FairSearch: A Tool For Fairness in Ranked Search Results

Meike Zehlike
Humboldt Universität zu Berlin
Max Planck Inst. for Software Systems
meikezehlike@mpi-sws.org

Tom Sühr
Technische Universität Berlin
tom.suehr@googlemail.com

Carlos Castillo
Universitat Pompeu Fabra
chato@acm.org

Ivan Kitanovski
Faculty of Computer Science and Engineering
University Saint Cyril and Methodius
ivan.kitanovski@finki.ukim.mk

ABSTRACT

Ranked search results and recommendations have become the main mechanism by which we find content, products, places, and people online. With hiring, selecting, purchasing, and dating being increasingly mediated by algorithms, rankings may determine business opportunities, education, access to benefits, and even social success. It is therefore of societal and ethical importance to ask whether search results can demote, marginalize, or exclude individuals of unprivileged groups or promote products with undesired features.

In this paper we present FairSearch, the first fair open source search API to provide fairness notions in ranked search results. We implement two well-known algorithms from the literature, namely FA*IR (Zehlike et al., 2017) and DELTR (Zehlike and Castillo, 2018) and provide them as stand-alone libraries in Python and Java. Additionally we implement interfaces to Elasticsearch for both algorithms, a well-known search engine API based on Apache Lucene. The interfaces use the aforementioned Java libraries and enable search engine developers who wish to ensure fair search results of different styles to easily integrate DELTR and FA*IR into their existing Elasticsearch environment.

CCS CONCEPTS

• **Information systems** → **Learning to rank**; • **Applied computing** → *Law, social and behavioral sciences*;

KEYWORDS

Ranking, Algorithmic Fairness, Disparate Impact

ACM Reference Format:
Meike Zehlike, Tom Sühr, Carlos Castillo, and Ivan Kitanovski. 2020. FairSearch: A Tool For Fairness in Ranked Search Results. In *Companion Proceedings of the Web Conference 2020 (WWW '20 Companion), April 20–24, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3366424.3383534

1 INTRODUCTION

With the volume of information increasing at a frenetic pace, ranked search results have become the main mechanism by which we find relevant content. Ranking algorithms automatically score and sort these contents for us, typically by decreasing probability of an item being relevant [6]. Therefore, more often than not, algorithms choose not only the products we are offered and the news we read, but also the people we meet, or whether we get a loan or an invitation to a job interview. With hiring, selecting, purchasing, and dating being increasingly mediated by algorithms, rankings may determine business opportunities, education, access to benefits, and even social success. It is therefore of societal and ethical importance to ask whether search algorithms produce results that can demote, marginalize, or exclude individuals of unprivileged groups (e.g., racial or gender discrimination) or promote products with undesired features (e.g., gendered books) [2, 4, 5, 8].

This paper operates on the concept of a historically and currently disadvantaged *protected group*, and the concern of *disparate impact*, i.e., a loss of opportunity for said group independently of whether they are treated differently. In rankings disparate impact translates into differences in exposure [7] or inequality of attention across groups, which are to be understood as systematic differences in access to economic or social opportunities.

In this paper we present FairSearch, the first fair open source search API that implements two well-known methods from the literature, namely FA*IR [9] and DELTR [10]. For both algorithms the implementation is provided as a stand-alone Java and Python library, as well as interfaces for Elasticsearch,[1] a popular, well-tested search engine, which is used by many big brands such as Amazon, Netflix and Facebook. Our goal with FairSearch is to provide various approaches for fair ranking algorithms, with a broad spectrum of justice definitions to satisfy many possible fairness policies in various business situations. By providing the algorithms as stand-alone libraries in Python and Java *and* for Elasticsearch we make the on-going research on fair machine learning accessible and ready-to-use for a broad community of professional developers and researchers, particularly those working in the realm of human-centric and socio-technical systems, as well as sharing economy platforms.

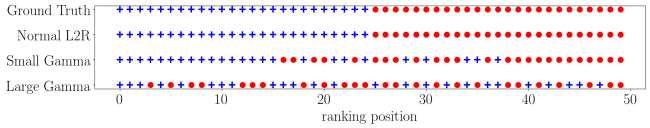
2 THEORETICAL BACKGROUND

This section explains the math behind FA*IR and DELTR and gives examples for their application domain. DELTR [10]constitutes a so-called *in-processing* approach, that incorporates a fairness term into its learning objective. This way it can learn to ignore the protected feature as well as non-protected ones that serve as proxies, such as

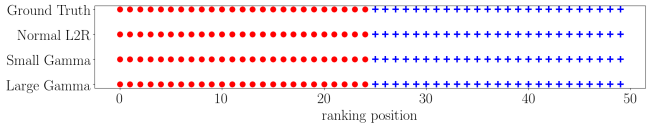
[1] https://www.elastic.co/

WWW '20 Companion, April 20–24, 2020, Taipei, Taiwan

ACM ISBN 978-1-4503-7024-0/20/04.
https://doi.org/10.1145/3366424.3383534



(a) Case where all non-protected elements appear first in the training set



(b) Case where all protected elements appear first in the training set

Figure 1: Depiction of test results using synthetic data. Top: DELTR reduces disparate exposure. Bottom: asymmetry in DELTR, which does not change rankings if protected elements already appear in the first positions.

ZIP code. FA*IR [9] belongs to the class of post-processing procedures and re-ranks a given search engine result to meet predefined fairness constraints.

2.1 DELTR: A Learning-To-Rank Approach

In traditional learning-to-rank (LTR) systems a ranking function f is learned by minimizing a loss function L, that measures the error between predictions $\hat{y}$ made by f and the training judgments y. For DELTR the loss function of ListNet [3], a well-known LTR algorithm is extended by a term U, which measures the "unfairness" of a predicted ranking. This way a new loss function $L_{\text{DELTR}} = L(y, \hat{y}) + \gamma U(\hat{y})$ *simultaneously* optimizes f for relevance *and* fairness. U is defined to be a measure of *disparate exposure* across different social groups in a probabilistic ranking $P_{\hat{y}}$. This means discrepancies in the probability to appear at the top position, received by items of the protected group G_1 vs items of the non-protected group G_0 are measured:

$$U(\hat{y}) = \max\left(0, \text{Exposure}(G_0|P_{\hat{y}}) - \text{Exposure}(G_1|P_{\hat{y}})\right)^2$$

Figure 1 shows how DELTR works on a synthetic dataset which has a total size of 50 items and each item x_i is represented by two features: their protection status and a score between 0 and 1: $x_i = (x_{i,1}, x_{i,2})$. The attribute $x_{i,1}$ is 1 if the item belongs to the protected group G_1, and 0 otherwise. The scores $x_{i,2}$ are distributed uniformly at random over two non-overlapping intervals. Training documents are ordered by decreasing scores, hence the top element is the one that has the highest score.

We first consider a scenario in which all protected elements have strictly smaller scores than all non-protected ones (Figure 1a). A standard learning to rank algorithm in this case places all non-protected elements above all protected elements, giving them a larger exposure. Instead, DELTR with increasing values of γ reduces the disparate exposure, while still considering the discrepancy in the score values. Figure 1b shows the asymmetry of the method: if the protected elements already receive larger predicted exposure than the non-protected by ranker f, DELTR will behave like a standard LTR approach.

2.2 FA*IR: A Re-Ranking Approach

Being a post-processing method, FA*IR [9] assumes that a ranking function has already been trained and a ranked search result is available. Its *ranked group fairness constraint* guarantees that in a given ranking of length k, the ratio of protected items does not fall far below a given p at *any ranking position*. FA*IR translates this constraint into a statistical significance test, using the binomial cumulative distribution function F with parameters p, k and α and declares a ranking as fairly representing the protected group if, for each k the following constraint holds:

$$F(\tau_p; k, p) > \alpha,$$

where τ_p is the actual number of protected items in the ranking under test. This constraint can now be used to calculate the minimum number of protected items at each ranking position such that the constraint holds (see table 1 with different examples of p). As an example consider the ranking in table 2 that corresponds to a job candidate search for an "economist" in the XING dataset used in [9]. We observe that the proportion of male and female candidates keeps changing throughout the top k positions, which in this case disadvantages women by preferring men at the top-10 positions. Suppose that the required proportion of female candidates is $p = 0.3$, this translates into having at least one female candidate in the top-10 positions. Hence the ranking in table 2 will be accepted as fair. However, if the required proportion is $p = 0.5$ this translates into needing at least one female candidate in the top-4, two in the top-7 and three in the top-9 positions. In this case the ranking will be reordered by FA*IR to meet the fairness constraints. Furthermore, our library implements the best possible adjustment of the desired significance level α. This is necessary, because the test for a representation like in table 1 is a multi-hypothesis test.

p \ k	1	2	3	4	5	6	7	8	9	10	11	12
0.1	0	0	0	0	0	0	0	0	0	0	0	0
0.3	0	0	0	0	0	0	1	1	1	1	1	2
0.5	0	0	0	1	1	1	2	2	3	3	3	4
0.7	0	1	1	2	2	3	3	4	5	5	6	6

Table 1: Example values of the minimum number of protected items that must appear in the top k positions to pass the ranked group fairness test with $\alpha = 0.1$. We call this an *MTable*. Table from [9]

Position	top 10 male	top 10 female	top 40 male	top 40 female
1 2 3 4 5 6 7 8 9 10				
f m m m m m m m m m	90%	10%	73%	27%

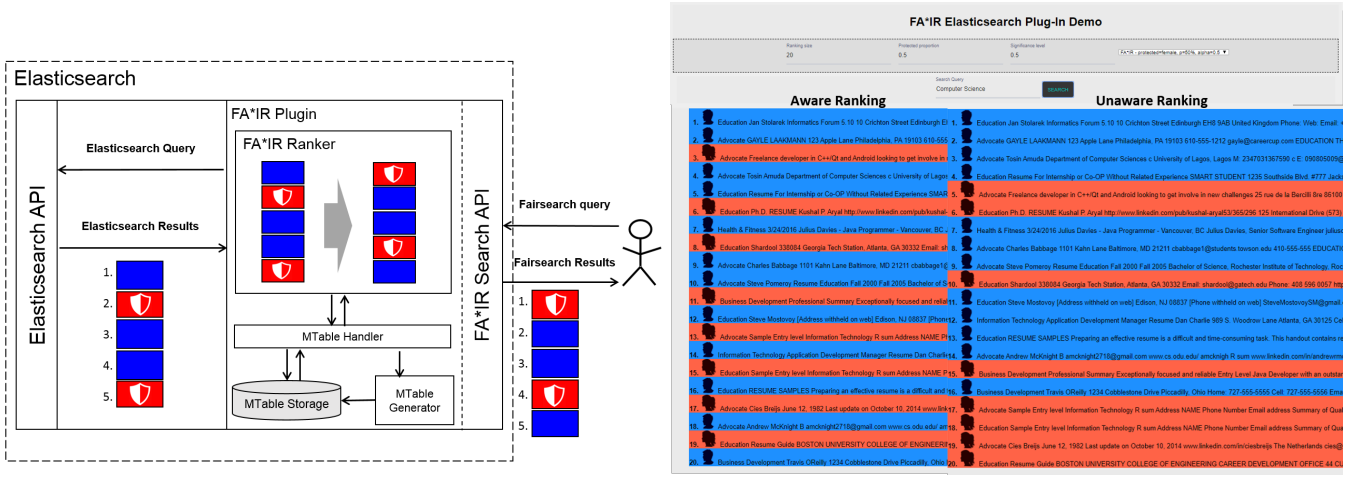
Table 2: Example of non-uniformity of the top-10 vs. the top-40 results for query "economist" in XING (Jan 2017). Table from [9]

3 FAIRSEARCH: THE DELTR PLUGIN

For the integration of DELTR into Elasticsearch we use the Elasticsearch Learning to Rank (LTR-ES) plugin [2]. The integration architecture is depicted on Figure 3. The logic consists of two phases: training and ranking.

Training. To apply DELTR at run-time for retrieval, LTR-ES needs a previously trained model that is uploaded into its model storage. Since training models is a very CPU intensive task that involves a lot of supervision and verification, it happens offline in a DELTR wrapper, which calls our stand-alone DELTR Python library to train a LTR-ES suitable model. The wrapper has to be provided

[2] https://elasticsearch-learning-to-rank.readthedocs.io/en/latest/



(a) Architecture of the FA*IR Elasticsearch Plugin

(b) Demo Application

Figure 2: (a) Architecture of the FA*IR Elasticsearch Plugin and (b) a Demo Webapp with the FA*IR Elasticsearh Plugin; red indicates protected items

with a training set, the training parameters and a name for the model. After training the wrapper calls the LTR-ES upload API, which stores the serialized model inside Elasticsearch's LTR plugin, making it available for up-coming retrieval tasks. Upon upload the wrapper specifies `model_name`, `type` (always DELTR), the model itself and the `feature_set` it was trained against. `feature_set` specifies query-dependent features, that tell LTR-ES which document features to use when applying the model.

Ranking. Elasticsearch ranks retrieved documents by applying re-scoring methods, because executing a query on the entire Elasticsearch cluster is very expensive. The system first executes a *baseline relevance* query on the entire index and returns the top N results. The `Rescorer` then modifies the scores for the top N results and returns the new list. DELTR implements Elastic's `Rescorer` interface, which it applies our previously learned weights to the document features of the top N results to produce the final ranking.

In the Rescorer, we have to specify two key parameters:

- `window_size` - the number of elements to re-score (usually N)
- `model` - the model name.

```
POST someindex/_search
{"query": {
"match": {
"_all": "Jon Snow"}},
"rescore": {
"window_size": 1000,
"query": {
"rescore_query": {
"sltr": {
"params": {
"keywords": "Jon Snow"},
"model": "deltr_model",}}}}}
```

The above code constitutes a sample rescore query using DELTR, in which we limit the result set to documents that match "Jon Snow". All results are scored based on Elasticsearch's default similarity (BM25). On top of those already somewhat relevant results we apply our DELTR model to get the best and fairest ranking of the top 1000 documents.

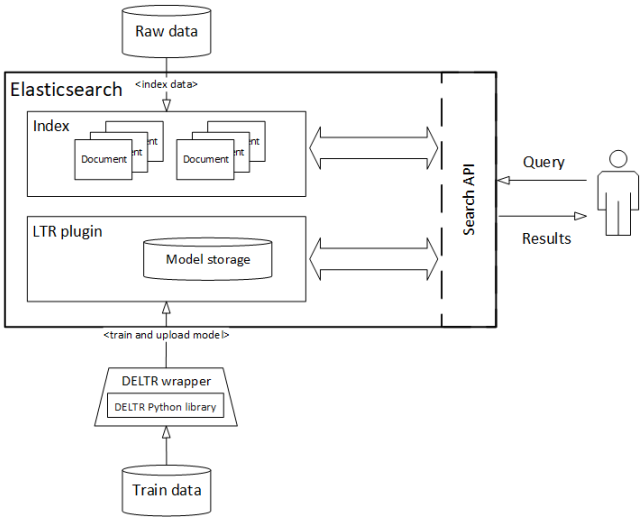


Figure 3: Architecture of the Elasticsearch plugin integration for DELTR

4 FAIRSEARCH: THE FA*IR PLUGIN

The FA*IR plugin enables Elasticsearch to process a search query and re-rank the result using FA*IR with parameters k, p and α. It extends the Elasticsearch API by two new endpoints and a *fair rescorer* JSON object, that contains the parameters for FA*IR. The two new endpoints create a new or request an existing `MTable`, an integer array that implements table 1. Once generated, MTables are persisted within Elasticsearch for further usage to avoid additional computational costs at search time. Figure 2a shows the control flow inside the plugin. A FA*IR query is passed to Elasticsearch, and

Algorithm 1: Construct MTable

INPUT: Ranking size k, minimum proportion p, significance α;
OUTPUT: MTable $M \in \mathbb{N}^k$
$M \leftarrow 0^k$;
$\alpha_c \leftarrow adjustAlpha(k, p, \alpha)$;
for $i := 1$ *to* k **do**
  $M_i \leftarrow inverseCDF(i, p, \alpha_c)$;
end
return M;

Elastic returns the standard result ranking to the plugin. The plugin then re-ranks the result according to the respective MTable that matches the input parameters p, k and α. Note that the execution of an *unaware* search query with all built-in features is still possible.

```
POST someindex/_search
{"from" : 0,
"size" : k,
"query" : { "match" : {"body" : q}},
"rescore" : {
"window_size" : k,
"fair_rescorer": {
"protected_key":"gender",
"protected_value":"f",
"significance_level": alpha,
"min_proportion_protected": p}}}
```

The components communicate via a REST API for HTTP requests and the above code represents a HTTP request to the plugin. With this Elasticsearch executes a regular search using the specified `query` object, the `match` object and query terms q. The result is re-ranked by the plugin using FA*IR, if the fairness constraints named in p, k and α are not met. First the `MTable Handler` will check if a MTable for parameters k, p, α already exists (right side of Figure 2a). If not, the plugin calls the `MTable Generator` to create it using algorithm 1 and stores it to `MTable Storage` as key-value pairs with key (k, p, α). We note that the MTable handler in Figure 2a is a simplification of Java classes and interfaces for the purpose of presentation. The FA*IR ranker (Figure 2a) re-ranks the Elasticsearch results according to the requested MTable (Figure 4) and returns them through a HTTP response in JSON format like a standard Elasticsearch result.

5 CONCLUSION

In this paper we presented FairSearch, the first open source API for search engines to provide fair search results. We implemented our previously published methods as stand-alone libraries in Python and Java and embedded those into a plugins for Elasticsearch. While the plugins are intended to be off-the-shelf implementations for Elasticsearch engineers, the stand-alone libraries allow great flexibility for those who use other technology such as Solr. This way we hope that fairness-aware algorithms will make their way faster into productive code and business environments to avoid bad social consequences such as discrimination in search results.

Acknowledgments. This project was realized with a research grant from Data Transparency Lab. Castillo is partially funded by La Caixa project LCF/PR/PR16/11110009. Zehlike is funded by the MPI-SWS.

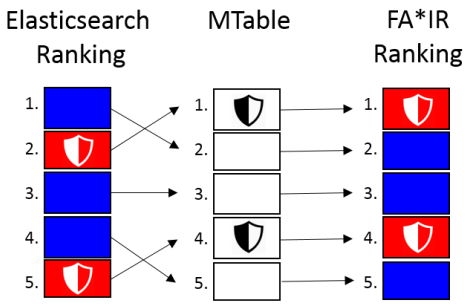


Figure 4: Re-ranking an Elasticsearch result according to a MTable; Shields indicate protected items

6 DEMONSTRATION

All libraries and plugins are available at https://github.com/fair-search. Our demo will consist of two main parts: First we will explain the architecture of FA*IR and DELTR by use of the figures in this paper. Next we will have a live coding session. For FA*IR we will code a mini example that is going to setup the algorithm in an Elasticsearch instance. It will show how to integrate the parameters p and α and how to further interact with the Elasticsearch plugin via search queries. An introduction into the FA*IR python library and Elasticsearch plugin is available on YouTube [11]. For DELTR we will use the synthetic dataset from section 2.1 to train a fair model. We will show how to upload this model into Elasticsearch using the DELTR-Wrapper and how it is used when issuing a search query.

Second using the results from the live coding session we will observe how the algorithms influence ranking results on a demo website (Figure 2b) for job candidate search, which operates on a resume dataset [1]. Lastly we will demonstrate how different input parameters for DELTR and FA*IR will affect the results and give intuition on best practice choices for the parameters. These two parts are also shown in the YouTube tutorial.

We require a large screen, so that attendees will be able to follow the coding examples from a distance.

REFERENCES

[1] 2018. Resumes Dataset with Labels. (2018). https://www.kaggle.com/iammhaseeb/resumes-dataset-with-labels Accessed: 2018-11-02.
[2] Toon Calders and Indrė Žliobaitė. 2013. Why unbiased computational processes can lead to discriminative decision procedures. In *Discrimination and Privacy in the Information Society*. Springer, 43–57.
[3] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. 2007. Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th international conference on Machine learning*. ACM, 129–136.
[4] Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard Zemel. 2012. Fairness through awareness. In *Proc. of ITCS*. ACM Press, 214–226.
[5] Moritz Hardt. 2014. How big data is unfair: Understanding sources of unfairness in data driven decision making. (2014).
[6] Stephen E Robertson. 1977. The probability ranking principle in IR. *Journal of documentation* 33, 4 (1977), 294–304.
[7] Ashudeep Singh and Thorsten Joachims. 2018. Fairness of exposure in rankings. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 2219–2228.
[8] Latanya Sweeney. 2013. Discrimination in online ad delivery. *Queue* 11, 3 (2013), 10.
[9] Meike Zehlike, Francesco Bonchi, Carlos Castillo, Sara Hajian, Mohamed Megahed, and Ricardo Baeza-Yates. 2017. FA*IR: A fair top-k ranking algorithm. In *Proc. of the 2017 ACM on Conference on Information and Knowledge Management*. ACM, 1569–1578.
[10] Meike Zehlike and Carlos Castillo. 2018. Reducing disparate exposure in ranking: A learning to rank approach. *arXiv preprint arXiv:1805.08716* (2018).
[11] Meike Zehlike and Tom Sühr. 2019. FA*IR in FairSearch – Tutorial. (05 01 2019). https://youtu.be/UXxTijlb5SY