	**100k**
I-tree	10%	10%	5.2%	2.79%
SPP	20.44%	9.57%	27.31%	26.52%

Varying Dataset. Figures 3.3, 3.5, and 3.7 compare the running times of our three augmented algorithms and their baselines using our three datasets. As N increases, the running times of each algorithm and its baseline increase, but we observe that our algorithms are consistently faster and they scale significantly better. Figure 3.3 shows **Aug-MMR**'s scalability on all three datasets. We fix m to 1000, $k = 20$ and $l = 1$ for all dataset sizes while N is increased from 5000 up to 1M. We can see that on **MovieLens**, varying N from 5000 to 1M, **Aug-MMR** is 5$\times$ faster than *MMR*. Figure 3.5 shows **Aug-GMM** 's scalability. On **MovieLens**, varying N from 5000 to 10M, **Aug-GMM** is 24$\times$ faster than *GMM*. Consistent with the theoretical analysis, **Aug-GMM** is faster than **Aug-MMR** for the same settings because **Aug-MMR** has an additional k term in the expected cost equation. Figure 3.7 shows **Aug-SWAP** 's scalability on all three datasets. For the 1M data of **MakeBlobs** we obtain a 3$\times$ speedup over *SWAP*. We obtain a 1.33$\times$ speedup for **Movielens** because the total number of swaps in **MovieLens** are higher.

Table 3.12 Index Comparisons

Index	Metric Functions	Non metric Functions	90% Pruning
I-tree	✓	✓	✓
KD-tree [16]	✓	×	×
Ball-tree [58]	✓	×	×
M-tree [25]	✓	×	×

Table 3.13 Aug-MMR vs MMR Running Time on MakeBlobs 100k Records

	Distance function		
Algorithm	**Euclidean**	**Cosine**	**Non-metric**
Aug-MMR	3.08	4.64	13.06
MMR	13.12	15.36	15.27

We also measure the scalability of **Aug-MMR** compared to MMR using large scale data sizes of 2M, 5M, and 10M using makeBlobs dataset. The results are shown in Figure 3.3(c) in which with $m = 1000$ and $l = 1$, we have up to 19× speedup.

Moreover, we run **Aug-MMR** on high-dimensional euclidean distance considering more number of features using 1M and 2M makeBlobs dataset. for 1M data, 1M and 20 features, MMR takes 12492.64 (s), and **Aug-MMR** takes 2817.14 (s). For 2M data and 20 features, MMR takes 25812.43 9 (s), **Aug-MMR** takes 6317.20 (s) which in both case show 4× speedup.

Additionally, Figure 3.12 presents the scalability of the proposed **Aug-MMR** algorithm compared to MMR using UCI Gas dataset with 10k records and 128

Table 3.14 Aug-MMR vs MMR on Movielens Non-metric Data

Algorithm	Running time (s)	Average Pruning
Aug-MMR	0.19	82.66%
MMR	0.52	0

Table 3.15 I-tree Maintenance on MakeBlobs 10k Records

$\lvert Y \rvert$	Algorithm	# updates	running time (s)
10	**OPTMn**	14	3.59
	GrMn	76	0.007
	NonOlMn	14	0.29
	NonIncrMn	2446	1.30
100	**OPTMn**	59	512.42
	GrMn	76	0.05
	NonOlMn	142	2.97
	NonIncrMn	2447	1.44
1000	**OPTMn**	59	18768.68
	GrMn	76	0.43
	NonOlMn	1068	34.58
	NonIncrMn	2449	1.45

features. We set $\lambda = 0.8$ and vary k from 10 to 25. By increasing k, **Aug-MMR** shows more scalability than MMR. **Aug-MMR** is about 1.7 times faster than the baseline implementation.

Finally, we run **Aug-MMR** on l more than 1 to show the efficiency of our proposed algorithm using multi-level **I-tree**. Table 3.7 shows that for l=2, **Aug-MMR** speedup is almost $4\times$ for all dataset sizes.

Varying Parameters. We study the effect of different parameters on running time. Some parameters belong to the offline indexing algorithm, such as the number of levels (l) and arity of **I-tree** (m) and the total number of nodes (C). Other parameters are part of the online augmented algorithms. For example, k for the number of returned records and λ coefficient for **Aug-MMR** . In Figures 3.4, 3.6, 3.8, we vary parameters using **Yelp** dataset with a fixed size of 50000 records. In our experiment, optimum

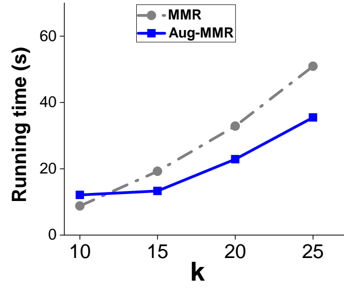


Figure 3.12 Aug-MMR vs MMR running time on UCI Gas data.

parameter settings for offline indexing are obtained by performing multiple runs and selecting the best. The index created using those parameter settings can be used in multiple runs of the online phase.

Varying k**.** Figures 3.4(a), 3.6(a), and 3.8(a) present how running time changes as we vary k from 5 to 50 for different baselines while fixing l, m, and λ to $1,500$, and 0.8, respectively. The running time increases quadratically for MMR and **Aug-MMR**, linearly for GMM and **Aug-GMM**, and in $\mathcal{O}(k * \log k)$ fashion for $SWAP$ and **Aug-SWAP**. These results are as consistent with our theoretical analysis, because of the presence of k^2 term in the MMR and **Aug-MMR**'s expected cost, k in GMM and **Aug-GMM**'s expected cost, and $k * \log k$ of that of $SWAP$ and **Aug-SWAP**. **Varying** m**.** Figures 3.4(c), 3.6(c), and 3.8(c) show the impact of varying m on the running time of the three algorithms. While varying m, we fix other parameters: $k = 20$, $l = 1$. The choice of m depends on the distribution of the dataset. As we increase m, the bounds for augmented algorithms become tighter while time for **DivGetBatch()** increases. We can see that there is a drop in running time and which indicates the optimum value for m for these three algorithms. For example, in **Aug-MMR** and **Aug-GMM**, the ideal value is $m = 500$ and for **Aug-SWAP**, it is $m = 100$.

Varying l**.** Figures 3.4(b), 3.6(b), and 3.8(b) show the impact of varying l on the running time of the three algorithms. We fix other parameters: $k = 20$,

Table 3.16 I-tree Maintenance Algorithm GrMn vs Construction from Scratch Algorithm NonIncrMn Running Time on MakeBlobs 10k Records

$\|Y\|$	Insertion Algorithm	Preprocessing time-offline (s)	query processing time-online (s)
10	**GrMn**	0.007	1.25
	NonIncrMn	1.30	0.55
100	**GrMn**	0.05	1.33
	NonIncrMn	1.44	0.60
1000	**GrMn**	0.43	1.96
	NonIncrMn	1.45	0.80
10000	**GrMn**	1.02	8.18
	NonIncrMn	4.65	1.61

and setting m to 2. C, the total number of nodes in **I-tree** becomes $2, 7, 15, 31, 63$, respectively for $l = 1, 2, 3, 4, 5$. In general, by fixing m and increasing l, C increases, and overall running time decreases. This is consistent with our theoretical analysis, as the expected running time contains a $1/C$ term.

Varying λ. Figures 3.4(d), 3.6(d), and 3.8(d) show that varying λ in *MMR* and **Aug-MMR** does not significantly change the running time. We have fixed $k = 20$, $l = 1$, and $m = 500$. The result is evident by observing the expected cost equations of *MMR* and **Aug-MMR** algorithms which do not contain a λ term. Though MR scores changes with λ, it has very little effect on the overall running time of *MMR* and **Aug-MMR** algorithms.

Varying diversity Functions Table 3.13 shows the results for **Aug-MMR** compared to MMR using different distance measures: euclidean distance measure, cosine similarity as general metric, and a non-metric distance function. Using 100k data from MakeBlobs dataset and m= 1000, $l = 1$ and number of features = 2, we can see that **Aug-MMR** performs $4\times$ better than MMR using both euclidean and cosine similarity metrics. For non-metric arbitrary distance function, the distance between records do not satisfy triangular inequality. Using this method, we see 15%

improvement, since the relevance and diversity scores are created arbitrarily and the result depends on the data distribution.

Table 3.12 shows overall comparison for **I-tree** and other baselines. SPP uses *KD-tree* as its index so we did not add it to the table. We can see that, unlike other baselines, **I-tree** can be used in non-metric functions and outperforms with 90% pruning of the original dataset.

Table 3.14 shows the results for **Aug-MMR** compared to MMR using non-metric distance function computed from MovieLens non-metric dataset. The total number of movies is 8,453, $\lambda = 0.8$, and $k = 20$. The diversity between a pair of items (movies) is calculated as the number of users that have rated either of those movies, but not both. Table 3.14 demonstrates that **Aug-MMR** outperforms *MMR* with 82.66% pruning of the original dataset, resulting in about 2.7 times speed up on an average.

Index construction and maintenance

Comparison with Baselines - Index Construction vs. Query Processing. In these set of experiments, we compare the index construction and query processing time trade-off of **I-tree** and compare that with of *KD-tree*, *Ball-tree*, and *M-tree* considering **Aug-MMR**. We adapt k-means and k-medoids [47] for building **I-tree** with number of iterations set to 300. The dataset that is used in this experiments is MakeBlobs. Figure 3.11 presents the **I-tree** speedup compared to other baselines for index preprocessing and query processing time. The results demonstrate that **I-tree** is always more than 18× faster in query processing and as much as 170× faster for certain configurations. For preprocessing, it is always more than 1.5× faster and at times it is more than 20× faster.

Index Construction. Now that it is obvious that **I-tree** outperforms the other indexing baselines, we further profile its efficacy.

In Figures 3.9(a) and (b), we vary dataset size and fix other parameters, $m = 1000$, $l = 1$. As we can observe in Figure 3.9(a), on the 100K **Yelp** dataset, indexing time is 172.69 seconds. In Figure 3.9(b), indexing time is 105 minutes on the 1M **MakeBlobs** dataset, and 109 minutes on the 1M **MovieLens**. Figures 3.9(c) and (d) show that the running time increases linearly when parameters m and l are systematically increased. In Figure 3.9(c), by varying l, we fix dataset size to 50000, and m to 2 (since $C = m^l$, by increasing l, the total number of nodes will increase). Finally, in Figure 3.9(d), we vary m, while fixing dataset size to 50000 and $l = 1$. These figures demonstrate that the preprocessing time increases linearly with varying parameters. **I-tree** takes 253 MB of space for 1M data with $m = 1000$ and $l = 1$.

Index Maintenance. For analyzing the index maintenance, we use two datasets, **MakeBlobs** and **MovieLens**. We compare **OPTMn** and its efficient counterpart **GrMn** with the baselines **NonOlMn**, and **NonIncrMn**. As expected, **OPTMn** has the least number of updates, but due to its inherent exponential nature, it does not scale beyond $10k$ dataset size with more than $|Y| = 1000$ records. Table 3.15 presents these results. We also see **GrMn**, even though not the optimal one, but stays consistently close to **OPTMn**. This table also shows that **GrMn** is better than the baselines in both running time and number of updates.Figures 3.10(a) and (b) present running time comparisons on very large datasets. **GrMn** is highly scalable, and the other two baselines take more time than **GrMn**. These results corroborate that **GrMn** is a suitable alternative to solve the index maintenance problem.

Incremental Index Maintenance vs Maintenance from Scratch. Table 3.16 shows comparison between **GrMn** and **NonIncrMn** index update algorithms. We present index preprocessing time in the offline phase, and query processing time in

the online phase for the **Aug-MMR** algorithm. Clearly, **GrMn** requires smaller preprocessing time and higher query processing time compared to **NonIncrMn**. As it could be seen from Table 3.16, with 10,000 updates, the query processing time of **GrMn** becomes almost 5× slower than that of **NonIncrMn**. Contrarily, the preprocessing time of **GrMn** is about 4.5× faster than that of **NonIncrMn** at that setting. Since query processing time is more important and must be optimized, it seems, for 10,000 updates, it is better to build the index from scratch instead of maintaining it incrementally.

3.8 Conclusion

We propose an access primitive **DivGetBatch()** to expedite diversification algorithms while returning their exact top-k results. We present a computational framework to develop **DivGetBatch()** that contains a pre-computed index structure **I-tree** and describe how to rewire popular diversification algorithms using **DivGetBatch()**. Unlike existing indexes that primarily work on vector spaces (assuming the records have co-ordinates), we consider the records to be atomic as opposed to a collection of vectors. We make rigorous theoretical analysis of the exactness and running times of the augmented algorithms. We present principled solutions to maintain **I-tree** under batch updates. Our experiments on large real-world datasets corroborate our theoretical analysis, and show that our solution yields a 24× speedup on large datasets.

In the future, we are interested to study how to enable approximate top-k result diversification with guarantees leading to even faster running times. We also intend to explore how to adapt our proposed framework if diversity is assumed to satisfy metric property, in particular, the triangle inequality.

CHAPTER 4

TOP-K DIVERSIFICATION CONSIDERING FAIRNESS

4.1 Introduction

The proliferation of e-commerce platforms such as Amazon.com, Netflix, and Spotify.com has given rise to the so-called "infinite-inventory", which offer an order of magnitude more records (products, movies, songs) than their brick-and-mortar counter-parts [8]. This result in a long-tail market, where a handful of records get heavily exposed to the end users and a long tail of "niche" records remain relatively unknown. As a concrete example, the top-1000 highest rated movies in IMDB [53] follow a long tail distribution in terms of number of views (refer to Y-axis in Figure 4.1), even though they all have highly similar (average rating between 8.34 and 7.9) "utility" (IMDB ratings). The problem gets further exacerbated by downstream applications in ranking and recommendation, such as Learning-to-Rank (LTR) [88,92] framework or Collaborative Filtering (CF) [57] that consume user-feedback (explicit or implicit) on the top-k results to re-evaluate the *utility* scores of the records. The process makes a small set of records getting heavily exposed to the end users and these records continue to "upgrade" their utility scores compared to the rest. This inequitable exposure of the records conforms to the *rich-gets-richer dynamics* [88].

We advocate that for such long tail data, it is better to return one of the *equivalent top-k sets to the users, as opposed to a fixed one* (although how much change in the top-k answers the users experience must also be tunable). Imagine a toy instance on the top-1000 IMDB movies, where all 1000 movies have highly similar ratings. Based on a user query, imagine there are 4 sets of top-3 movies that are equivalent in utility and any of these four could be returned to the user. These are $s1$: $\{r_1, r_2, r_3\}$, $s2$: $\{r_2, r_3, r_4\}$, $s3$: $\{r_2, r_3, r_5\}$, and $s4$: $\{r_3, r_4, r_5\}$. However, if the

probability of returning any of these sets to the users is uniform (i.e., 1/4), then, the popular movie r_3's selection probability is 1 (no matter which set is selected, r_3 will always be present), whereas, a niche movie r_1's selection probability is only 1/4 (r_1 is only present in s1). Clearly, this process leads to inequitable and unfair selection probability of the movies (users will get to experience popular record r_3 many more times than niche r_1) hurting their equitable exposure. *Our focus is to redesign the existing top-k algorithms to address this unequal exposure concern.* To the best of our knowledge, we are the first to study this aspect of fairness inside top-k algorithms.

There is no single general definition of fairness, and it varies among different scenarios. In the next subsections, we gathered most popular categories of fairness definitions that most problems will belong to. We will explain each category based on hiring example.

4.1.1 Demographic parity

Consider two groups, one majority and one minority group. Demographic Parity or Statistical Parity states that the acceptance rate of the candidates from both groups should be equal. Geyik et al. [42] describes demographic parity such that the predictor outcome Y' be independent of the protected attribute A, that is:

$P(Y'|A = 0) == P(Y'|A = 1)$.

Singh et al. [76] defines it based on exposure and says the average exposure of items in two groups must be equal. In Pitoura et al. definition [68] demographic parity ensures that the proportion of each part of a protected group (e.g., gender) should receive the positive outcome at identical rates.

4.1.2 Equalized odds

Equalized odds states that the protected and unprotected groups should have the same rates for true positives and false positives. Unlike demographic parity, in

equalized odds Y' depends on the protected attribute A but only via the target variable Y [49]. It can be formulated as:

$P(Y' = 1|A = 0, Y = y) == P(Y' = 1|A = 1, Y = y)$.

In the binary case and many applications such as hiring, we care more about the true positive rate (Y = 1) rather than true negative rate. Hence, we focus on the relaxed version of the previous formula:

$P(Y' = 1|A = 0, Y = 1) == P(Y' = 1|A = 1, Y = 1)$.

this is called Equality of Opportunity. In our example, it is equivalent to hiring equal proportion of applicants from the "qualified" selection of each group.

Demographic Parity and Equalized Odds fall into a larger category named "group fairness".

4.1.3 Unawareness

This definition simply says we should not consider the sensitive attribute as a feature in the train set. This notion is also called as disparate treatment. Consider c is binary classifier that decides hiring based on X un protected features (such as college GPA) and A is protected attribute (such as sex) then the formulation is as follow:

$c(X, A) = c(X)$

4.1.4 Individual fairness

Individual fairness states that any two similar individuals should receive the same outcome. In [31] this definition is expressed as interpreting the goal of "mapping similar people similarly". The formulation is then:

$D(M(X), M(X')) \leq d(X, X')$,

where X, X' are two input feature vectors, and D and d are two metric functions on the input and the output space, respectively.

4.1.5 Counterfactual fairness

It states that a decision for an individual should be the same in both the actual world and a counterfactual world, where the individual is assigned to a different demographic group. The formulation is:

$$P[Y'_{A\leftarrow 0} = y|X, A = a] = P[Y'_{A\leftarrow 1} = y|X, A = a]$$

4.1.6 Proportionate fairness

Baruah et al. [13] introduced a new notion of fairness in resource allocation for periodic scheduling problems, named proportional fairness or P-Fairness. They defined p-fairness by introducing a notion named 'lag' that measures the difference between the number of resource allocations that task x should have received and the number that it actually received. In their definition, a schedule S is P-fair if and only if for all tasks x and periods t:

$$-1 < lag(S, x, t) < 1.$$

Our work in inspired by the individual fairness notion and we try to be fair to the products which have almost similar score but are under-represented.

Problem Motivation and Models. To address such fairness concern over long tail data, we adapt a political theory, namely, the *Sortition Act* [27, 77] and redesign existing top-k algorithms to have them compute *a set S of multiple top-k sets that are equivalent in utility* as opposed to a fixed top-k set. Given S, an end user still draws one of the sets at random. Hence, the goal is to assign a probability distribution over S, i.e., $PDF(S)$, such that after many such draws from many end users, the records returned inside the top-k sets have as uniform selection probability as possible. To that end, we formalize the notion of **θ-Equiv-top-k-MMSP** that finds for a given query and a scoring function $\mathcal{F}$. Each set $s \in S$ contains k number of records whose score is at most θ%-smaller than the optimum top-k score, and the $PDF(S)$ is computed such that the selection probabilities of the records in it are as uniform as

possible. Enabling equal selection probabilities of the records ensure that each record is equally likely to be returned to the end users and promotes fairness. This proposed notion rooted on maxmin fairness theory that maximizes the minimum fairness. We are aware of a few related works that we borrows inspiration from. [11] studies how to enable fairness in similarity search by returning points within distance r from the given query with the same probability. Both [36, 41] study how group fairness alone can hurt equitable exposure of the records and thus define computational frameworks that enable equal selection probability of the records in conjunction with group fairness constraints. These existing works do not have any easy extension to our problem - although we study how **θ-Equiv-top-k-MMSP** can complement group fairness.

Technical Contributions. We formalize key definitions, such as, θ-equivalent top-k sets, selection probability of records, and present **θ-Equiv-top-k-MMSP** that has two steps **(Section 4.2)**. (A) **θ-Equiv-top-k-Sets** generates S, the set of θ equivalent top-k sets (where θ is a tunable parameter that can control how much changes is desirable across different top-k sets for different applications), (B) **MaxMinFair** computes $PDF(S)$ such that the minimum selection probability of a record is maximized. We prove that the counting problem involved in **θ-Equiv-top-k-Sets** is #P-hard, which makes **θ-Equiv-top-k-MMSP** an NP-Complete problem.

We first present an exact algorithm **OptTop-k-θ** that produces S, all θ-equivalent top-k sets and is exact in nature. The algorithm is inspired by the celebrated NRA algorithm [34] that only allows sorted accesses on the input lists and returns exact answers as long as the scoring function is *monotonic*. At the heart of the process, **OptTop-k-θ** intends to maintain a set of candidate top-k sets, efficiently compute and maintain their best and worst possible scores through upper and lower bounds, and decide if it is safe to terminate and produce the exact S without having to read any more records. However, there are several non-trivial computational challenges that it has to deal with, mainly because the number of

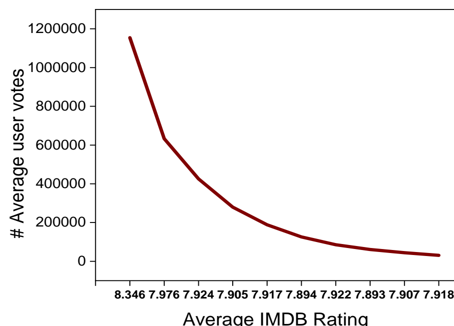


Figure 4.1 Viewership distribution of top-1000 IMDB movies.

possible size-k sets increases exponentially with new records being read. Therefore, **OptTop-k-θ** leverages an efficient data structure based on the concept of item lattice that allows efficient computation of the possible size-k sets and incremental updates of their score bounds by reusing previously calculated scores. For producing $PDF(S)$, we present a linear programming-based exact solution **Opt-SP**.

We present **RWalkTop-k-θ** is highly scalable to solve both **θ-Equiv-top-k-Sets** and **MaxMinFair**. We realize that the possible size-k set of sets over N records could be represented as a hierarchically ordered lattice containing $\binom{N}{k}$ nodes. Hence an efficiency opportunity lies in producing some of these nodes on the go, as opposed to discovering them from scratch one-by-one. We leverage this intuition in designing a *probabilistic algorithm* based on random walk that is backed by the Good Turing Test [40]. Good Turing Test is often used in population studies to estimate the number of unique species in a large unknown population [40], which we use to determine when **RWalkTop-k-θ** could stop and still discover all θ-equivalent top-k sets with high probability. Given S, **RWalkTop-k-θ** calls a highly efficient greedy solution **Gr-SP** to produce a probability distribution over it.

We finally design **ARWalkTop-k-θ**, an adaptive random walk based approach that solves **θ-Equiv-top-k-Sets** and **MaxMinFair** at the same time. The intuition comes from the fact (that we formally prove) that if S contains records that only appears in one and exactly one set $s \in S$, then $PDF(S)$ is a uniform probability

distribution which ensures equal selection probabilities for all records. **ARWalkTop-k-θ** is similar to the random walk described in **RWalkTop-k-θ**, except it performs the random walk adaptively, by lowering the probability of the records that are already part of some valid s, and boosting the probability of the remaining records that have not been part of any valid s yet. After that, $PDF(S)$ becomes a uniform probability distribution over the sets produced during the adaptive random walk.

Experimental Evaluations (Section 4.5). Our final contributions is empirical. We first demonstrate how the proposed problem becomes critical inside existing Learning-to-Rank(LTR) framework and compare that against existing group fairness notion. We also empirically demonstrate how our proposed notion of fairness complements group fairness. We use 4 different large scale real world datasets and two large synthetic datasets to extensively evaluate our designed solutions and compare them against several intuitive baseline algorithms. Our experimental evaluations also corroborate our theoretical analysis, it terms of the quality and the scalability of the designed solutions.

4.2 Data Model and Problem Definition

In this section, we present a running example, introduce key notations used throughout the chapter (Table 4.1), describe our data model, present key definitions, formalize **θ-Equiv-top-k-MMSP**, and study its hardness.

4.2.1 Running example

Consider the IMDB dataset D contains a large number of movies. The attributes are movie name, IMDB rating, year, genre, and director. Assume that a user searches for top-3 movies ($k = 3$) released in year 2022.

Given a threshold $\theta = 0.02$, the goal is to return a set S containing a set of sets, where each set s contains 3 movies, and the score of each set is at most 2% smaller than

the score of the set of 3-movies with the highest utility score. Let the scoring/utility function $\mathcal{F}$ be the weighted relevance and max sum diversity (WRMSD in short), as proposed below (with $\lambda = 0.5$). After that, compute a probability distribution over S, i.e., $PDF(S)$. Given $PDF(S)$, an end user draws one of these sets s randomly with probability $P(s)$. Thus, design $PDF(S)$, such that, after many such draws from many end users, the records returned inside the top-k sets have as uniform selection probabilities as possible.

Imagine only 5 movies as described in Table 4.2 are released in 2022. Therefore, the size-3 sets are only constructed from those 5 movies. Let IMDB rating reflect the relevance scores of the records. Let diversity be computed considering genre and director score. The sorted pairwise diversity score list is given in Table 4.3.

The maximum utility score is $= 19.85$ and the goal is to create a set S of sets, each s contains 3 movies such that, each s has utility score $\geq (19.85 - [0.02 \times 19.85]) = 19.45$. It is easy to notice that even with only 5 records, there are three sets that satisfy this condition (Table 4.4). After that, produce $PDF(S)$.

4.2.2 Data model

Database. A database D contains N records, where each record is represented as r.

Utility Based Scoring Functions. Given a query q and D, a utility based scoring function $\mathcal{F}$ scores each record with utility value $\mathcal{F}(r, q)$ and produces $\mathcal{F}(s, q), r \in s, |s| = k$, which is the the aggregated utility score of set s with k records.

- Relevance: $\mathcal{F}(r, q) = Rel(r, q)$, where Rel is the *relevance* between record r and query q.
- Diversity: Diversity is the dissimilarity between any two records, $Div(r_i, r_j)$ that is used to capture results that are representative of the population.

The attributes of the records could be used to calculate these values. Tables 4.2, 4.3 have some of those for Example 4.2.1.

Representative $\mathcal{F}$. Some representative utility functions appear as follows.

- Sum-relevance. $\mathcal{F}(s, q) = \Sigma_{r \in s} Rel(r, q)$
- Weighted relevance and max sum diversity (WRMSD). $\mathcal{F}(s, q) = \lambda \times \Sigma_{r \in s} Rel(r, q) + (1 - \lambda) \times \Sigma_{r \in s} Max_{r, r_j \in \{s-r\}} Div(r, r_j)$, where λ is a weight between $[0, 1]$.
- Maximal marginal relevance [23] or MMR. $\mathcal{F}(s, q) = \lambda \times \Sigma_{r \in s} Rel(r, q) + (1 - \lambda) \times \Sigma_{r \in s} Min_{r, r_j \in \{s-r\}} Div(r, r_j)$

Top-k Algorithms Given D, q, and an integer k, return a set s of k records from D that has the highest $\mathcal{F}(s, q)$, i.e.,

- $|s| = k$;
- s has the highest utility score, i.e., for any other set of k records s', $\mathcal{F}(s, q) \geq \mathcal{F}(s', q)$.

Promoting Fairness inside Top-k Algorithms It is easy to see that there could be more than one set of k-records that have highly similar utility score. To that end, we define the notion of equivalent size-k sets.

Definition 1. Equivalent size k sets. *Given a threshold θ, a query q and size k, two sets s_i and s_j each with k records are equivalent if the score of the set with lower score is not smaller than* a predefined threshold *θ% of that with the higher score, i.e.,*

$$s_i \equiv s_j \text{ if } \mathcal{F}(s_i, q) \geq (1 - \theta) \times \mathcal{F}(s_j, q), \text{ when } \mathcal{F}(s_i, q) < \mathcal{F}(s_j, q)$$

Running Example. Considering the example from Section 4.2.1 again, considering $\theta = 0.02$ and WRSMD as the scoring function, s1: $\{r_1, r_2, r_3\}$, s3: $\{r_2, r_3, r_5\}$ are two equivalent size k sets with scores 19.7 and 19.85, respectively.

Definition 2. Probability Distribution over size k sets. *Given a set S of sets, each with k records, a probability distribution $PDF(S)$ assigns a probability $P(s)$ to each $s \in S$, such that $\sum_{s \in S} P(s) = 1$.*

Definition 3. Selection probability of a record. *Given a probability distribution $PDF(S)$ of a set S containing many size k sets, the selection probability [36] of a record r is the sum of probability values of all the sets that contain r.*

$$\mathcal{P}(r) = \sum_{r \in s, s \in S} P(s) \tag{4.1}$$

Running Example. If $s1$:$\{r_1, r_2, r_3\}$, $s2$:$\{r_2, r_3, r_4\}$, and $s3$:$\{r_2, r_3, r_5\}$, and $P(s_1) = P(s_2) = P(s_3) = 1/3$, selection probability $\mathcal{P}(r_1) = P(s1) = 1/3$, whereas, $\mathcal{P}(r_3) = P(s1) + P(s2) + P(s3) = 1$. Indeed, no matter which set the end users draw, r_3 will be always returned, whereas, r_1 will be returned only 1/3 of the time.

4.2.3 Problem definition and hardness

Our overarching goal is to produce top-k set of sets that are "equivalent" in utility w.r.t. the set with the highest utility (i.e., the optimum top-k set), and ensure that all records present in any of the equivalent top-k sets have an equal selection probability. Generally speaking, we adapt the Egalitarian Social Welfare notion [32], which maximizes the lowest selection probability of a record present in any top-k sets.

Problem Definition 3. (θ-Equiv-top-k-MMSP) Maximize Minimum Selection Probability in θ-Equivalent Top-k Sets.

Given a database D with N records, scoring function $\mathcal{F}$, threshold θ, query q, and integer k, produce a set S of equivalent top-k sets and a probability distribution $PDF(S)$ over S, such that, the minimum selection probability of a record present in any $s \in S$ is maximized. Specifically, we define the following two sub-problems.

- **θ-Equiv-top-k-Sets.** *Produce a set S of all θ-equivalent top-k sets, such that, $s \in S$ satisfies:*

  $\mathcal{F}(s,q) \geq (1-\theta) \times argmax_{s' \in S}\mathcal{F}(s',q)$

- **MaxMinFair.** *Compute probability distributions S such that the smallest selection probability $\mathcal{P}(r)$ of a record $r \in s, s \in S$ is maximized. That is:*

$$Maximize\ Min\ \mathcal{P}(r), r \in s, s \in S, \tag{4.2}$$

In general, our proposed framework can accommodate any scoring function. However, when the scoring function is non-monotone, such as, MMR [23], the designed solutions become approximation.

Theorem 1. *The problem of finding the number of θ-**Equiv-top-k-Sets** is #P-hard.*

Proof. We show a polynomial time reduction from the problem of computing all maximal frequent itemsets of size at most t [45, 90] to the problem of computing all θ-equivalent top-k sets, that has a simple mapping between the number of solutions. This suffices since the problem of finding the number of σ-frequent maximal itemsets (threshold $\sigma \in [0,1]$) with at most t items of a given 0-1 database D is known to be #P-hard [90].

We take an instance of such 0-1 database with m transactions over N items. The σ is set to be $1/m$. Given one such instance of a 0-1 database, we create an instance of our problem as follows: each item becomes a unique record r, such that $\mathcal{F}(r,q) = 1$, for an arbitrary query q. $\mathcal{F}(s,q) = \Sigma_{\forall r \in s}\mathcal{F}(r,q)$. θ is set to be any number between $[0,1]$. A set of items is σ-frequent maximal itemset of size at most k, iff the set of records corresponding to the itemset forms a set s with score $\mathcal{F}(s,q) = k$. Therefore, the number of θ-equivalent top-k sets is at least as many as the number of σ frequent maximal itemsets of size at most k. This completes the reduction. □

Theorem 2. *The θ-**Equiv-top-k-MMSP** problem is NP-Complete.*

Proof. (sketch) We omit the details for brevity. Intuitively, the hardness comes from the fact that **θ-Equiv-top-k-MMSP** needs to enumerate all θ-equivalent top-k sets, which is at least as hard as counting all such sets that is proved to be #P-hard. □

Table 4.1 Table of Notations

Symbol	**Definition**
N	# records in D
k, q	size of result sets, query
θ,s, S	equivalence threshold, a top-k set, θ-equivalent top-k sets
$\mathcal{C}$, $\mathcal{L}$, $\mathcal{F}$	candidate set, sorted input lists, scoring function
$\mathcal{P}(r)$	selection probability of record r

Table 4.2 Records with Sorted Relevance (Example 4.2.1)

Record	**Movie Name**	**IMDB Score**
r1	Top Gun: Maverick	8.6
r2	K.G.F: Chapter 2	8.5
r3	Everything Everywhere All at Once	8.3
r4	RRR	8.1
r5	The Batman	7.9

Table 4.3 Sorted Diversity List Based on Example 4.2.1

Pair of records	**(r2,r3)**	**(r3,r5)**	**(r1,r3)**	**(r3,r4)**	**(r1,r4)**	**(r4,r5)**	**(r1,r2)**	**(r2,r4)**	**(r2,r5)**	**(r1,r5)**
Diversity Score	5	5	4	4	2	2	2	2	1	1

Table 4.4 WRMSD Scores of All Set of Sets, Each with Three Movies

sets	**{r1,r2, r3}**	**{r1,r2, r4}**	**{r1,r2,r5}**	**{r1,r3,r4}**	**{r1,r3,r5}**	**{r1,r4,r5}**	**{r2,r3,r4}**	**{r2,r3,r5}**	**{r2,r4,r5}**	**{r3,r4,r5}**
Utility Score	19.7	15.6	14.5	18.5	19.4	15.3	19.45	19.85	15.25	19.15

4.3 Exact Algorithms

We first describe an exact solution that solves both the sub-problems **θ-Equiv-top-k-Sets** and **MaxMinFair** exactly, thereby ensuring exact solution for **θ-Equiv-top-k-MMSP**.

The framework is described in Algorithm 6. To solve **θ-Equiv-top-k-Sets**, it runs in a loop and finds the i-th best top-k set in the i-th iteration - that is, $\mathcal{F}(s,q) =$ **TopkSets**$(i) \geq \mathcal{F}(s',q) =$ **TopkSets**(j), where $i < j$. It maintains all records that are seen throughout. This process continues until the utility score of a top-k set falls $\theta\%$ below from the optimum top-k. After that, it calls the **MaxMinFair** S to produce $PDF(S)$.

In Section 4.4.2, we will show how these two steps could be combined to design a highly scalable solution.

4.3.1 Algorithm for θ-Equiv-top-k-Sets

Our proposed algorithm **OptTop-k-θ** borrows inspiration from the celebrated **NRA** (No Random Access) algorithm [34]. It runs in a loop by performing sorted accesses over the input lists through a cursor movement by calling **DivGetBatch()**, gradually produces **TopkSets**(i) sets whose scores monotonically decreases, and finally terminates when all θ equivalent top-k sets are found. Since NRA requires the scoring functions to be monotonic, we demonstrate **OptTop-k-θ** using one of the representative function WRMSD described in Section 4.2.2. It performs three key operations.

1. Generates and maintains a candidate set $(\mathcal{C}, i, j)$ of top-k sets as it reads j-th records from the cursors. $(\mathcal{C}, i, j)$ is needed for deciding **TopkSets**(i).
2. Local stopping: if the **TopkSets**(i) is present in $(\mathcal{C}, i, j)$.
3. Global stopping: if all θ Equivalent top-k Sets are found.

Algorithm 6 Generic Framework for **θ-Equiv-top-k-MMSP**

Inputs: q, k, θ, database D, $\mathcal{F}$

Outputs: $PDF(S)$: probability distribution over a set S of top-k sets

1: $S \leftarrow \{\}$

2: $flag = 0$

3: $Opt = \infty$

4: $s =$ **TopkSets**$(1)(\mathcal{F}, D, k)$

5: $Opt = s.score$, $Score = Opt$

6: $S \leftarrow S \bigcup s$

7: i $\leftarrow$ 2

8: **while** $(Score \geq$ (1-θ) $\times$ $Opt)and(flag \neq 1)$ **do**

9: $\quad s =$ **TopkSets**$(i)(\mathcal{F}, D, k)$

10: $\quad S \leftarrow S \bigcup s$

11: $\quad Score = s.score$, $i \leftarrow i + 1$

12: **end while**

13: $PDF(S) \leftarrow$ **MaxMinFair**(S)

Generate i-th best top-k set The first two operations are done inside Algorithm **TopkSets**(i), whose pseudo-code is presented in Algorithm 7. **TopkSets**(i) is responsible for generating the i-th best top-k set. Without loss of generality, we assume there exists only one unique top-k set in each round. The argument extends when that is not true. Given the set $\mathcal{L}$ of sorted input lists, the algorithm sets a cursor on each list, and fetches the next record from those lists through $\mathcal{L}$ **DivGetBatch()** calls. As an example, if the input lists consist of both relevance and diversity, then **DivGetBatch()** fetches the next record from *sortedRelList* list as well as that from the *sortedDivList* list and their corresponding scores. The cursor points to the current position in the lists (let us assume that position to be j). It keeps track of the

all seen records upto j-th position. Then **createNewSets** creates all possible size-k sets (lines 1-4).

Algorithm 7 TopkSets (i)

Inputs: a set $\mathcal{L}$ of input lists, i, $\mathcal{F}$, k, **TopkSets**$(i-1).score$, θ, Opt

Outputs: $nextBest$: i-th best set

1: $cursor \leftarrow 0$, $seenR \leftarrow \emptyset$
2: **for** $j = cursor$ **to** $Max_{l \in \mathcal{L}} Len(l)$ **do**
3: $\quad seenR = \{seenR \bigcup \mathbf{DivGetBatch()}(l_1(j)), \mathbf{DivGetBatch()}(l_{|\mathcal{L}|}(j))\}$
4: $\quad (\mathcal{C}, i, j) \leftarrow$ **createNewSets**(seenR[j])
5: $\quad$ **for** s **in** $(\mathcal{C}, i, j)$ **do**
6: $\quad\quad$ lb(s), ub(s) $\leftarrow$ **LowerBound**(s), **UpperBound**(s)
7: $\quad$ **end for**
8: $\quad$ threshold[j] $\leftarrow$ **max**(ub)
9: $\quad$ **if** $threshold[j] < Opt \times (1-\theta)$ **then**
10: $\quad\quad nextBest = argmax(\mathcal{C}, i, j)$, $flag = 1$
11: $\quad\quad$ **return** $nextBest$
12: $\quad$ **end if**

13: **for** s **in** $(\mathcal{C}, i, j)$ **do**

14: **if** lb[s] $\geq$ **max**(ub($(\mathcal{C}, i, j) - s$)) **then**

15: $nextBest \leftarrow s$

16: **return** $nextBest$

17: **end if**

18: **if** ub[$(\mathcal{C}, i, j)$] ¡ **max**(lb($(\mathcal{C}, i, j) - s$)) **then**

19: **Prune** $\{(\mathcal{C}, i, j) - s\}$

20: **end if**

21: **end for**

22: **if** **max**(lb[$(\mathcal{C}, i, j)$ $\geq$ **min**(threshold[j], **TopkSets**$(i-1).score$ **then**

23: $nextBest \leftarrow$ **argmax**(lb($\mathcal{C}, i, j$))

24: **Break**

25: **end if**

26: $cursor \leftarrow j + 1$

27: **end for**

28: **return** $nextBest$

In order to accomplish (2), the other challenge involves score computations of size-k sets that are encountered so far. Since, **OptTop-k-**θ performs only sorted accesses, it may not be able to produce the exact score of a set of k records immediately - rather has to consider upper and lower bounds of score to argue if this set is a possible candidate for **TopkSets**(i). Upper bound score of a set s, $ub(s)$ (similarly lower bound score $lb(s)$) is the the maximum possible (similarly the smallest) possible score s can get. Moreover, when more records are being read, these bounds are to be updated as well. Section 4.3.1 describes how that could be done efficiently.

Lower and upper bound score of a set. Clearly, the lower bound (upper bound) score of a set s, $lb(s)$ (similarly $ub(s)$) is the minimum (similarly maximum) possible

score of s that **LowerBound** and **UpperBound** calculate. **LowerBound**(s) is calculated based on an objective function $\mathcal{F}$ and using the scores of any unseen component of $\mathcal{F}(s)$ by the smallest possible value. **UpperBound**(s) is done analogously, except the unseen component is replaced by the cursor reading at the j-th position. Lines 6-7 do that task.

Illustration using WRMSD. Imagine $\mathcal{F}$ is (weighted rel, max div). In that case $\mathcal{L}$ consists of two lists - a sorted relevance list *sortedRelList* and a sorted pairwise diversity lists *sortedDivList* in decreasing order of relevance and diversity values, respectively. Imagine the cursor is at the 2nd position of both these lists (i.e., $j = 2$)- therefore, so far it has seen $rel(r_1)$, $rel(r_2)$, $div(r_2, r_3)$, $div(r_3, r_5)$. Clearly, 4 records are seen so far, but all of their scores are not known - 3 different size-k ($k = 3$) sets could be produced. But, because of sorted access, the score of none of these sets could be calculated exactly. As an example, $ub(r_1, r_2, r_3) = 8.6 + 8.5 + 8.5 + 5 + 5 + 5$ if the weight λ is ignored. However, when the cursor reads another record, either from the relevance or from the diversity list, the ub of all sets need to be updated.

Deciding the i-th top-k set. Line 9 of **TopkSets**(i) produces and maintains a *threshold* and lines 10-14 decide if it needs to continue the computation any further or it is safe to terminate.

Definition 4. *Threshold is the maximum utility score of any unseen top-k set.* $threshold[j] = Max[ub(\mathcal{C}, i, j)]$

Given the cursor is at the j-th position of the input lists, if $threshold[j]$ falls below $Opt \times$ (1-θ), there is no point of looking any further,**TopkSets**(i) can terminate by returning the best set present in $(\mathcal{C}, i, j)$.

Lemma 7. $s =$ ***TopkSets***(i), *if* $s = argmax(lb(\mathcal{C}, i, j))$ *and* $lb(s) \geq max(ub(\mathcal{C}, i, j) - s))$

Lines 15-19 make another key calculation based on Lemma 7. It checks if there exists a set s in $(\mathcal{C}, i, j)$ with the maximum lower bound, such that the $lb(s)$ is not smaller than the upper bound scores of all other remaining sets in $(\mathcal{C}, i, j)$. In that case, s is the i-th best set and **TopkSets**(i) terminates upon returning that set and its values. Indeed, when $\mathcal{F}$ is monotonic, no other unseen sets can have higher score than s.

Lemma 8. *$s =$ **TopkSets**(i), if $s = argmax(lb(\mathcal{C}, i, j))$ and $lb(s) \geq$ **min**$(threshold[j]$, **TopkSets** $(i-1).score)$*

Similarly, based on Lemma 8, the algorithm makes another important decision in Lines 24-27. If the maximum $lb(s)$ of s is not smaller than the minimum of $threshold[j]$ and the score of the top-k set seen in the $i-1$-th iteration, then $lb(s)$ is the top-k set in the i-th iteration. This lemma holds good, since the scores of the returned top-k sets decrease monotonically over iterations.

Pruning sets. Even when **TopkSets**(i) can not terminate, it checks if all sets in $(\mathcal{C}, i, j)$ are potential candidates to be the i-th best set - clearly, if the upper bound score of a set s in $(\mathcal{C}, i, j)$ is not larger than the lower bound scores of all other sets in $\mathcal{C}$, s could be pruned.

Subroutine createNewSets Given $N' < N$ number of items that are encountered by **TopkSets**(i) already, when a new item r is read through a **DivGetBatch()** call, **OptTop-k-θ** has to to perform some hefty tasks.

- It needs to update $(\mathcal{C}, i, j)$ by adding additional size k sets that involve r.
- More importantly, it needs to update the lower and upper bound scores of the sets in $(\mathcal{C}, i, j)$ - or see if the score could be calculated exactly, if all required scores are read.

A naive idea is to regenerate all size $\binom{(N'+1)}{k}$ sets from scratch, which is computationally wasteful and exponential. To that end, we abstract the representation of the size k sets over a hierarchically ordered space as a lattice, and store *ub* and *lb* scores of the record sets there. This data structure offers a great benefit for doing both of these aforementioned tasks efficiently enabling incremental computation.

Data Structure. Given N' seen records, the lattice data structure maintains all $\binom{N'}{1}, \binom{N'}{2}, \ldots \binom{N'}{k}$ sets, as well as their utility score. A node in the lattice represents a possible set, singletons, pairs, triples, ..., size k sets, and so on. An edge represents the membership between two size l and $l+1$ sets. A complete lattice for our running example is shown in Figure 4.2 given $N = 5$, although the data structure only stores information upto size k sets. The set $\{r_1,\ r_2,\ r_3\}$ at level three is created by union of three sets in level two, which are $\{r_1,\ r_2\}$, $\{r_1,\ r_3\}$, $\{r_2,\ r_3\}$. Hence the edges represent the connection between these sets in level l and $l+1$.

Maintaining the structure. This data structure is updated incrementally as new records are read by **OptTop-k-θ**. Take the running example again and imagine $rel(r_1)$, and $div(r_2, r_3)$ is read. So far, the data structure have the following nodes r_1, r_2, r_3, $\{r_1, r_2\}$, $\{r_2, r_3\}$, $\{r_1, r_3\}$, and $\{r_1, r_2, r_3\}$. Next, imagine it reads $div(r_3, r_5)$, thus a new record r_5 is encountered. This creates a singleton, 4 new pairs, and 3 additional size-3 sets. Clearly, r_5 will include the following three additional size-k sets in $(\mathcal{C}, i)$, $\{r_1, r_2, r_5\}$, $\{r_2, r_3, r_5\}$, $\{r_1, r_3, r_5\}$.

Efficient bound computation and maintenance Imagine the cursor on the diversity list now moves to the third position and reads $div(r_1, r_3) = 4$. The upper bound scores of all of these following sets. $\{r_1, r_2, r_3\}$, $\{r_1, r_2, r_5\}$, $\{r_2, r_3, r_5\}$, $\{r_1, r_3, r_5\}$ are to be updated now. One can naively calculate these bounds from the scratch - but there exists an opportunity of reusing previously done computation that is clearly more efficient.

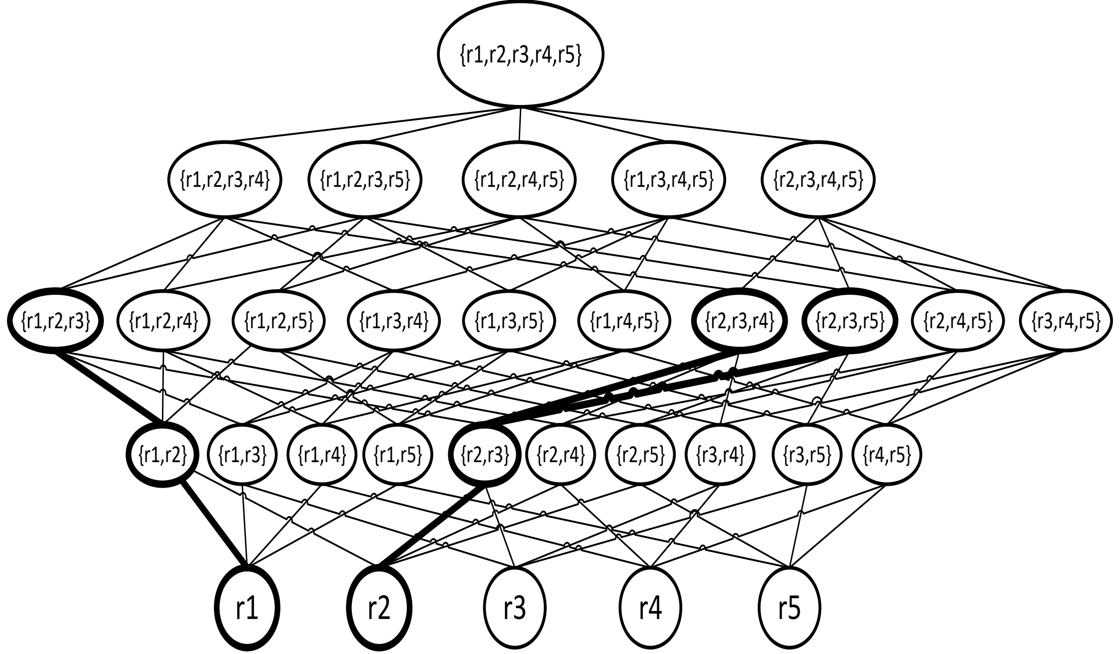


Figure 4.2 A complete lattice based on Example 4.2.1.

After reading $div(r_1, r_3) = 4$, our representation updates the score of the node $\{r_1, r_3\}$ in the lattice. All nodes that have a direct or indirect edge to $\{r_1, r_3\}$, their scores are also updated.

Similar situation occurs, when a new record r is encountered - the lattice representation allows us to quickly identify the new nodes that now contains r, as well as how to efficiently reuse the previously computed score of a set s' of size smaller than k to compute score of set $\{s' \bigcup r\}$.

$$\mathcal{F}(s' \bigcup r, q) = \mathcal{F}(s', q) + \mathcal{F}(r, q) \tag{4.3}$$

Formally, our effort is to study score update as an incremental process and reuse sub-computations that are done before. We express the score (lb, ub, or exact) of a set as a summation of scores over the subsets and retrieve the previously computed scores and reuse it, as opposed to calculating the scores from scratch every time.

Indeed, the lattice representation over the seen records allows us to decompose the score of a set as an aggregation over the sub-sets and reuse what has been done before.

Score reuse for WRMSD. Imagine an instance of **OptTop-k-θ** and the **DivGetBatch()** call has just returned the second row in the diversity list, namely $div(r_3, r_5) = 4$ and the goal is to produce top-k sets, where $k = 4$. A brand new record r_5 is just seen and this will add three additional size-3 sets $\{r_1, r_2, r_5\}$, $\{r_2, r_3, r_5\}$, $\{r_1, r_3, r_5\}$, four size-2 sets $\{r_1, r_5\}$, $\{r_2, r_5\}$, $\{r_3, r_5\}$, and one singleton r_5 on the lattice. The lattice structure facilitates score calculation of $WRMSD(\{r_1, r_2, r_3, r_5\})$ by reusing the scores that are calculated before. For the purpose of illustration, lets just consider the diversity component of the WRMSD calculation $WRMSD - Div(\{r_1, r_2, r_3, r_5\})$ and see how upper bound of scores could be calculated incrementally.

$$\begin{aligned} ub - div(\{r_1, r_2, r_3, r_5\}) = & \; Maxdiv[(r_1, \{r_2, r_3, r_5\})] \\ & + Maxdiv[(r_2, \{r_1, r_3, r_5\})] \\ & + Maxdiv[(r_3, \{r_1, r_2, r_5\})] \\ & + Maxdiv[(r_5, \{r_1, r_2, r_3\})]. \end{aligned}$$

Now consider $Maxdiv[(r_3, \{r_1, r_2, r_5\})]$ and note that this could simply be expressed as follows:

$$Maxdiv[(r_3, \{r_1, r_2, r_5\})] = Max(div(r_3, r_5), Maxdiv[(r_3, \{r_1, r_2\})] \quad (4.4)$$

$Maxdiv[(r_3, \{r_1, r_2\})]$ is pre-calculated. Hence, Equation (4.4) could be efficiently computed by taking a maximum over $Maxdiv[(r_3, \{r_1, r_2\})]$ score and $div(r_3, r_5)$. This allows sharing computation across sets.

Global stopping **OptTop-k-θ** halts when all θ-equivalent top-k sets are produced. This is checked by when one of the following two conditions is satisfied; (i). the last score received from **TopkSets**(i) is smaller than $(1 - \theta) \times Opt$, or (ii). the latest threshold fell below $(1 - \theta) \times Opt$ (which sets a flag to 1). It is guaranteed that there is no future unseen sets with score at most θ% smaller than the best top-k sets. At that point, **OptTop-k-θ** safely terminates and produces the exact solution.

Theorem 3. ***OptTop-k-θ*** *is an exact solution for* θ***-Equiv-top-k-Sets*** .

Proof. (sketch). Given a monotonic scoring function, it is easy to see that **TopkSets**(i) produces the i-th best top-k set in the i-th iteration. **OptTop-k-θ** maintains all records across iteration, forms all potential top-k sets. Finally, when **OptTop-k-θ** terminates, the global stopping condition guarantees that no unseen set of k records will be θ-equivalent of the top-k set. Hence the proof. □

Running time of OptTop-k-θ. In Section 4.2, we prove that the counting problem involved in θ**-Equiv-top-k-Sets** is #P-hard. In reality, the running time is dominated by the number of records **OptTop-k-θ** reads before termination and is dominated by the factor $\binom{\#\text{ seen records}}{k}$.

4.3.2 Algorithm for MaxMinFair

The last line of Algorithm 6 calls Algorithm **MaxMinFair**, which maximizes the minimum selection probability of the records present in S. We propose a linear programming based optimum solution **Opt-SP** that takes the set of sets S as input, and produces $PDF(S)$, such that **MaxMinFair** optimizes. The problem is formally defined as,

Maximize: x

subject to:

$$\mathcal{P}(r_i) \quad = \sum_{\forall r_i \in s, s \in S} P(s)$$

$$\mathcal{P}(r_i) \geq x, r_i \in s, s \in S$$

$$\sum_{\forall s \in S} P(s) = 1$$

Given the linear objective function and constraints this could be solved using an off-the-shelf linear programming solver using Simplex or Ellipsoid method.

Running Time. Opt-SP involves solving a linear program using Simplex or Ellipsoid method. Since the feasible region of thee objective function is a polytope, these algorithms take polynomial time to the input size N and $|S|$.

4.4 Approximation Algorithms

We realize that the possible size-k set of sets over N records could be represented as a hierarchically ordered lattice containing $\binom{N}{k}$ nodes. Hence an efficiency opportunity lies in producing some of these nodes on the go, as opposed to discovering them from scratch one-by-one. We present two approximate solutions to that end. The first one is **RWalkTop-k-θ**. To solve **θ-Equiv-top-k-Sets**, instead of designing a deterministic exact solution that could be exponential, it leverages a random walk based approach on the item lattice that is highly efficient and is backed by probability theory. To solve **MaxMinFair**, it presents an highly efficient greedy solution **Gr-SP**. **ARWalkTop-k-θ** solves both **θ-Equiv-top-k-Sets** and **MaxMinFair** at the same time through an adaptive random walk.

Algorithm 8 RWalkTop-k-θ

Inputs: query q, D, k, $\mathcal{F}$, θ

Outputs: $PDF(S)$

1: **while** true **do**
2: $s = \{\}, S = \{\}$
3: **while** $|s| \leq k$ **do**
4: pick a uniform random $r \in \{D - s\}$,
5: $s \leftarrow \{s \bigcup r\}$
6: **end while**
7: **if** $\mathcal{F}(s, q) \geq (1 - \theta) \times Opt$ **then**
8: $S \leftarrow \{S \bigcup s\}$
9: **end if**
10: $visit.s \leftarrow visit.s + 1$
11: **if** $visit.s \geq 2, \forall s \in S$ **then**
12: break
13: **end if**
14: **end while**
15: $PDF(S) \leftarrow$ **Gr-SP**(S)

4.4.1 Algorithm RWalkTop-k-θ

Algorithm 8 leverages probabilistic computation for producing **θ-Equiv-top-k-Sets** by making random walks on the item lattice. Following that, it solves **MaxMinFair** using a greedy technique.

Inputs to the algorithm are the query, k, objective function $\mathcal{F}$, θ, and the items in D. Additionally, it takes the optimum top-k set and its corresponding score from **TopkSets** 1. It starts by assigning each record a uniform probability of $1/N$. At each step it does uniform random sampling without replacement to select a record

and repeats the process until a set has k records. This completes a single random walk on the item lattice, where the walk consists of the edges that are traversed. After it retrieves a size k set s, it computes $\mathcal{F}(s, q)$ and retains s, if $\mathcal{F}(s, q) \geq Opt - \theta$. It keeps repeating the process and stops when each retained s is visited atleast twice in the process.

Termination Condition of the Random Walk The termination condition used for random walk is inspired by the Good Turing Test that is often used in population studies to determine the number of unique species in a large unknown population [40]. Consider a large population of individuals drawn from an unknown number of species with diverse frequencies, including a few common species, some with intermediate frequencies, and many rare species. Let us draw a random sample of N_{samp} individuals from this population, which results in $n1$ individuals that are the lone representatives of their species, and the remaining individuals belong to species that contain multiple representatives in the sample population. Then, $P0$, which represents the frequency of all unseen species in the original population can be estimated as follows:

Lemma 9. *Lemma 1 (Good Turing Test).* $P0 = n1/N_{samp}$.

The assumption here is that the overall probability of hitting one rare species is high while the probability of hitting the same rare species is low. Therefore, the more the sample hits the rare species multiple times, the less likely there are unseen species in the original population. We apply Lemma 9 to the θ-equivalent top-k sets construction, where a valid θ-equivalent top-k sets maps to the species and the probabilities of finding each such set in **RWalkTop-k-**θ are the frequencies. The set of θ-equivalent top-k sets discovered during **RWalkTop-k-**θ is the sample population. By ensuring this process visits each constructed set at least twice, we are essentially ensuring that n_1 is 0. Thus, using Lemma 9, P_0 can be estimated to be 0, which means it is highly likely that all θ-equivalent top-k sets are discovered.

Illustrative Example. Figure 4.2 shows the complete lattice involving Example 4.2.1. To solve **θ-Equiv-top-k-Sets**, the algorithm uniform randomly adds a record and continues the process until a size-3 is obtained. This way the set $s1$:{r_1, r_2, r_3} is formed. If $s1$ is a valid answer, it is retained. The process continues until all valid sets are discovered at least twice.

Subroutine Gr-SP Subroutine **Gr-SP** is designed by leveraging the following lemma.

Lemma 10. *If every record r in S appears in only one set $s \in S$, the $PDF(S)$ is a uniform distribution that guarantees equal selection probability of the records.*

Proof. Lemma 10 demonstrates an ideal scenario, where a record $r \in s$, $s \in S$ appears in only one s. If the $PDF(S)$ is a uniform distribution, that is, $P(s) = 1/|S|, \forall s \in S$, by leveraging the definition of selection probability of a record (Definition 3), then, $\mathcal{P}(r) = 1/|S|$. Clearly, this guarantees that each records r to have the same selection probability. □

Basically, the greedy algorithm is iterative and attempts to select a subset of sets from S that contains different records. Those subset of sets become part of O and gets a non-zero probability value. Specifically, It selects a set s from S in each iteration and adds to O, which includes the highest number of records that are not yet present in O but present in S. The process terminates when O contains all records in S. After that, each set that is present in O gets uniform probability of $\frac{1}{|O|}$. Any set $s \in \{S - O\}$, gets probability 0. We conjecture that this simple yet highly efficient algorithm accepts a 2-approximation factor, the formal proof is left to be explored in the future.

Illustrative Example. Imagine S contains the following 5 sets ($k = 2$), $s1$: $\{r_1, r_2\}$, $s2$: $\{r_3, r_4\}$, $s3$: $\{r_1, r_5\}$, $s4$: $\{r_3, r_5\}$, $s5$: $\{r_1, r_3\}$. If **Gr-SP** first adds $s1$ to O, then, in the next iteration it will add $s2$, and finally $s3/s4$. One possible solution will be

$O = \{s1, s2, s3\}$. Each of these sets will get a probability of 1/3 and the remaining two sets will have probability 0. The minimum selection probability of the records will be 1/3.

Running time. With an appropriate data structure, such as bucket queue, **Gr-SP** takes $\mathcal{O}(N \times |S|)$ to run.

4.4.2 Algorithm ARWalkTop-k-θ

The last algorithm **ARWalkTop-k-θ** we discuss does not separately compute **θ-Equiv-top-k-Sets**, and then, **MaxMinFair** - instead, solves these two problems together. It makes use of Lemma 10 to design an adaptive random walk.

The adaptive random walk based algorithm **ARWalkTop-k-θ** is similar to the random walk part of **RWalkTop-k-θ**, except it performs the random walk adaptively, by lowering the probability of the records that are already part of some valid s, and boosting the probability of the remaining records that have not been part of any valid s yet. The goal is to discover θ-equivalent top-k sets where the same record r repeats as few times as possible across the sets - ideally appears in one and only one s. The stopping condition is still guided by the Good Turing Test as described above. Once the process terminates, each set s in S gets uniform probability, and accordingly the selection probability of the records are calculated.

For each record $r \in N$, the algorithm keeps track of the sets in S that contain r ($r.seenCnt$). Instead of picking a record uniformly at random, it then, selects r with a probability that is inversely proportional to $r.seenCnt$. The intuition is that if a record r has already appeared in many $s \in S$, picking it again will hurt the minimum selection probability of other records r' that did not appear as frequently. Therefore, in the i-th iteration of the random walk, it is likely to discover a set of k records that contains new records that are not present in S yet.

Table 4.5 Dataset Statistics

Dataset	Size	Used Attributes
Yelp	112,686	latitude, longitude, review count
IMDB-top 1000	1,000	numVotes, genre,rating
IMDB	10,000	numVotes
Airbnb	39,882	price
Synthetic	10,000	random samples from uniform distribution
Makeblobs	1,000,000	random samples from guassian distribution

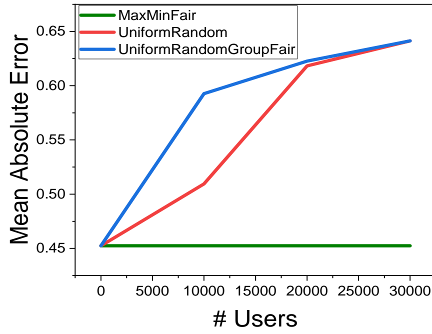


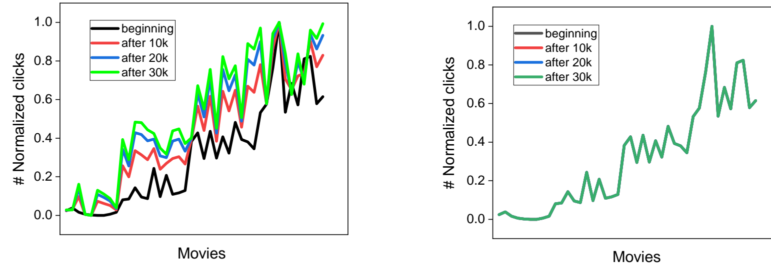
Figure 4.3 θ-Equiv-top-k-MMSP inside LambdaRank
Read source: [20].

Illustrative Example. Imagine Example 4.2.1 again and assume that $s1$: $\{r_1, r_2, r_3\}$ is discovered. After that, the $r_1.seenCnt$, $r_2.seenCnt$, $r_3.seenCnt$ are increased to 1, and the probabilities of these records are readjusted proportional to their $1/r.seenCnt$. Consequently r_1, r_2, r_3 now have smaller probabilities, whereas, r_4, r_5 have higher probability. Then the random walk is repeated again and the process terminates based on the Good Turing Test. Once S is obtained, each $s \in S$ is assigned uniform probability to produce $PDF(S)$.

4.5 Experimental Evaluations

Our experimental evaluations have four primary goals.

Goal (1). How **θ-Equiv-top-k-MMSP** promotes fairness in compelling downstream applications, such as, Learning-to-Rank(LTR) [20].



(a) Only group-fairness

(b) **θ-Equiv-top-k-MMSP** + group-fairness

Figure 4.4 θ-Equiv-top-k-MMSP complements group-fairness.

Goal (2). How **θ-Equiv-top-k-MMSP** complements existing group fairness criteria.

Goal (3). Examine the quality of our designed solutions and compare them with baselines.

Goal (4). Investigate scalability of the proposed algorithms and compare them with baselines.

1. Experimental setup. All algorithms are implemented in Python 3.8. All experiments are conducted on a server machine with 128GB RAM memory, OS: windows server 2019 datacenter, version: 1809, CPU: Processor 11th Gen Intel(R) Core(TM) i9-11900K @ 3.50GHz, 3504 Mhz, 8 Core(s), 16 Logical Processor(s). Obtained results are the average of three separate runs. Github has the code and data [9].

2. Datasets. Experiments are conducted on six datasets, four real and two synthetic data. For real datasets, we use **Yelp** [94], **IMDB-top 1000** [53], **IMDB** [51], and **Airbnb** [6]. For one of the synthetic data, we generate random samples for relevance and diversity scores from uniform distributions. The other synthetic data is **MakeBlobs** [63] from the sklearn package that produces data points from a normal distribution. Table 4.5 has an overview.

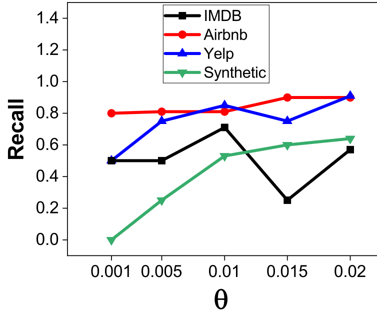


Figure 4.5 Recall of RWalkTop-k-θ varying θ.

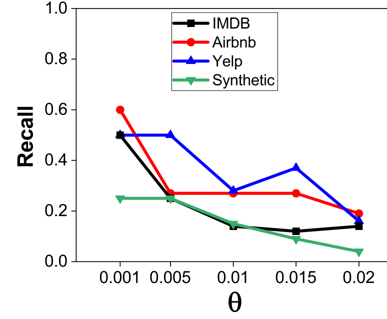


Figure 4.6 Recall of ARWalkTop-k-θ varying θ.

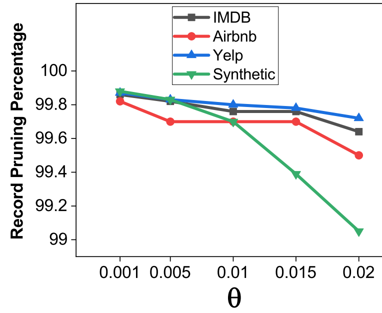


Figure 4.7 Record pruning percentage OptTop-k-θ.

3. Implemented Algorithms.

To evaluate fairness consideration using Learning-to-Rank(LTR) applications, we implement three different solutions inside LambdaRank [20]. **(1)** How **θ-Equiv-top-k-MMSP** promotes fairness inside LambdaRank. **(2)** Group fairness. How LambdaRank [20] behaves when top-k set satisfies only demographic parity constraints [75]. **(3)** No fairness. How LambdaRank [20] behaves with classical top-k solutions without any fairness consideration.

Additionally, we implement and compare the following Algorithms. We note that existing works [11,36,41] do not have an easy extension to solve **θ-Equiv-top-k-MMSP** because the solution frameworks do not adapt to solve **θ-Equiv-top-k-Sets**.

- **θ-Equiv-top-k-Sets.** We compare the exact algorithm **OptTop-k-θ** with the two approximate solutions **RWalkTop-k-θ** and **ARWalkTop-k-θ**.

- **MaxMinFair.** We implement a simple baseline **H-SP** first. It goes over the sets in S one by one and checks if all records in a set s are present in other sets in $\{S - s\}$. If yes, s is deleted from S. After that, the remaining sets are returned, each associated with uniform probability. We compare the LP-based exact solutions **Opt-SP**, with approximate solutions **Gr-SP** and **H-SP**.

4. Representative utility functions.

1. Maximize relevance. $\Sigma_{\forall r \in s} Rel(r, q)$
2. Weighted relevance and max sum diversity (WRMSD)
   Maximize $\lambda \times \Sigma_{r \in s} Rel(r, q) + (1 - \lambda) \times \Sigma_{r \in s} Max_{r, r_j \in \{s-r\}} Div(r, r_j)$, where λ is a weight between $[0, 1]$.
3. Maximize diversity. Maximize $\Sigma_{r \in s} Max_{r, r_j \in \{s-r\}} Div(r, r_j)$

5. Query & Parameters. Queries are selected randomly. Unless specified, the default parameters are $N = 10k$, $k = 5$, $\mathcal{F}$ = WRMSD with $\lambda = 0.99$, $\theta = 0.01$.

6. Evaluation Measures.

- **Goal (1)**, we present the mean absolute error (MAE) of a popular LTR model LambdaRank [20] and argue why equal exposure is necessary, compared to existing group fairness notion [75].
- **Goal (2)**, we present #clicks (min-max normalized) made by the users.
- **Goal (3)**, for **θ-Equiv-top-k-Sets** , we present recall [48] of the efficient alternatives **RWalkTop-k-θ** and **ARWalkTop-k-θ** compared to **OptTop-k-θ**. For **MaxMinFair**, we present approximation factors (objective function of approximate solution/ objective function of exact solution) of **Gr-SP** and **H-SP** wrt **Opt-SP**.
- **Goal (4),** for **θ-Equiv-top-k-Sets**, we present pruning capabilities of **OptTop-k-θ** , as well as study the scalability of the different algorithms designed for **θ-Equiv-top-k-Sets** and **MaxMinFair** varying pertinent parameters.

4.5.1 Goal 1: Fairness inside LambdaRank

In this experiment, the mean absolute error (MAE) of a LambdaRank [20] model is measured, where top-k results are returned satisfying three different considerations (**θ-Equiv-top-k-MMSP**, group-fairness, and no-fairness) while varying user exposure using the top-1000 IMDB movies that satisfy long tail distribution (recall Figure 4.1). The dataset is first split into train and test set. LambdaRank is built on the train set with the following input features: (i) number of views, (ii) action, (iii) horror, to predict its utility (IMDB rating). All three algorithms return all θ-equivalent top-k sets on the test data, where $\theta = 0.30$. **θ-Equiv-top-k-MMSP** assigns probability based on its proposed fairness criteria, whereas, the other two assigns equal probability to each set. Group fairness is imposed based on genre: drama, action, or neither. The total number of views for a movie is then updated as previous number of views + viewing probability $\times$ number of users. After modifying the test dataset based on updated views, the new features are passed on to trained LambdaRank model as input, and MAE is calculated. As Figure 4.3 demonstrates, **θ-Equiv-top-k-MMSP** ensures equal exposure leading to MAE having no effect for varying number of users (shown as a flat line). On the contrary, MAEs of group-fairness and no-fairness significantly increase (LTR model performs poorly) with the increase of number of users. This validates the necessity of **θ-Equiv-top-k-MMSP** for long tail data.

4.5.2 Goal 2: MMSP complements Group-fairness

Figures 4.4 show exposure of IMDB-1000 movies (as normalized clicks) considering group-fairness [75] alone and that of when **θ-Equiv-top-k-MMSP** is implemented inside group-fairness [75] considering different numbers of times it is returned to the end users. Group-fairness is imposed using the genre attribute. It is clear when the top-k records are returned based on **θ-Equiv-top-k-MMSP**, while satisfying

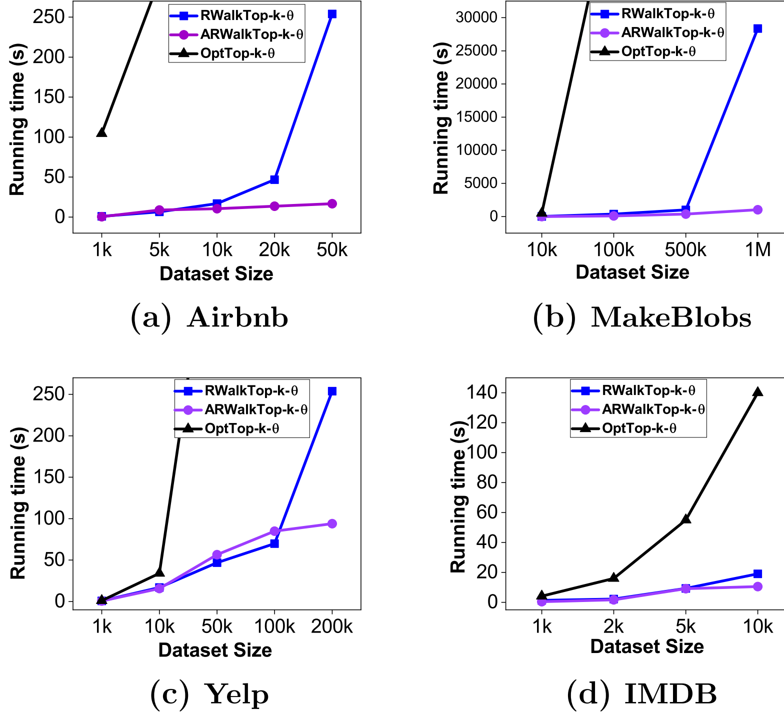


Figure 4.8 RWalkTop-k-θ vs ARWalkTop-k-θ vs OptTop-k-θ scalability by varying dataset size N.

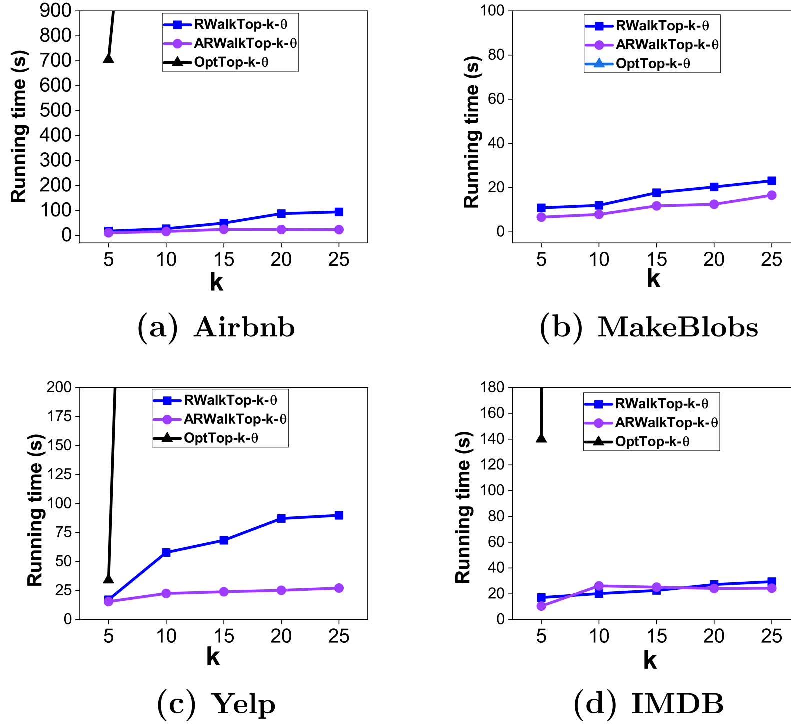
group-fairness [75], the exposure of the records remain unchanged (Figure 4.4(b)), whereas, records get inequitable exposure when only group-fairness is ensured. Thus, our proposed problem complements existing fairness definition.

4.5.3 Algorithms for θ-Equiv-top-k-Sets

In this section, we study the algorithms designed for **θ-Equiv-top-k-Sets** , qualitatively and scalability-wise.

Goal 3: Quality analysis: Recall We present recall results first.

A. Vary θ, RWalkTop-k-θ . We measure the quality of **RWalkTop-k-θ** by comparing the returned sets to the exact solution **OptTop-k-θ** . We present *Recall* percentage [48], which is the percentage of equivalent top-k sets returned by **RWalkTop-k-θ** algorithm w.r.t. **OptTop-k-θ**. Figure 4.5 shows the *Recall* value of **RWalkTop-k-θ** algorithm. It is encouraging to see that the recall of **RWalkTop-k-θ**



(a) **Airbnb** (b) **MakeBlobs**

(c) **Yelp** (d) **IMDB**

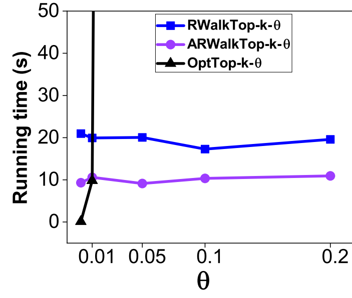
Figure 4.9 RWalkTop-k-θ vs ARWalkTop-k-θ vs OptTop-k-θ scalability by varying k.

is mostly above 80%, for almost all real datasets. At some point it becomes as high as 91%. When data distribution is uniform (synthetic data), clearly **RWalkTop-k-θ** becomes more effective with increasing θ, which is unsurprising.

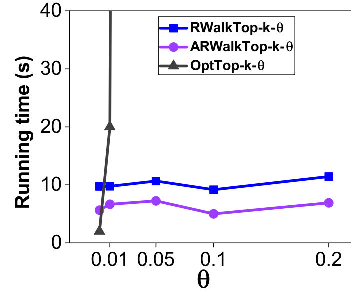
B. Vary θ, ARWalkTop-k-θ. Figure 4.6 shows the *Recall* value for **ARWalkTop-k-θ** algorithm. As expected, **ARWalkTop-k-θ** is inferior to solve **θ-Equiv-top-k-Sets** compared to **RWalkTop-k-θ** , as it only produces sets that are highly different from each other, giving rise to fewer number of sets. **ARWalkTop-k-θ** reaches up to 60% recall for Airbnb dataset.

Goal 4: Scalability analysis We present computational efficiency here.

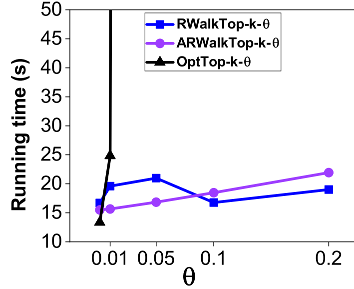
A. Pruning Effectiveness. We show that **OptTop-k-θ** solves **θ-Equiv-top-k-Sets** by accessing a very few records in the sorted lists. Figure 4.7 shows effective record pruning of **OptTop-k-θ** varying θ. Record pruning percentage is $= \frac{(N - \text{number of seen records})}{N}$. **OptTop-k-$\theta$** is able to prune 99% of the dataset



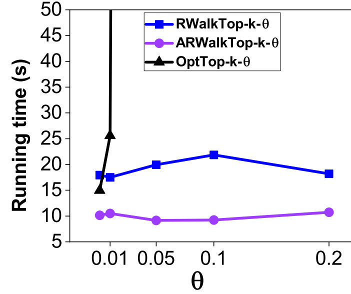
(a) **Airbnb**



(b) **MakeBlobs**



(c) **Yelp**



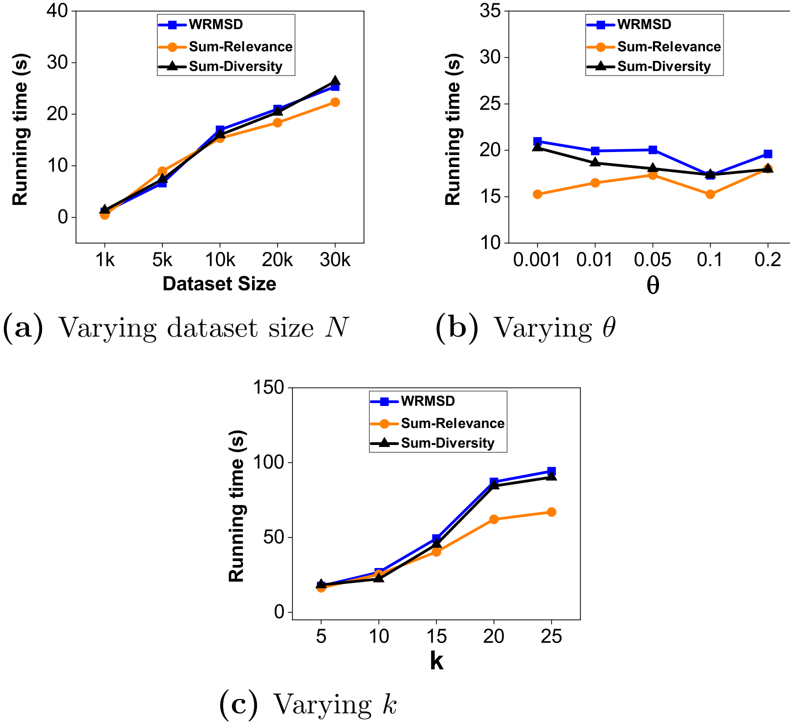
(d) **IMDB**

Figure 4.10 RWalkTop-k-θ vs ARWalkTop-k-θ vs OptTop-k-θ scalability by varying θ.

to exactly solve **θ-Equiv-top-k-Sets**. Also with θ, more equivalent sets are to be found, **OptTop-k-θ** needs to read more records, thereby pruning percentage slightly decreased by increasing θ.

B. Running time varying N. Figure 4.8 shows the scalability of the three proposed algorithms for **θ-Equiv-top-k-Sets** by increasing N. As expected, due to the exponential nature of **θ-Equiv-top-k-Sets** , **OptTop-k-θ** is not scalable over large value of N. In contrast, the other two proposed algorithms are scalable. **ARWalkTop-k-θ** is more scalable than **RWalkTop-k-θ** since it finds less number of sets because of its adaptiveness, it stops earlier. With 1M records in MakeBlobs, **ARWalkTop-k-θ** takes only a few minutes to finish.

C. Running time varying k. Figure 4.9 demonstrates the scalability of the three proposed algorithms by varying k. As expected, **OptTop-k-θ** does not scale well.



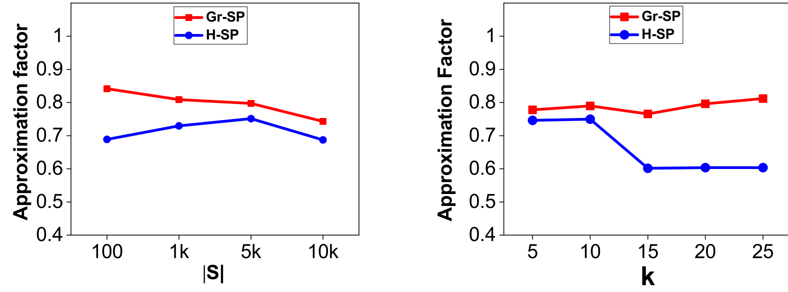
(a) Varying dataset size N **(b)** Varying θ

(c) Varying k

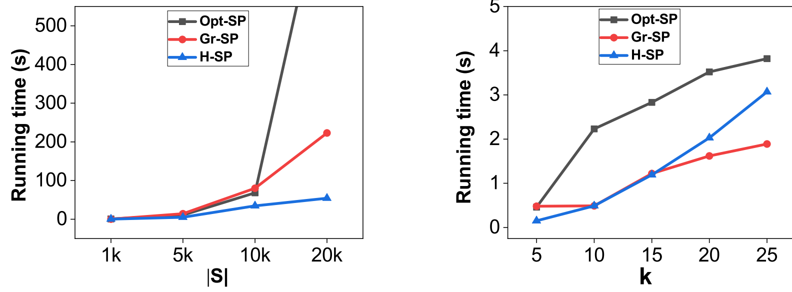
Figure 4.11 RWalkTop-k-θ scalability for different utility functions.

Consider Figure 4.9(c) using Yelp dataset. When $k = 5$, **OptTop-k-θ** takes 34.02 seconds to run, and the number of seen records is 28. $\binom{28}{5} = 98280$ sets are generated and examined only to produce 12 final top-k sets. Now consider that it is increased to $k = 10$. This may end up producing $\binom{28}{10} = 13123110$ sets even with only 28 seen records, which is 133$\times$ larger than before. This exponential increase is expected due to the computational nature of **θ-Equiv-top-k-Sets**. On the other hand, **RWalkTop-k-θ** and **ARWalkTop-k-θ** are highly scalable, and not very sensitive to increasing k.

D. Running time varying θ. Figure 4.10 demonstrates the scalability of the three proposed algorithms by varying θ. Increasing θ increases the size of $|S|$. As expected, **OptTop-k-θ** is highly sensitive to this parameter and does not scale well. In comparison, the random walk based algorithms **RWalkTop-k-θ** and



(a) Approx factor varying $|S|$ **(b)** Approx factor varying k



(c) Scalability by varying $|S|$ **(d)** Scalability by varying k

Figure 4.12 MaxMinFair approximation factor and scalability.

ARWalkTop-k-θ are less sensitive and scale reasonably well with increasing θ.

E. Running time varying $\mathcal{F}$. In Figure 4.11 we present the running time of **RWalkTop-k-θ** using the three representative utility functions, described at the beginning of the Section 4.5: Figure 4.11(a), Figure 4.11(b), Figure 4.11(c) demonstrate the scalability by varying parameters N, θ and k. As we can see, the running times of all three objective functions increase by increasing N, k, θ. However, the nature of the underlying objective function does not as such impact the running time. Similar observation holds for **ARWalkTop-k-θ** (the graphs are not presented for brevity). This is highly encouraging, as it demonstrates the effectiveness of our designed solutions across different objective functions.

4.5.4 Algorithms for MaxMinFair

In this section, we present the quality and scalability analysis of the three algorithms designed for **MaxMinFair**.

Goal 3: Quality analysis We present qualitative analysis first.

A. Approximation Factor. We calculate the approximation factor by dividing the minimum selection probability of the records returned by **Gr-SP** with that of **Opt-SP**. Since **MaxMinFair** is a maximization problem, hence the approximation factor is always ≤ 1. Similarly, the approximation factor of **H-SP** is also computed. As we shall demonstrate in Section 4.5.4, despite being an exact solution, **Opt-SP** is not highly scalable, since it involves a linear program. Figure 4.12 (a) shows the approximation factor using the sets returned by **RWalkTop-k-**θ algorithm for **Gr-SP** and **H-SP**. Since minimum selection probability for **Gr-SP** is higher than **H-SP**, its approximation factor is larger. The approximation factors demonstrate an encouraging facts. the minimum approximation factor value for **Gr-SP** is 0.74 and that of **H-SP** is 0.68, where as the maximum is 0.84 and 0.75, respectively. Figure 4.12 (b) present the approximation factor by varying k on 1000 sets returned by **RWalkTop-k-**θ algorithm for **Gr-SP** and **H-SP**. The minimum value of approximation factor of **Gr-SP** is 0.77, and for **H-SP** is 0.60, and the maximum values are 0.81 and 0.74, respectively.

Goal 4: Scalability analysis We evaluate the scalability varying $|S|$, N, k.

A. Running time varying $|S|$. Figure 4.12 (c) shows running time of the **Opt-SP**, **Gr-SP**, **H-SP** with $k = 5$. The heuristic **H-SP** exhibits the highest scalability among all and the linear programming based exact algorithm **Opt-SP** has the least scalability, as expected. Similar observation holds when N is varied. Nevertheless, both **Gr-SP** and **H-SP** are highly scalable and the results corroborate their theoretical running time.

B. Running time varying k. Figure 4.12 (d) shows the scalability with varying k and $|S| = 1000$. Similar observation holds as before that agorithms **Gr-SP** and **H-SP** are highly scalable to increasing k. This observation is also consistent to their theoretical analysis.

4.5.5 Summary of results

(a) Our most impactful observation is that **θ-Equiv-top-k-MMSP** is equipped to promote equitable exposure of records inside long tail data and benefits LTR models, compared to existing group fairness criteria. (b) **θ-Equiv-top-k-MMSP** complements group-fairness [75]. (c) Our third observation demonstrates the computational effectiveness of **OptTop-k-θ** - despite the fact **θ-Equiv-top-k-MMSP** is computationally intractable, our designed solution **OptTop-k-θ** is highly effective in pruning the vast majority of the records from the input database to produce the exact solution for **θ-Equiv-top-k-Sets**. The pruning effectiveness is at times as high as 99%. (d) We experimentally observe that **RWalkTop-k-θ** is a highly scalable algorithm that is several order of magnitude faster than the exact solutions **OptTop-k-θ** and **Opt-SP**, yet the produced results are highly comparable qualitatively. This solution achieves high recall, sometime, as high as 91% recall value. These results demonstrate the efficiency as well as effectiveness of **RWalkTop-k-θ** to be used and deployed inside real world applications. (e) Our final observation is that **ARWalkTop-k-θ** is a highly efficient solution that can easily scale to a very large N with millions of records, and is suitable for applications that can accommodate modest inaccuracy.

4.6 Conclusion

We formalize **θ-Equiv-top-k-MMSP** to redesign existing top-k algorithms for long tail data to ensure fairness. Given a query, **θ-Equiv-top-k-MMSP** computes a

set of top-k sets that are *equivalent* and assigns a probability distribution over these sets, such that, after many users draw a set from these sets according to its assigned probability, the selection probabilities of the records present in these sets are as uniform as possible. We present multiple algorithmic results with theoretical guarantees as well as present extensive experimental evaluation. We demonstrate how our proposed notion of fairness positively impacts compelling downstream applications, and complements group fairness.

This work opens up many interesting directions - one of the directions that we are currently exploring lies in understanding pre-processing techniques that can speed up the computation of θ**-Equiv-top-**k**-Sets**.

CHAPTER 5

SUMMARY AND FUTURE WORK

5.1 Summary

In this dissertation, we focus on two broadly defined challenges related to top-k results:

First, we study how to expedite existing diversification algorithms. We propose a computational framework that consists of two phases: offline and online phase. In the offline phase, we design an index structure over the groups of records instead of individual records, keeping similar records together in a node and dissimilar records separate. We keep minimum similarity and maximum similarity between nodes in a tree-based data structure named I-tree. Then in the online phase, we redesign three representative diversity algorithms to leverage this index to expedite their running times without losing the accuracy of the results. We design an API DivGetBatch(), which uses the index I-tree to prune the nodes which do not have the potential to be returned as the answer. We calculate the lower and upper bounds for each node and only select the node having the potential records as the result. After calling API and returning only a small number of records, we call original diversity algorithms to find the top-k. We also provide maintenance of our data structure over the dynamic data. Our goal is to insert a record to a node that minimizes the number of updates in the index, which is minimum similarity and maximum similarity between nodes. We provide an integer programming solution and a greedy solution for the insertion problem. Our index assumes the records to be atomic and the diversity function to be arbitrary, even metric or non-metric functions. We achieve up to 24× speedup and 90% pruning of the original dataset.

Second, we study the problem of equitable exposure of records. We study the long-tale data phenomenon where there exists a long sequence of records having equal

utility. However, given the existing algorithms, which are static, they only return a fixed top-k to the user query and making the process inequitable to the records which have similar utility. However, they never get the chance to be returned to the user. Then we show that there exist multiple equivalent top-k sets as the result that have similar utility, and we perform a probability distribution over these sets such that each record in these sets has an almost equal opportunity to be presented as a result. We present exact and approximate algorithms and provide proofs with theoretical guarantees as well as present extensive experimental evaluation. We present how the proposed notion of equitable exposure impact downstream applications, such as, Learning-to-Rank framework, and compare that against existing group fairness criteria. We run extensive experiments using multiple datasets and design intuitive baseline algorithms that corroborate our theoretical analysis.

5.2 Open and ongoing problems

This dissertation opens up several new problems, some of which are being investigated currently and some left for future work. Based on the current dissertation, we have identified three subproblems that we are currently working on.

Adaptation of other index data structures. One ongoing work is to study the adaptation of other index data structures in our DivGetBatch() framework. For instance, we want to study R-tree [46]. The R-tree data structure organizes spatial objects into a hierarchical structure of nested rectangles, where each rectangle corresponds to a node in the tree. Each node can contain multiple objects, and each object is contained in exactly one node. The rectangles of a node overlap to some degree, and they are chosen to minimize the overlap between adjacent nodes. R-trees support efficient queries for finding all objects that intersect with a given query region. The search algorithm starts at the root node and traverses the tree recursively, following the branches that intersect with the query region. The algorithm terminates

when it reaches a leaf node that contains objects that intersect with the query region. The similarity between our I-tree and R-tree is that both are height-balanced, and they group nearby objects in one node. This index is easily adaptable, and in order to use it, we need to calculate min and max distances between nodes and create $SimMatrixNode$. We are interested in doing experiments to see if using R-tree can help to prune or not. Another index structure that we intend to study which has non-trivial adaptation is Cover tree [18]. The tree is a series of levels arranged in a hierarchical order, where the highest level includes the root point and the lowest level includes all the points in the metric space. Each level, denoted by C, corresponds to a specific integer value i that decreases as one moves down the tree. The cover tree has three significant characteristics at every level C. (a) Nesting: $C_i \subset C_{i-1}$. This implies that once a point p appears in C_i, then every lower level in the tree has a node associated with p. (b) Covering tree: for every $p \in C_{i-1}$, there exists a $q \in C_i$ that $d(p, q) < 2^i$, and the node in level i associated with q is a parent of the node in level $i - 1$ associated with p. (c) Separation: for all distinct $p, q \in C_i$, $d(p, q) > 2^i$. Hence, we have fewer records at the higher levels and more records as we go down. The records that are part of the same node have a certain distance satisfied. The records that are part of the same node are actually farther from each other than the records that are apart from each other in two consecutive levels. We can find the node ID of the records and also the first node that this record appears. We can calculate the diversity between records using a cover tree based on the first level (first time) that these two records appear because we can ensure that the actual distance is more than the level gap, which is 2^i. The cover tree can be adapted in $SWAP$ algorithm. In $SWAP$, we go down the sorted relevance list until a threshold is satisfied. So by calling a getNext() interface, it gives the next best record that has the highest relevance with the query. Then it finds a candidate record from the current top-k set that has the smallest diversity contribution, and in each iteration,

it attempts to swap one record from the remaining records with the candidate record. DivGetBatch() calculates the upper bound and lower bound swap score between each node and the nodes of the current result set. So leveraging the cover tree in $SWAP$ process is that every time we read a new record, we can probe the cover tree and find out what would be the maximum and minimum diversity score that this new record can give rise to based on the distance criteria given in the cover tree. If this new record's lower bound of diversity contribution is not smaller than the maximum diversity contribution of the current set of nodes, then this record is swappable. Modifying MMR needs more change in DivGetBatch() API it is non-trivial since the cover tree does not give monotonic access to diversity, and we will continue that in future work.

Adaptation of other top-k criteria: Serendipity. In our second problem, we are interested in studying the adaptation of other top-k criteria, such as serendipity. The serendipity criterion measures how surprising or unexpected the selected records are. A top-k algorithm that prioritizes serendipity will aim to select items that are not only relevant but also unexpected, creating a sense of delight or surprise for the user. It is commonly agreed that serendipity consists of two components: surprise and relevance. The concept of serendipity in recommendation systems is still a relatively new and evolving area of research, and there is not yet a universally accepted definition or metric for measuring it. The idea of an item being surprising or unexpected is subjective and difficult to quantify [54]. We study the notion of serendipity from [101], where a record is considered to be serendipitous to a user if it is both relevant and unexpected. Based on an existing work [89], we define relevance and unexpectedness to define serendipity. The concept of relevance in recommendation systems is personalized and refers to a user's interest in items. To determine relevance, we compare a user's rating of an item to their average rating of all items. An item i is relevant if the rating given on i is greater than the average

value of all ratings provided. Given a returned list L of size N, the following metric defines the relevance of L as the ratio between the size of the subset of L that contains relevant items and the size of L:

$$Relevance@N = \frac{\sum_{i \in L} R(i)}{N}$$

where R(i) = 1 if i is relevant, and 0 otherwise.

The popularity of the item i is defined as the ratio between the number of users who rated i and the total number of users in the dataset. The item i is unexpected if its popularity score is below the average popularity computed across all the items in the dataset. This means that the average value of popularity allows splitting items in the dataset into two parts: the short head, containing the most popular (and expected) items, and the long tail, containing the less popular (and unexpected) items. The *Unexpectedness@N* metric defines the unexpectedness of set L as the ratio between the size of the subset of L that contains just unexpected items and the size of L:

$$Unexpectedness@N = \frac{\sum_{i \in L} U(i)}{N}$$

where U(i) = 1 if i is unexpected, and 0 otherwise.

Finally, *Serendipity@N* defines the serendipity of L as the ratio between the size of the subset of L that contains serendipitous items, i.e., those relevant and unexpected at the same time, and the size of L:

$$Serendipity@N = \frac{\sum_{i \in L} S(i)}{N}$$

where S(i) = 1 if i is serendipitous, and 0 otherwise.

So the objective function is:

$$Serendipity(r) = argmax_{r \in N} \lambda Relevance(r, q) + (1 - \lambda) unexpectedness(r)$$

where λ is a balancing coefficient between 0 and 1.

So our problem is to maximize serendipity in the top-k, which is to maximize the relevance and unexpectedness of the records in the top-k at the same time. Since calculating the relevance is dependent on the query, which is online, we need to go

through all records to select the maximum relevance, which is $\mathcal{O}$(N). Our challenge is how to use efficient indexing and preprocessing of some of the calculations offline to expedite the running time. One general solution is to precalculate the unexpectedness of all records since it is not query dependent, and the unexpectedness of a record does not depend on that of other records. We can precalculate and sort the unexpectedness in decreasing order and use it in the online phase when we have sorted lists of relevance and unexpectedness in decreasing order and access the records of each of these lists in sequential order like NRA algorithm [34]. We can set the threshold to be the aggregate(or minimum) of the scores seen in current access, and we stop when the scores of the top-k are greater or equal to the threshold. So using an NRA-based algorithm and this simple index, we can stop earlier and not go through all N records.

Adaptation of other notions of result diversification and other notions of fairness. As another ongoing work, the adaptation of other notions of result diversification and other notions of fairness that are relevant are being explored. One of the popular notions of diversification is attribute-aware diversification, which is not studied in this dissertation. In this definition, the diversity of a list is defined by how much each item in the list differs from the others in terms of their attribute values. For example, in [102], it can be defined on the attributes of each movie, including genre, actor, and director. Similarly, group fairness constraint in top-k results, such as demographic parity in top-k, is not studied in this dissertation. Group fairness is usually expressed in the form of constraints on the fraction of records from some protected groups that should be included in the top-k set for any relevant k. It ensures that the proportion of protected candidates in the top-k set is proportionate to the original data distribution [98]. For example, if the protected attribute is race and in original data 20% are African-American, 50% are Caucasian, and 30% are Asian, it means out of $k = 10$, the result will include 2 African-Americans, 5 Caucasian, and 3 Asian. These two definitions of attribute-aware diversification and group

fairness based on demographic parity can be treated similarly from a computational standpoint since they are the same but are defined on different types of attributes. Attribute-aware diversification ensures a certain percentage of variety in the top-k, for example, a certain value of genre, production company, and language, which are diversification attributes, while in demographic parity, the result needs to satisfy certain representation of protected attributes in the top-k. Hence, in our ongoing work, we are interested in making our DivGetBatch() API group fairness aware and studying how the result of existing diversity algorithms can satisfy the group fairness constraint, such as demographic parity. The problem is a constraint optimization problem which is to find the top-k set result satisfying the group fairness constraint as well as to maximize the diversification objective function. One naive solution and brute force is to create all possible k sets that satisfy this group fairness constraint and calculate its score based on the objective function, whether it is MMR or GMM and select the one with the highest score. This solution is exponential to the number of possible k sets that satisfy group fairness criteria. A possible efficiency opportunity comes through preprocessing by creating an index structure corresponding to some attribute values. Consider protected attribute gender having two values, male and female, then in the highest level, the records will be divided into two large partitions, male and female, and each of these partitions can be divided into more partitions. These nodes are created to ensure how many records are required from each group to be represented in the top-k set. Then inside each node, the DivGetBatch() framework is applied as is and keeps track of min and max distances between nodes inside each protected attribute value node. Hence, the algorithmic solution can stay the same, except that to ensure the number of records getting from each of these nodes is driven by the group fairness criteria. It is also challenging to understand how many nodes are to be created on the modified index structure, as it could generate an exponential

number of nodes if every unique attribute value is preserved as a node. We continue to explore it closely in our ongoing work.

REFERENCES

[1] Sofiane Abbar, Sihem Amer-Yahia, Piotr Indyk, and Sepideh Mahabadi. Real-time recommendation of diverse related articles. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1–12, 2013.

[2] Sofiane Abbar, Sihem Amer-Yahia, Piotr Indyk, Sepideh Mahabadi, and Kasturi R Varadarajan. Diverse near neighbor problem. In *Proceedings of the Twenty-ninth Annual Symposium on Computational Geometry*, pages 207–214, 2013.

[3] Zeinab Abbassi, Vahab S Mirrokni, and Mayur Thakur. Diversity maximization under matroid constraints. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 32–40, 2013.

[4] Pankaj K Agarwal, Stavros Sintos, and Alex Steiger. Efficient indexes for diverse top-k range queries. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 213–227, 2020.

[5] Rakesh Agrawal, Sreenivas Gollapudi, Alan Halverson, and Samuel Ieong. Diversifying search results. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, pages 5–14, 2009.

[6] Airbnb. Dataset, san francisco, ca, 2023. Available at: `http://insideairbnb.com/get-the-data`, retrieved on 4/7/2023.

[7] Albert Angel and Nick Koudas. Efficient diversity-aware search. In *ACM SIGMOD International Conference on Management of Data*, pages 781–792, 2011.

[8] Robert Armstrong. The long tail: Why the future of business is selling less of more. *Canadian Journal of Communication*, 33(1):127, 2008.

[9] Mahsa Asadi, 2023. Codes and data are available at: `https://anonymous.4open.science/r/FairSelectionInsideTopk-2F4F/README.md`, Retrieved on 4/7/2023.

[10] Abolfazl Asudeh, HV Jagadish, Julia Stoyanovich, and Gautam Das. Designing fair ranking schemes. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1259–1276, 2019.

[11] Martin Aumuller, Sariel Har-Peled, Sepideh Mahabadi, Rasmus Pagh, and Francesco Silvestri. Fair near neighbor search via sampling. *ACM SIGMOD Record*, 50(1):42–49, 2021.

[12] Krisztian Balog, Filip Radlinski, and Shushan Arakelyan. Transparent, scrutable and explainable user models for personalized recommendation. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 265–274, 2019.

[13] Sanjoy K Baruah, Neil K Cohen, C Greg Plaxton, and Donald A Varvel. Proportionate progress: A notion of fairness in resource allocation. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, pages 345–354, 1993.

[14] Rudolf Bayer. The universal b-tree for multidimensional indexing: General concepts. In *International Conference on Worldwide Computing and Its Applications*, pages 198–209. Springer, 1997.

[15] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 322–331, 1990.

[16] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 1975.

[17] S Berchtold, D Keim, and HP Kriegel. The X-tree: An efficient and robust access method for points and rectangles. In *Proceedings of 1996 International Conference Very Large Data Bases*, pages 28–39, 1996.

[18] Alina Beygelzimer, Sham Kakade, and John Langford. Cover trees for nearest neighbor. In *Proceedings of the 23rd International Conference on Machine Learning*, pages 97–104, 2006.

[19] Asia J Biega, Krishna P Gummadi, and Gerhard Weikum. Equity of attention: Amortizing individual fairness in rankings. In *The 41st international ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 405–414, 2018.

[20] Christopher JC Burges. From ranknet to lambdarank to lambdamart: An overview. *Learning*, 11(23-581):81, 2010.

[21] Zhi Cai, Georgios Kalamatianos, Georgios J Fakas, Nikos Mamoulis, and Dimitris Papadias. Diversified spatial keyword search on rdf data. *The Very Large Data Bases Journal*, pages 1–19, 2020.

[22] David Campos, Tung Kieu, Chenjuan Guo, Feiteng Huang, Kai Zheng, Bin Yang, and Christian S Jensen. Unsupervised time series outlier detection with diversity-driven convolutional ensembles–extended version. *arXiv preprint arXiv:2111.11108*, 2021.

[23] Jaime Carbonell and Jade Goldstein. The use of mmr, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st annual international ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 335–336, 1998.

[24] L Elisa Celis, Damian Straszak, and Nisheeth K Vishnoi. Ranking with fairness constraints. *arXiv preprint arXiv:1704.06840*, 2017.

[25] Paolo Ciaccia et al. M-tree: An efficient access method for similarity search in metric spaces. In *Very Large Data Bases*, volume 97, pages 426–435, 1997.

[26] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2009.

[27] Gil Delannoi and Oliver Dowlen. *Sortition: Thoery and Practice*, volume 3. Andrews UK Limited, 2016.

[28] Marina Drosou et al. Disc diversity: result diversification based on dissimilarity and coverage. *arXiv preprint arXiv:1208.3533*, 2012.

[29] Marina Drosou and Evaggelia Pitoura. Diversity over continuous data. *IEEE Data Eng. Bull.*, 32(4):49–56, 2009.

[30] Marina Drosou and Evaggelia Pitoura. Diverse set selection over dynamic data. *IEEE Transactions on Knowledge and Data Engineering*, 26(5):1102–1116, 2013.

[31] Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard Zemel. Fairness through awareness. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 214–226, 2012.

[32] Ulle Endriss. Lecture notes on fair division. *arXiv preprint arXiv:1806.04234*, 2018.

[33] Mohammadreza Esfandiari, Ria Mae Borromeo, Sepideh Nikookar, Paras Sakharkar, Sihem Amer-Yahia, and Senjuti Basu Roy. Multi-session diversity to improve user satisfaction in web applications. In *Proceedings of the Web Conference 2021*, pages 1928–1936, 2021.

[34] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656, 2003.

[35] Benjamin Fish, Ashkan Bashardoust, Danah Boyd, Sorelle Friedler, Carlos Scheidegger, and Suresh Venkatasubramanian. Gaps in information access in social networks? In *The World Wide Web Conference*, pages 480–490, 2019.

[36] Bailey Flanigan, Paul Gölz, Anupam Gupta, Brett Hennig, and Ariel D Procaccia. Fair algorithms for selecting citizens' assemblies. *Nature*, 596(7873):548–552, 2021.

[37] Will Fleisher. What's fair about individual fairness? In *Proceedings of the 2021 AAAI/ACM Conference on AI, Ethics, and Society*, pages 480–490, 2021.

[38] Piero Fraternali, Davide Martinenghi, and Marco Tagliasacchi. Top-k bounded diversification. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 421–432, 2012.

[39] Sorelle A Friedler, Carlos Scheidegger, and Suresh Venkatasubramanian. The (im) possibility of fairness: Different value systems require different mechanisms for fair decision making. *Communications of the ACM*, 64(4):136–143, 2021.

[40] William A Gale and Geoffrey Sampson. Good-turing frequency estimation without tears. *Journal of Quantitative Linguistics*, 2(3):217–237, 1995.

[41] David García-Soriano and Francesco Bonchi. Maxmin-fair ranking: individual fairness under group-fairness constraints. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 436–446, 2021.

[42] Sahin Cem Geyik, Stuart Ambler, and Krishnaram Kenthapadi. Fairness-aware ranking in search and recommendation systems with application to linkedin talent search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2221–2231, 2019.

[43] Sreenivas Gollapudi and Aneesh Sharma. An axiomatic approach for result diversification. In *Proceedings of the 18th International Conference on World Wide Web*, pages 381–390, 2009.

[44] Teofilo F Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

[45] Dimitrios Gunopulos, Roni Khardon, Heikki Mannila, Sanjeev Saluja, Hannu Toivonen, and Ram Sewak Sharma. Discovering all most specific sentences. *ACM Transactions on Database Systems (TODS)*, 28(2):140–174, 2003.

[46] Antonin Guttman. *R-trees: A dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[47] Jiawei Han, Micheline Kamber, and Jian Pei. Data mining concepts and techniques third edition. *The Morgan Kaufmann Series in Data Management Systems*, 5(4):83–124, 2011.

[48] Jiawei Han, Jian Pei, and Hanghang Tong. *Data mining: Concepts and Techniques*. Morgan Kaufmann, 2022.

[49] Moritz Hardt, Eric Price, and Nathan Srebro. Equality of opportunity in supervised learning. *arXiv preprint arXiv:1610.02413*, 2016.

[50] Tom Hope, Joel Chan, Aniket Kittur, and Dafna Shahaf. Accelerating innovation through analogy mining. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 235–243, 2017.

[51] IMDB. Dataset, 2023. Available at: `https://www.kaggle.com/datasets/isaactaylorofficial/imdb-10000-most-voted-feature-films-041118`, Retrieved: 4/7/2023.

[52] Zhongjun Jin, Mengjing Xu, Chenkai Sun, Abolfazl Asudeh, and HV Jagadish. Mithracoverage: a system for investigating population bias for intersectional fairness. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2721–2724, 2020.

[53] Kaggle. Top-1000 IMDB Movies. `https://www.kaggle.com/datasets/harshitshankhdhar/imdb-dataset-of-top-1000-movies-and-tv-shows,Retrievedon4/7/2023`.

[54] Marius Kaminskas and Derek Bridge. Diversity, serendipity, novelty, and coverage: a survey and empirical analysis of beyond-accuracy objectives in recommender systems. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 7(1):1–42, 2016.

[55] Norio Katayama and Shin'ichi Satoh. The sr-tree: An index structure for high-dimensional nearest neighbor queries. *ACM SIGMOD Record*, 26(2):369–380, 1997.

[56] Donald E. Knuth. *The Art of Computer Programming*, volume 1 of *Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., 3rd edition, 1998. (book).

[57] Yehuda Koren, Steffen Rendle, and Robert Bell. Advances in collaborative filtering. *Recommender Systems Handbook*, pages 91–142, 2022.

[58] Neeraj Kumar et al. What is a good nearest neighbors algorithm for finding similar patches in images? In *European Conference on Computer Vision*, pages 364–378. Springer, 2008.

[59] Chang Li, Haoyun Feng, and Maarten de Rijke. Cascading hybrid bandits: Online learning to rank for relevance and diversity. In *RecSys 2020: The ACM Conference on Recommender Systems*, pages 33–42. ACM, September 2020.

[60] Yunqi Li, Hanxiong Chen, Zuohui Fu, Yingqiang Ge, and Yongfeng Zhang. User-oriented fairness in recommendation. In *Proceedings of the Web Conference 2021*, pages 624–632, 2021.

[61] Rischan Mafrur, Mohamed A Sharaf, and Hina A Khan. Dive: diversifying view recommendation for visual data exploration. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 1123–1132, 2018.

[62] Sepideh Mahabadi and Ali Vakilian. Individual fairness for k-clustering. In *International Conference on Machine Learning*, pages 6586–6596. PMLR, 2020.

[63] Makeblobs. Dataset, 2023. Available at: `https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html`.

[64] Kyriakos Mouratidis. Geometric aspects and auxiliary features to top-k processing. In *2016 17th IEEE International Conference on Mobile Data Management (MDM)*, volume 2, pages 1–3. IEEE, 2016.

[65] Sepideh Nikookar, Mohammadreza Esfandiari, Ria Mae Borromeo, Paras Sakharkar, Sihem Amer-Yahia, and Senjuti Basu Roy. Diversifying recommendations on sequences of sets. *The Very Large Data Bases Journal*, pages 1–22, 2022.

[66] Francisco Parreño, Ramón Álvarez-Valdés, and Rafael Martí. Measuring diversity. a review and an empirical analysis. *European Journal of Operational Research*, 289(2):515–532, 2021.

[67] Gourab K Patro, Arpita Biswas, Niloy Ganguly, Krishna P Gummadi, and Abhijnan Chakraborty. Fairrec: Two-sided fairness for personalized recommendations in two-sided platforms. In *Proceedings of The Web Conference 2020*, pages 1194–1204, 2020.

[68] Evaggelia Pitoura, Georgia Koutrika, and Kostas Stefanidis. Fairness in rankings and recommenders. In *International Conference on Extending Database Technology (EDBT)*, pages 651–654, 2020.

[69] Evaggelia Pitoura, Kostas Stefanidis, and Georgia Koutrika. Fairness in rankings and recommendations: an overview. *The Very Large Data Bases Journal*, pages 1–28, 2021.

[70] Evaggelia Pitoura, Panayiotis Tsaparas, Giorgos Flouris, Irini Fundulaki, Panagiotis Papadakos, Serge Abiteboul, and Gerhard Weikum. On measuring bias in online information. *ACM SIGMOD Record*, 46(4):16–21, 2018.

[71] Shameem A Puthiya Parambath, Nicolas Usunier, and Yves Grandvalet. A coverage-based approach to recommendation diversity on similarity graph. In *Proceedings of the 10th ACM Conference on Recommender Systems*, pages 15–22, 2016.

[72] Lu Qin, Jeffrey Xu Yu, and Lijun Chang. Diversifying top-k results. *arXiv preprint arXiv:1208.0076*, 2012.

[73] Pengjie Ren, Zhumin Chen, Zhaochun Ren, Furu Wei, Jun Ma, and Maarten de Rijke. Leveraging contextual sentence relations for extractive summarization using a neural attention model. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 95–104, 2017.

[74] John T Robinson. The KDB-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, pages 10–18, 1981.

[75] Babak Salimi, Bill Howe, and Dan Suciu. Database repair meets algorithmic fairness. *ACM SIGMOD Record*, 49(1):34–41, 2020.

[76] Ashudeep Singh and Thorsten Joachims. Fairness of exposure in rankings. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2219–2228, 2018.

[77] Peter Stone. Sortition, voting, and democratic equality. *Critical review of international social and political philosophy*, 19(3):339–356, 2016.

[78] Chun-Hua Tsai and Peter Brusilovsky. Beyond the ranked list: User-driven exploration and diversification of social recommendation. In *23rd International Conference on Intelligent User Interfaces*, pages 239–250, 2018.

[79] Saúl Vargas, Linas Baltrunas, Alexandros Karatzoglou, and Pablo Castells. Coverage, redundancy and size-awareness in genre diversity for recommender systems. In *Proceedings of the 8th ACM Conference on Recommender Systems*, pages 209–216, 2014.

[80] Saúl Vargas and Pablo Castells. Rank and relevance in novelty and diversity metrics for recommender systems. In *Proceedings of the fifth ACM Conference on Recommender Systems*, pages 109–116, 2011.

[81] Sanne Vrijenhoek, Gabriel Bénédict, Mateo Gutierrez Granada, Daan Odijk, and Maarten de Rijke. Radio – rank-aware divergence metrics to measure normative diversity in news recommendation. In *RecSys 2022: The ACM Conference on Recommender Systems*. ACM, September 2022.

[82] Dongjing Wang, Shuiguang Deng, and Guandong Xu. Sequence-based context-aware music recommendation. *Information Retrieval Journal*, 21(2-3):230–252, 2018.

[83] Lina Wang, Xuyun Zhang, Tian Wang, Shaohua Wan, Gautam Srivastava, Shaoning Pang, and Lianyong Qi. Diversified and scalable service recommendation with accuracy guarantee. *IEEE Transactions on Computational Social Systems*, 2020.

[84] Dong Wei, Md Mouinul Islam, Baruch Schieber, and Senjuti Basu Roy. Rank aggregation with proportionate fairness. In *Proceedings of the 2022 International Conference on Management of Data*, pages 262–275, 2022.

[85] David A White and Ramesh Jain. Similarity indexing with the ss-tree. In *Proceedings of the Twelfth International Conference on Data Engineering*, pages 516–523. IEEE, 1996.

[86] Wen Wu, Li Chen, and Yu Zhao. Personalizing recommendation diversity based on user personality. *User Modeling and User-Adapted Interaction*, 28(3):237–276, 2018.

[87] Yingying Wu, Yiqun Liu, Fei Chen, Min Zhang, and Shaoping Ma. Beyond greedy search: pruned exhaustive search for diversified result ranking. In *Proceedings of the 2018 ACM SIGIR International Conference on Theory of Information Retrieval*, pages 99–106, 2018.

[88] Himank Yadav, Zhengxiao Du, and Thorsten Joachims. Fair learning-to-rank from implicit feedback. In *SIGIR*, 2020.

[89] Sadok Ben Yahia and Imen Ben Sassi. Poi based serendipitous recommender algorithm.

[90] Guizhen Yang. The complexity of mining maximal frequent itemsets and maximal frequent patterns. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 344–353, 2004.

[91] Ke Yang and Julia Stoyanovich. Measuring fairness in ranked outputs. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, pages 1–6, 2017.

[92] Tao Yang and Qingyao Ai. Maximizing marginal fairness for dynamic learning to rank. In *Proceedings of the Web Conference 2021*, pages 137–145, 2021.

[93] Jin-ge Yao, Xiaojun Wan, and Jianguo Xiao. Recent advances in document summarization. *Knowledge and Information Systems*, 53(2):297–336, 2017.

[94] Yelp. Dataset, 2023. Available at: `https://www.yelp.com/dataset/documentation/main`.

[95] Cong Yu, Laks Lakshmanan, and Sihem Amer-Yahia. It takes variety to make a world: diversification in recommender systems. In *Proceedings of the 12th international Conference on Extending Database Technology: Advances in Database Technology*, pages 368–378, 2009.

[96] Muhammad Bilal Zafar, Isabel Valera, Manuel Gomez Rodriguez, and Krishna P Gummadi. Fairness beyond disparate treatment and disparate impact: Learning classification without disparate mistreatment. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1171–1180, 2017.

[97] Michele Zanitti et al. A user-centric diversity by design recommender system for the movie application domain. In *Companion Proceedings of WWW*, pages 1381–1389, 2018.

[98] Meike Zehlike, Francesco Bonchi, Carlos Castillo, Sara Hajian, Mohamed Megahed, and Ricardo Baeza-Yates. Fa* ir: A fair top-k ranking algorithm. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 1569–1578, 2017.

[99] Rich Zemel, Yu Wu, Kevin Swersky, Toni Pitassi, and Cynthia Dwork. Learning fair representations. In *International Conference on Machine Learning*, pages 325–333. PMLR, 2013.

[100] Hantian Zhang, Xu Chu, Abolfazl Asudeh, and Shamkant B Navathe. Omnifair: A declarative system for model-agnostic group fairness in machine learning. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2076–2088, 2021.

[101] Mingwei Zhang, Yang Yang, Rizwan Abbas, Ke Deng, Jianxin Li, and Bin Zhang. Snpr: A serendipity-oriented next poi recommendation model. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*, pages 2568–2577, 2021.

[102] Cai-Nicolas Ziegler, Sean M McNee, Joseph A Konstan, and Georg Lausen. Improving recommendation lists through topic diversification. In *Proceedings of the 14th International Conference on World Wide Web*, pages 22–32, 2005.