Assignment of master's thesis

Title:	Counterfactual Learning-to-Rank in Personalized Search
Student:	Bc. Michael Kolínský
Supervisor:	Ing. Tomáš Řehořek, Ph.D.
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2023/2024

Instructions

Explore the field of search engines and specifically focus on the topics of:
- Counterfactual learning-to-rank [1]
- Position bias and selection bias of historical interactions and possible ways to eliminate them [2]
- Personalization of search results
- Success metrics for offline evaluation

Design and implement a framework for learning appropriate models using counterfactual learning-to-rank methods with estimated bias. Furthermore, design and implement appropriate document-specific features to help improve the performance of the models.

Conduct a set of offline experiments on the provided industrial datasets to compare the performance of retrieval on different models using the selected counterfactual learning-to-rank methods.

Present and discuss the results obtained using appropriate metrics.

[1] Harrie Oosterhuis and Maarten de Rijke. 2020. Policy-Aware Unbiased Learning to Rank for Top-k Rankings. In Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '20). Association for Computing Machinery, New York, NY, USA, 489–498. https://doi.org/

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 24 February 2023 in Prague.

10.1145/3397271.3401102
[2] Xuanhui Wang, Nadav Golbandi, Michael Bendersky, Donald Metzler, and Marc Najork. 2018. Position Bias Estimation for Unbiased Learning to Rank in Personal Search. In Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining (WSDM '18). Association for Computing Machinery, New York, NY, USA, 610–618. https://doi.org/10.1145/3159652.3159732

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 24 February 2023 in Prague.

Master's thesis

COUNTERFACTUAL LEARNING-TO-RANK IN PERSONALIZED SEARCH

Bc. Michael Kolínský

Faculty of Information Technology
Department of Applied Mathematics
Supervisor: Ing. Tomáš Řehořek, Ph.D.
May 4, 2023

Czech Technical University in Prague
Faculty of Information Technology
© 2022 Bc. Michael Kolínský. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Kolínský Michael. *Counterfactual Learning-to-Rank in Personalized Search*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Contents

Acknowledgments vii

Declaration viii

Abstract ix

List of abbreviations xi

1 Introduction 1
1.1 Structure of thesis 1

2 Theory and methods 3
2.1 Search engine 3
2.1.1 Search engine metrics 3
2.2 Learning to rank 5
2.2.1 Supervised Learning to Rank 9
2.2.2 Online Learning to Rank 11
2.2.3 Counterfactual Learning to Rank 11
2.2.4 Position bias estimation 22
2.2.5 Selection bias estimation 25
2.2.6 Item/User K-Nearest-Neighbors (*KNN*) 27
2.2.7 *RankNet*, *LambdaRank*, *LambdaMART* 30

3 Implementation 39
3.1 Framework 39
3.2 Search engines 40
3.3 Data description/preprocessing 42
3.4 Bias estimation 44
3.5 Training of models 45
3.5.1 Baseline 45
3.5.2 Linear 46
3.5.3 *LambdaMART* 47
3.6 Feature engineering 47
3.7 Counterfactual evaluation 48

4 Experiment design 51
4.1 Expectation-Maximization experiments 51
4.2 Personalization experiments 52
4.3 Unbiased rankers experiments 52

4.4 Comparison of rankers 53

5 Discussion of the experimental results 55

5.1 Expectation-Maximization experiments results 55

5.2 Personalization experiments results 57

5.3 Unbiased rankers experiments results 63

5.4 Comparison of rankers results 67

6 Conclusion 75

6.1 Future work 76

Content of the enclosed media 81

List of Figures

2.1 A diagram showing the historical data 7
2.2 A decision treen showing Jaccard index $J@k$ evaluation of a document . . 9
2.3 An example of user behavior according to Position-based Model 14
2.4 A probability tree diagram showing the used User Model 14
2.5 A rating matrix $\mathbf{R}$ denoting the important elements 30
2.6 A ranking with depicted lambdas 34

3.1 A graph of the framework in which we conduct experiments 41

5.1 Comparison of data log-likelihood progression with and without weight enhancement 56
5.2 Comparison of the *EM* algorithms with the improvement on the ranking performance for the *e-commerce* platform 57
5.3 Comparison of the *EM* algorithms with the improvement on the ranking performance for the *VOD* platform 58
5.4 Dependence of $DCG@6$ on parameter k used in collaboration-filtering algorithms for *e-commerce* platform 59
5.5 Dependence of DCG on parameter k used in collaboration-filtering algorithms for *VOD* platform 60
5.6 Comparison of different personalization approaches to the ranking performance for the *e-commerce* platform 61
5.7 Feature importance of the *Position* model for the *e-commerce* platform . . 62
5.8 Feature importance of the *Selection* model for the *e-commerce* platform . 62
5.9 Comparison of different personalization approaches to the ranking performance for the *VOD* platform 64
5.10 Feature importance of the *Position* model for the *VOD* platform 65
5.11 Feature importance of the *Selection* model for the *VOD* platform 65
5.12 Comparison of different debiasing approaches on the ranking performance of the models for the *e-commerce* platform 66
5.13 Comparison of different debiasing approaches on the ranking performance of the models for the *VOD* platform 68
5.14 Comparison of different ranking approaches for the *e-commerce* platform . 69
5.15 Comparison of different ranking approaches for the *VOD* platform 71
5.16 Comparison of different ranking approaches for the *VOD* platform with policy-oblivious estimator 72
5.17 Dependence of $DCG@6$ on the number of iterations in the learning process for train and test data for the *e-commerce* platform 72

5.18 Dependence of *DCG* on the number of iterations in the learning process for training and test data for the *VOD* platform 73
5.19 LambdaMART feature importances of *e-commerce* platform 74
5.20 LambdaMART feature importances of *VOD* platform 74

I would like to thank Ing. Tomáš Řehořek, Ph.D. for the time given and valuable advice. I thank Recombee for the opportunity to use the data and computational resources used to train the models. Last but not least, I would like to thank my family for their kindness and support throughout my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Czech Technical University in Prague has the right to conclude a licence agreement on the utilization of this thesis as a school work pursuant of Section 60 (1) of the Act.

In Prague on May 4, 2023 ....

Abstract

This thesis explores the field of search engines, with a particular emphasis on Counterfactual Learning to Rank, position bias, and document selection bias in historical interactions, personalization of search results, and success metrics for offline ranking evaluation. The study aims to design and implement a framework to learn suitable models utilizing Counterfactual Learning to Rank methods that are used to compare the ranking performance of the models and train unbiased models. Additionally, some document-specific search features as well as user-specific features are proposed to enhance the performance of these models. Offline experiments were conducted on two significantly different provided industrial datasets to assess the retrieval performance of various models using the selected methods. Part of the experiments are dedicated to the comparison of different personalization approaches. The performance of these models was evaluated using appropriate success metrics for offline counterfactual evaluation, as well as other offline evaluation metrics. In conclusion, this research contributes to search engine optimization. The study's findings have implications for the personalization of search results and the development of more effective search engine algorithms.

Keywords search engines, Counterfactual Learning to Rank, position bias, document selection bias, historical interactions, personalization, ranking success metrics, offline evaluation

Abstrakt

Tato práce se zabývá oblastí vyhledávačů, s důrazem na kontrafaktuální metody *Learning to Rank*, zkreslení způsobené pozicí dokumentů a zkreslení způsobené výběrem dokumentů v historických interakcích, personalizaci vyhledávacích výsledků a úspěšnostní metriky pro offline vyhodnocení seřazených dokumentů. Cílem studie je navrhnout a implementovat *framework* pro učení vhodných modelů s využitím kontrafaktuálních *Learning to Rank* metod, které se používají pro srovnání úspěšnosti řazení těchto modelů a k jejich tréninku. Navíc jsou navrženy některé specifické vyhledávací příznaky pro dokumenty i uživatele, které mají zlepšit úspěšnost těchto modelů. Off-line experimenty byly prováděny na dvou významně odlišných průmyslových datasetech s cílem posoudit úspěšnost řazení různých modelů pomocí vybraných metod. Část experimentů je věnována srovnání různých příznaků k personalizaci pro konkrétního uživatele. Úspěšnost těchto modelů byla hodnocena pomocí vhodných úspěšnostních metrik pro offline kontrafaktuální vyhodnocení i s dalšími metrikami pro offline vyhodnocení. Toto dílo přispívá k optimalizaci vyhledávačů. Zjištění mohou být použita pro personalizaci vyhledávácích výsledků a vývoj efektivnějších modelů pro vyhledávání.

Klíčová slova vyhledávače, kontrafaktuální učení se řadit, zkreslení dáno pozicí dokumentů, zkreslení dáno výběrem dokumentů, historické interakce, personalizace, metriky řazení, offline vyhodnocení

List of abbreviations

ARP	Average Relevant Position
CTR	Click Through Rate
DCG	Discounted Cumulative Gain
EM	Expectation-Maximization
IDF	Inverse Document Frequency
IR	Information Retrieval
KNN	K-Nearest-Neighbors
MART	Multiple Additive Regression Trees
MRR	Mean Reciprocal Rank
NDCG	Normalized Discounted Cumulative Gain
VOD	Video-On-Demand
GBDT	Gradient Boosted Decision Trees

Chapter 1
Introduction

Today, the amount of data on the Internet is enormous and is still growing very fast. It is a difficult task to find the information we need. This large amount of data calls for effective search capabilities. Search engines are crucial in making the Internet an easy-to-use, navigable space. They allow users to quickly and efficiently search through large amounts of information to find the specific content they are looking for. Without search engines, users would have to manually browse through countless data to locate the information they need, resulting in a time-consuming and frustrating experience. In general, search engines have become part of our daily lives and have revolutionized the way we access and consume information on the Internet. From the perspective of the search engine operator, the quality of the results could improve the Click Through Rate (*CTR*), the number of purchases, the popularity of the web page, or increase the duration of user engagement. Thus, it makes sense to optimize the search system.

This thesis will focus on a search system for an e-commerce and Video-On-Demand (*VOD*) platform. The engine is given a query, and the engine returns a subset of all indexed documents ranked by a relevancy score from the most relevant to the least relevant. This solution will be the baseline approach. The main effort is to train some ranking models that are able to rerank the set of documents returned so that the new ranking is better from the perspective of the evaluation metrics used. Using Counterfactual Learning to Rank methods, the models will be trained using historical interactions between users and the documents, products in an e-commerce platform, or movies in a *VOD* platform. We will also estimate the biases that appear in the interactions and apply this estimate to eliminate them. To improve ranking performance, we present new document features and some of them are personalized to a user. We design and conduct a set of experiments that compare the models using the selected offline success evaluation metrics with each other and with the baseline. Finally, we present the results and discuss them.

1.1 Structure of thesis

In Chapter 2 we present and describe all the theories and methods we use in the following chapters. In Chapter 3 is described the whole framework in which we conduct

experiments. In this chapter, we also present the concrete implementation of the presented methods. Chapter 4 describes the experiments we conduct. In Chapter 5, we describe and show the outputs of the conducted experiments. We also discuss the results. The last Chapter 6 contains a summarization and conclusion of the experiments and findings.

Chapter 2
Theory and methods

In this chapter, we present all the methods and concepts that we will use in the next chapters. We start by presenting how a search engine creates a candidate set of documents with scores. The documents can be sorted solely by the scores from the search engine. To get a better user experience from the entire search system, we present some Learning to Rank methods that use the scores with additional data to rerank the documents.

2.1 Search engine

A search engine is a specific type of Information Retrieval (*IR*) system that is designed to enable users to quickly and easily search and retrieve relevant information from large collections of data based on a query. Data could be, for example, web pages, videos, documents, etc. In this thesis, we are interested in a search engine that contains documents in the database. Although a good search engine is a very complex system, in this thesis, we are not interested in the exact implementation. We are interested in how the engine creates the document result set that is returned based on a query and how the relevancy scores are computed. Each document has some text fields. When we send the search engine a query, the search engine performs a full-text search over all the fields of the documents and returns the relevant ones for the given query. Each returned document has a score, where the score represents the relevance to the given query based on a metric. A full-text search means searching for words or phrases within a text by searching for matches throughout the whole text. We start with a definition of a document used in the work.

▶ **Definition 2.1** (Document)**.** *A document is a set of text fields with a label for each field.*

Documents could be, for example, products of an e-shop with title and description text fields.

2.1.1 Search engine metrics

Although there are many metrics available that could be used, we will use only the Levenshtein edit distance and the *BM25* metric. A scoring function is applied to all text

fields in a document, and the maximum score is returned. A higher degree of query match is indicated by a higher score. Given a query, the set of results created by a scoring function consists of all documents with scores greater than 0. For each document in the set, we say that the document matches the query.

▶ **Definition 2.2** (Levenshtein edit distance)**.** *Let* $A = a_1, a_2, \dots, a_n, B = b_1, b_2, \dots, b_m$ *be strings of lengths* n *and* m*, respectively, and let* Σ *be an alphabet* $a_i, b_j \in \Sigma$ *for every* $1 \leq i \leq n, 1 \leq j \leq m$*. The edit distance between* A *and* B *is the minimum number of the following operations required to make the string* B *from the string* A*.*

- *Insert* $c \in \Sigma$ *into* A*.*
- *Delete* a_i *from* A*.*
- *Replace* a_i *with* b_j*.*

For example, the edit distance between strings $KITTEN$ and $SITTING$ is 3. The operations are replace, replace, and insert:

$$KITTEN \rightarrow \mathbf{S}ITTEN \rightarrow SITT\mathbf{I}N \rightarrow SITTIN\mathbf{G}.$$

The Levenshtein edit distance is defined just between two strings, not between a document and a string. Direct generalization to compute the Levenshtein distance between the query and each text field of the document and return the maximum is not applicable in a full-text search. To use the edit distance between a document and a string, we compute the distance for every document text field substring of the same length in characters as the query. This is a very time-consuming operation, and thus an approximation is used. The exact method is not part of this thesis, and thus will not be discussed. The distance could easily be converted to a score where larger means better by subtracting the computed distance from the maximal distance. The maximum of the Levenshtein distance is the length of the longer string.

BM25 is a scoring function that gives a document a score based on a query. The function uses each term in the strings separately and is thus not sensitive to term proximity. First, we need some definitions [1].

▶ **Definition 2.3** (Average document length)**.**

$$avgdl := \frac{\sum_{D \in \mathcal{D}} |D|}{|\mathcal{D}|}, \tag{2.1}$$

where $|D|$ is the number of terms in document D. A term is a sequence of characters separated by whitespace. $\mathcal{D}$ is a collection of documents and $|\mathcal{D}|$ is the number of documents in the collection $\mathcal{D}$. We have data from two platforms, each platform contains its own set of documents $\mathcal{D}$. The data will be described further in the thesis.

▶ **Definition 2.4** (Inverse document frequency)**.**

$$IDF(t) := \begin{cases} \log \dfrac{|\mathcal{D}|}{\sum_{D \in \mathcal{D}} \mathbb{1}_{t \in D}}, & \textit{if } t \textit{ is present in at least one document} \\ 0, & \textit{otherwise,} \end{cases} \tag{2.2}$$

where $t \in D$ means that the term t is present in a text field of document D. And $\mathbb{1}$ is an indicator function that is 1 when the condition in the subscript is true and false otherwise. By log we mean the $\log_2$ logarithm, and we use this notation throughout the thesis. Inverse Document Frequency is a measure of how important a term is in a collection of documents. The intuition behind *IDF* is that terms that appear in many documents are less informative or discriminative than terms that appear in fewer documents. Thus, the *IDF* of a term is higher when it appears in fewer documents and lower when it appears in more documents. By taking the logarithm, we can obtain a smoother weighting function that avoids extreme values for rare terms.

▶ **Definition 2.5** (Term frequency). *Term frequency $f(t, D)$ is the frequency of occurrence of the term t in document D.*

There are multiple definitions of the *BM25* function, but we will use the following:

▶ **Definition 2.6** (*BM25*).

$$BM25(D, Q) := \sum_{i=1}^{n} IDF(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{avgdl})} \tag{2.3}$$

In this formula, D represents a document and Q is a query. n is the number of query terms, q_i represents the i-th query term. $|D|$ is the length of the document D. k is a tuning parameter that controls the scaling of the term frequency, and b is a parameter that controls the effect of document length normalization. Note that the $BM25$ is defined between a query and a document.

In this section, we have shown how to obtain documents that match a query. Each document from this set must have a non-zero score and the score is calculated based on the defined metrics. The following chapters will describe the concrete search engine implementation used, including how the specific metrics are utilized. We have not shown how to obtain the set of documents efficiently, but it is beyond the scope of this thesis.

2.2 Learning to rank

According to [2], the Learning to Rank task is defined as: "Learning to rank for Information Retrieval (*IR*) is a task to automatically construct a ranking model using training data, such that the model can sort new objects according to their degrees of relevance, preference, or importance." We will use the Learning to Rank apparatus to improve the basic ranking induced by a search engine from the perspective of a ranking quality measure. There are several approaches to the Learning to Rank that we briefly present, but our main focus is on the Counterfactual Learning to Rank approach.

2.2.0.1 Problem definition

First, we describe in more detail the data we use for training and evaluation. All data are collected in the past, and the data for both platforms are different. We have a set of queries $\mathcal{Q}$ that the users sent. A query $Q \in \mathcal{Q}$ is a triplet (q, t, u), where q is a search string that we pass to a search engine, t is a timestamp of when the query was issued, and u is a user who wrote the query. In the thesis, we mainly use the triplet

Q as a whole and not the individual values of the triplet. For each query, we have a set of candidate documents $\mathcal{D}$ that match the query string q and are obtained from a search engine. The $\mathcal{D}$ is assumed to contain every document that could be relevant to the given query. Documents that are not in the set are not relevant to the user, and the user would not click on that document. There are two types of ranking $\bar{R}$ and R. $\bar{R}$ is a ranking of documents that were shown to a user in the past. The ranking $\bar{R}$ is a tuple of documents:

$$\bar{R} := (D_1, D_2, D_3, \dots), D_i \in \mathcal{D}.$$

Each document in the ranking has a position in the vector called a rank. The vector does not necessarily contain all the documents in the set $\mathcal{D}$. This is common for a ranking system showing users just the top-k relevant documents. Now we define the ranking R that is not historical and is created using a model f_θ. Let f_θ be a model with a set of parameters θ able to give each document D in a set of documents $\mathcal{D}$ a score $f_\theta(D)$. The ranking R of the document set $\mathcal{D}$ is defined as:

$$R := (D_1, D_2, D_3, \dots), D_i \in \mathcal{D},$$

where

$$f_\theta(D_1) \geq f_\theta(D_2) \geq f_\theta(D_3) \geq \dots,$$

and every document is present in the tuple R if and only if it is present in the set $\mathcal{D}$. Although this ranking R contains all the documents, it is possible that only the top-k documents with the lowest rank can be shown to users or used in a ranking metric. We also have the user's clicks. When a ranking $\bar{R}$ was shown to a user, the user decided whether to click on a document or not. A user could click on multiple documents or not click at all. So, for all rankings issued to a user, we have a click label denoted $c(D)$, which is 1 if the user clicked and 0 otherwise. The data we want, but do not have in the thesis, are relevance labels. For each ranking issued to a user, we want labels that indicate how much a document is relevant. A label could be a real number indicating relevance, a number of stars from 1 to 5, where 5 stars are the most relevant, or a binary label. Labels can be collected from human judges. Labels can also be just hypothetical, and we call it the full information setting, and we denote the true label by $y(D)$. The click labels and relevance labels are assigned to a concrete user and query, but we do not indicate it in the notation, since we will use the notation where we use a single query and user. So there will be no ambiguity.

Figure 2.1 is a diagram showing the historical data. On the left side, there are users who write queries to a search system. The queries are then passed to a search engine that returns a set of candidates for each query with a score. The documents are then passed to a Learning to Rank model that reranks the documents. The model used in the past is optional, the documents can be sorted just by the scores returned by the search engine. The model can be arbitrary, currently, we do not have any requirements for the model. The output of the model is a ranking shown to the end user. The rankings in the figure are the $\bar{R}$ rankings with the bar. Users decide which documents to click on, and the clicks are further recorded by the system. A click on a document is denoted by bold text in the diagram. Based on this data, we try to train a new model.

The goal of Learning to Rank is to find the parameters θ so that the final rankings R are the most optimal from the perspective of a ranking metric [3]. In further sections, we

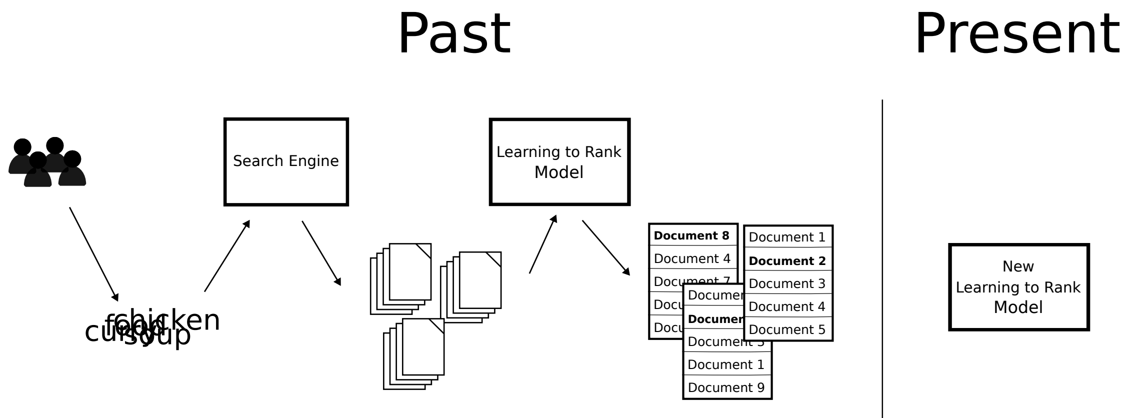

Figure 2.1 A diagram showing the historical data

show the concrete ranking models used and their parameters. We also show the concrete metrics for which we optimize the model parameters.

There are three primary approaches to obtaining the optimal parameters θ:

- Supervised Learning to Rank
- Counterfactual Learning to Rank
- Online Learning to Rank

Each approach has some advantages and disadvantages. We will discuss the basics but in-depth only the Counterfactual Learning to Rank approach. But now we have to introduce the ranking evaluation metrics that we will use.

2.2.0.2 Ranking evaluation

We need to evaluate a ranking created by a model to tell whether the ranking is good or not. There are many ranking quality measures available in *IR*. We will evaluate our ranking model f_θ applied to a set of documents $\mathcal{D}$ with relevance $y(D)$ to the document D as:

$$\Delta(f_\theta, \mathcal{D}, y) := \sum_{D \in \mathcal{D}} \lambda(rank(D|f_\theta, \mathcal{D})) \cdot y(D), \tag{2.4}$$

where $rank(D|f_\theta, \mathcal{D})$ is the rank of document D in the ranking R of the document set $\mathcal{D}$ induced by f_θ. The model f_θ gives each document a score and the documents are sorted by score. The function λ is a rank-weighting function that helps us to create several different measures, but we use only the following. Note that we use the true relevance label $y(D)$ that we do not have [4].

Discounted Cumulative Gain (*DCG*) The relevance is weighted by the inverse of the logarithm of rank p. Therefore, for each position in a ranking, we have a different weight that decreases with the position. There is a normalized version called Normalized *DCG* (*NDCG*) where the result is divided by an ideal *DCG*. The ideal *DCG* is *DCG* of a ranking where relevant documents from the whole set of candidates are at the top positions sorted from the most relevant to the least relevant. The ideal

DCG is the maximum value of the DCG between all possible rankings. The resulting value is then between 0 and 1.

$$\lambda(p) = -\frac{1}{\log_2(p+1)} \tag{2.5}$$

Average relevant position (*ARP*) The relevance is weighted by rank p. The weight of each position increases with the position. So a relevant document at position hundred is more important than a relevant document at position ten.

$$\lambda(p) = p \tag{2.6}$$

For the ARP and DCG metric, a suffix $@k$ can be appended to signal that we are interested only in the top-k documents of the ranking. We calculate it by multiplying the $\lambda(p)$ by the expression $\mathbb{1}_{p \leq k}$. For example, for the $DCG@k$ metric:

$$\lambda(p) = -\frac{\mathbb{1}_{p \leq k}}{\log_2(p+1)}.$$

This is a measure of a single ranking, and when we want to measure the performance of a model over a whole dataset of rankings, we will use the average value.

Now, we present different recommendation measures that are used to evaluate a recommendation algorithm. The presented measures are not sensitive to the ranks of documents, they only use the set of documents in a ranking to a fixed position k. The first is $recall@k$, since we do not know which documents are relevant, we use just click labels c. Often, we are interested only in the top-k documents of a ranking. We want to know whether a document D in a historical ranking $\bar{R}$ that was clicked is recommended in the first k positions of a newly created ranking R by a model f_θ that is being evaluated. $Recall@k$ is defined across a whole dataset as:

$$recall@k := \frac{\sum_{Q_i \in \mathcal{Q}} \sum_{D \in \mathcal{D}_i} c(D) \cdot \mathbb{1}_{rank(D|f_\theta, \mathcal{D}_i) \leq k}}{\sum_{Q_i \in \mathcal{Q}} min(k, \sum_{D \in \mathcal{D}_i} c(D))}. \tag{2.7}$$

It is the number of documents clicked by a user and recommended by the model f_θ to position k divided by the total of documents clicked on the entire set of queries $\mathcal{Q}$.

Now we define a model performance measure based on the Jaccard index [5]. The Jaccard index is a measure of similarity between two sets. It is defined as the size of the intersection of the sets divided by the size of the union of the sets:

$$J(A, B) := \frac{|A \cap B|}{|A \cup B|}. \tag{2.8}$$

The value is always between 0 and 1. When the value is 1, the sets are equal. When the value is 0, the sets do not have any element in common. The similarity is defined only for two non-empty sets.

We define two sets that are bound to a single query Q. A set named *clicked* denotes all documents that were clicked for the query. A set named $matched@k$ is a set of documents created by an intersection of documents in a ranking $\bar{R}$ that was shown to a user and a set of documents in the first k positions of a ranking R that is created

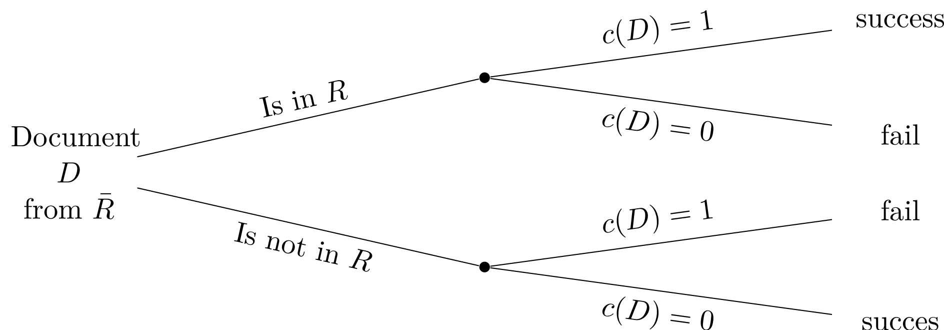

Figure 2.2 A decision treen showing Jaccard index $J@k$ evaluation of a document

by a model f_θ. In other words, the set $matched@k$ is a set of documents that were recommended in a historical ranking $\bar{R}$ and are also in a current ranking R (in the first k positions). A $J@k$ is defined as the Jaccard index between these two sets:

$$J@k := J(clicked, matched@k) = \frac{|clicked \cap matched@k|}{|clicked \cup matched@k|}. \tag{2.9}$$

Intuitively, when we recommend a document in the top-k positions and the document is the same as in the historical ranking, we know if we have succeeded or failed. The user clicks on the document or not. On the other hand, when we do not show a clicked document to the kth position, we have failed. When we recommend a document that is not present in the historical ranking $\bar{R}$, then we do not know. We can see a similar situation in a decision tree in Figure 2.2. We will use only those ranking $\bar{R}$ with clicks, therefore, solving the case where the Jaccard index is not defined. We will discuss how and why we use these measures in the next chapter.

2.2.1 Supervised Learning to Rank

As the name suggests, we need some labels on the documents. To use the supervised approach, we need for each user-issued query in our dataset [3]:

Documents A set of documents that match the query.

Labels For each of the documents, we need a *quality* that indicates the relevance of this document to the given user and the query issued.

Labels for each document are usually obtained from human judges, so they are expensive and difficult to obtain. For example, not all websites that want to improve the relevancy of their search results are able to get enough annotated records to train a useful model. Even when labels are collected, they do not evolve over time, so they are outdated.

There are pointwise, pairwise, and listwise approaches used to obtain the model parameters θ or evaluate a model. The intuitive difference between the functions is how many documents are used to get a single error, the errors are then summed up. These approaches are not limited to supervised Learning to Rank but we present them here.

2.2.1.1 Pointwise approach

The pointwise approach used in Learning to Rank is the standard regression or classification problem where we try to predict the label directly. For the regression task, we want to minimize the error between the true and predicted labels. For example, it is the same as predicting the price of houses based on some features. Classification could be used, for example, when there are only binary relevance labels, relevant and irrelevant. Documents can be sorted by probability of relevance, or documents deemed relevant could be placed at top positions in random order.

Because the models are optimized only for the relevancy of a single document to the query, the resulting ranking induced by these models is not optimized for the correct ordering between the documents. The performance of this method is usually the worst of the three [6].

Predicting the probability of relevance is a more difficult problem than ranking documents. If we have the true probability of relevance, we can sort the documents in an optimal order, but if we have an optimal order, we do not have relevance probabilities.

We will show a pointwise loss in Equation 2.38 where we try to predict the likelihood of relevance in a binary relevant–irrelevant scenario. A pointwise evaluation measure can be, for example, a sum of differences between predicted and true labels in a setting where the labels are real numbers from 0 to 1.

2.2.1.2 Pairwise loss

The pointwise approach loss is not interested in other documents, takes into account only one document, and is not dependent on any other document. The pairwise approach is interested in the relative score of a pair of documents. It tries to put a pair of documents in the correct order but without dependency on other pairs. The problem with this method is that it treats every pair of documents equally important. Often we are interested only in a top-k ranking where the ranks of pairs above k should not be optimized at the cost of the pair above [3].

An example of pairwise loss is given in Definition 2.72. This loss is also an evaluation measure, where we are interested in the number of pairs of documents with different relevance labels that are in the wrong order. Meaning that the more relevant document has a higher rank than the other document.

2.2.1.3 Listwise loss

The listwise optimizes the ranking directly. That is, the quality measure or loss function is interested in all the documents at once. They are not differentiable since they use the rank of a document, and thus the range of values consists of integers. An example of a listwise evaluation measure is DCG, where we use the ranks of the documents directly. An example of a listwise loss function is Equation 2.38 where we optimize the DCG measure.

2.2.2 Online Learning to Rank

The online Learning to Rank approach does not use historical data. Instead, the ranker is optimized based on direct interactions with the users. It is done by training a ranking model in real-time as users interact with the ranking system. It is called online since the learning of ranker occurs as queries are received and results are presented, rather than being precomputed offline. Using user feedback, such as clicks, the model is optimized to provide better results. An approach that is commonly used in online Learning to Rank is the multi-armed-bandit model. Since we are interested in the counterfactual approach, we will not discuss this topic further [3].

2.2.3 Counterfactual Learning to Rank

Counterfactual Learning to Rank uses historical data to learn the parameters θ offline. The data are presented in Section 2.2.0.1. The system does not directly interact with the user, as opposed to the online approach, and the system does not have the true relevance labels, as in the supervised approach. User interactions such as clicking on a document, purchasing a product, adding a product to a cart, etc. provide implicit feedback on user preferences and are collected in the past. This method uses these data to learn a model. The focus of this thesis will be solely on the clicks made by users on a document when a ranking is shown to a user. The name *counterfactual* is based on the counterfactual evaluation of a ranking model that we present in the following section [3].

2.2.3.1 Counterfactual evaluation

Suppose that we have a ranking model f_θ with parameters of the model θ and our objective is to assess how performance would look in a production setting using a specific metric before deployment. For the rest of the thesis, we assume we have only binary document relevance labels – 1 for relevant and 0 for non-relevant. If we had the true document relevance function y, we could compute the following formula over a set of queries $\mathcal{Q}$ and a set of corresponding matching documents $\mathcal{D}_i$ for each $Q_i \in \mathcal{Q}$ [4, 6]:

$$\mathcal{L} := \frac{1}{|\mathcal{Q}|} \sum_{Q_i \in \mathcal{Q}} \Delta(f_\theta, \mathcal{D}_i, y). \tag{2.10}$$

In the real world, we usually do not have true labels, but we have user clicks. When a user clicks on a document, does that mean that the document is relevant? No, a click could happen randomly on a non-relevant document, and when a user does not click on a document, it does not mean that the document is irrelevant. When a document is higher in ranking, it is more likely that the document is clicked. Even when a document is not visible, the document could not be clicked. The interactions can not be used directly, but certainly, the clicks are related to relevance. There are two major problems:

Noise When a user randomly clicks on a non-relevant document or does not click on a relevant document, then it would be at a random document, and this should not happen systematically. Averaging over large data, we can infer a preference, so averaging will cancel the effect [3].

Biases If a relevant document is shown in the 500th position, then the probability of a click is very low compared to a document in the first position. This systematic bias will occur for every query, so the effect will not be canceled out by averaging over a large dataset. A bias influences how we collect the interactions. This phenomenon is called a position bias, and we will define the biases in subsequent sections.

In this thesis, we will handle only position bias and document selection bias in historical interactions. There are more types of biases that we do not address, such as trust bias. A bias is a systematic error or deviation in the interaction data that was caused by user behavior or the way the rankings are presented to the user.

If we show users for a given query a random permutation of the documents, the relevant documents that are at a random position will receive a different number of clicks. For example, when a relevant document is in the first position, the number of clicks will be different from that of a relevant document in position ten. This phenomenon is called position bias. This effect arises from the user's behavior and from how the documents are presented to the user.

Until now, we assumed that we showed the user all the documents that match a given query. It is often not the case when we want to show a user just the top-k best documents. Some of the relevant documents could be above the k threshold. Or even there could be more than k relevant documents. Users are not able to click on the documents, so some of the clicks are not collected, and thus the interactions are biased. This phenomenon is called document selection bias. This bias emerges purely from the way the documents are presented to the user: the documents were not shown, so they cannot be clicked.

Both biases will become more obvious when we calculate the expected value of a metric in a simplified formalization of user behavior called a User Click model [7].

2.2.3.2 User Click Model

To formalize the biases, we will use a simplified model of how users browse a ranking and collect clicks. Although many possible models can be chosen, we will use the following Position-based Model.

In this model, the user click depends on whether the user has examined the document and whether the document is relevant. A document's examination (or observation) could be seen as if the user saw the document in the ranking. There are three Bernoulli variables. The first is the $c(D)$ variable that represents a user click and is dependent on a $o(D)$ variable that represents whether the user has examined the document and also on the $y(D)$ relevance variable. The variable $c(D)$ is fully observed — we know if a user clicked or not in every ranking. The variables $o(D)$ and $y(D)$ are observed only partially when a user clicks. In this model, every user clicks on a document if and only if the document is observed and relevant [3, 6]:

$$c(D) = \mathbb{1}_{y(D) \wedge o(D)}. \tag{2.11}$$

When a user does not click, we do not know if the document was not observed or irrelevant. Since we do not know if a document was observed or is relevant, we model it using probabilities. We denote the probability that the user will examine a document D in position k in a ranking $\bar{R}$ as $P(o(D) = 1|\bar{R}, D, k)$. We can omit k in the condition

since the position k is already hidden in $\bar{R}$ and D and thus redundant:

$$P(o(D) = 1|\bar{R}, D, k) = P(o(D) = 1|\bar{R}, D). \tag{2.12}$$

The probability that a document D is relevant to a query string q as $P(y(D)|q, D)$. And the probability of a click $c(D)$ on document D in a ranking $\bar{R}$ for query q in position k as $P(c(D) = 1|q, \bar{R}, D)$. These probabilities are the parameters of the Bernoulli distribution. The model assumes that the variables depend only on the given conditions and nothing else. Note that the probability of relevance depends only on the query string without a user and timestamp. We will use the dependency on a user in further sections, where we use personalization of the rankings for a concrete user to improve performance, but in this model, the relevance is dependent only on the query. The documents are relevant to a query without dependence on a user. The variables $y(D_i)$ and $o(D_j)$ are assumed to be independent for all documents D_i and D_j in a ranking. So, the probability of a click is:

$$P(c(D) = 1|q, \bar{R}, D) = P(o(D) = 1|\bar{R}, D) \cdot P(y(D) = 1|q, D). \tag{2.13}$$

The equation could be rewritten as:

$$\begin{aligned} P(c(D) = 1|q, \bar{R}, D) &\stackrel{1}{=} P(c(D) = 1 \wedge o(D) = 1|q, \bar{R}, D) \\ &= P(c(D) = 1|o(D) = 1, q, \bar{R}, D) \cdot P(o(D) = 1|\bar{R}, D) \\ &\stackrel{2}{=} P(y(D)|q, D) \cdot P(o(D) = 1|\bar{R}, D). \end{aligned} \tag{2.14}$$

Equation marked 1 is true since every clicked document must be observed before clicking. In other words, the set of observed documents is a superset of clicked documents. Since $o(D) \Leftrightarrow y(D)$ in the case $o(D) = 1$ is given, equality 2 is correct.

Whenever we see a click, we know that the user has examined the document and that the document is relevant. For now, we assumed in Equation 2.13 that there is no click noise — clicking on a non-relevant document.

In Figure 2.3 we can see an example of how a user can browse a ranking and click on documents. The first document is relevant and observed, and thus clicked. The second document is relevant, but not observed for a reason, and thus not clicked. In the third position, there is a non-relevant document that is examined but not relevant for the query, and thus not clicked. In the fourth and fifth positions, there are two non-relevant documents that would not be clicked even if they are observed. Even when there are three relevant documents, the user clicked only on the first document, so we do not have the information that the documents D_2 and D_6 are relevant.

Whether these assumptions are valid or oversimplified depends on the application. We believe that these assumptions are reasonable for our search engine application, and we show the results in Chapter 5. In Figure 2.4 there is a probability tree of the User Model. We can see that when a document is clicked, we know that the document was observed and is relevant. The same statement does not hold in the case of non-click.

Historical rankings were shown to users using a ranking model that was deployed at the time clicks were collected. There could be a single model, an ensemble of models, or even the models could be changed in time. Even some A/B testing could be done

$y(D_1) = 1$	$o(D_1) = 1$	Document 1	$c(D_1) = 1$
$y(D_2) = 1$	$o(D_2) = 0$	Document 2	$c(D_2) = 0$
$y(D_3) = 0$	$o(D_3) = 1$	Document 3	$c(D_3) = 0$
$y(D_4) = 0$	$o(D_4) = 0$	Document 4	$c(D_4) = 0$
$y(D_5) = 0$	$o(D_5) = 0$	Document 5	$c(D_5) = 0$
$y(D_6) = 1$	$o(D_6) = 0$	Document 6	$c(D_6) = 0$

Figure 2.3 An example of user behavior according to Position-based Model

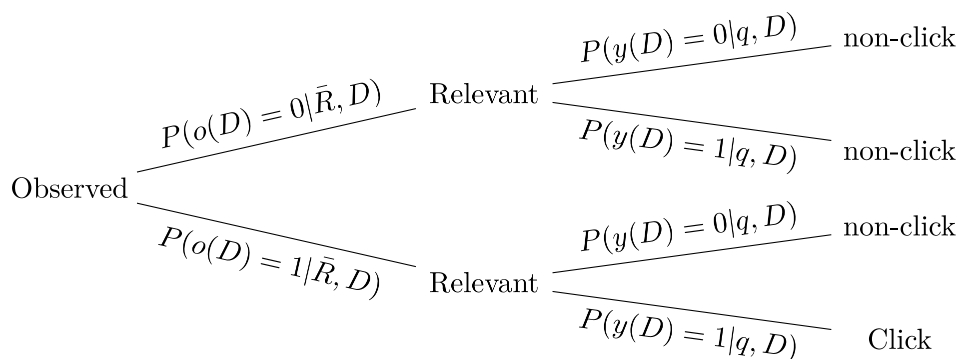

Figure 2.4 A probability tree diagram showing the used User Model

so that two different users could receive rankings from different models. We model the ranking shown to a user for a query using probabilities, we call it a logging policy π with the probability $\pi(\bar{R}|q)$ of showing a ranking $\bar{R}$ for a query Q. We denote a set of all rankings $\bar{R}$ with non-zero probability $\pi(\bar{R}|q)$ as $\pi(\cdot|q)$. We will use this probability in a top-k scenario where a user will see only a few documents with the lowest rank.

With this model of user behavior and historical interactions, we can calculate the expected value of a metric. We will see that the resulting value is weighted and thus biased.

We want to estimate the value of a metric on the whole interaction dataset as if we had the true relevance labels. The goal of Counterfactual evaluation is to find an estimator $\hat{\Delta}$ that unbiasedly estimates $\mathcal{L}$ using:

$$\hat{\mathcal{L}} := \frac{1}{|\mathcal{Q}|} \sum_{Q_i \in \mathcal{Q}} \hat{\Delta}(f_\theta, \mathcal{D}_i, c_i, \pi). \tag{2.15}$$

Note that the $\hat{\Delta}$ estimator accepts the click labels c_i for the query Q_i instead of the true relevance labels y for a set of documents $\mathcal{D}_i$. The estimator also accepts the logging policy π.

If we try to estimate $\mathcal{L}$ naively using a $\hat{\Delta}_{naive}$ estimator where true relevance labels are replaced by click labels [4]:

$$\hat{\Delta}_{naive}(f_\theta, \mathcal{D}, c, \pi) := \sum_{D \in \mathcal{D}} \lambda(rank(D|f_\theta, \mathcal{D})) \cdot c(D), \tag{2.16}$$

we end up with a biased estimate caused by the position and document selection. We calculate the expected value over the observation and logging policy random variables to see that we cannot use the click labels directly. We are interested in the expected value, since we want to know whether we get the same value on average on a large dataset as if we used the true relevance labels. The expected value of this estimator is as follows:

$$\begin{aligned}
&\mathbb{E}_{o,\bar{R}}\left[\hat{\Delta}_{naive}(f_\theta, \mathcal{D}, c, \pi)\right] \\
&\quad = \mathbb{E}_{o,\bar{R}}\left[\sum_{D \in \mathcal{D}} \lambda(rank(D|f_\theta, \mathcal{D})) \cdot c(D)\right] \\
&\quad \overset{1}{=} \mathbb{E}_{o,\bar{R}}\left[\sum_{D \in \mathcal{D}} o(D) \cdot \lambda(rank(D|f_\theta, \mathcal{D})) \cdot y(D)\right] \\
&\quad = \sum_{D \in \mathcal{D}} \mathbb{E}_{o,\bar{R}}\left[o(D) \cdot \lambda(rank(D|f_\theta, \mathcal{D})) \cdot y(D)\right] \\
&\quad = \sum_{D \in \mathcal{D}} \mathbb{E}_{\bar{R}}\left[P(o(D)=1|\bar{R}, D) \cdot \lambda(rank(D|f_\theta, \mathcal{D})) \cdot y(D)\right] \\
&\quad = \sum_{\bar{R} \in \pi(\cdot|q)} \pi(\bar{R}|q) \sum_{D \in \mathcal{D}} P(o(D)=1|\bar{R}, D) \cdot \lambda(rank(D|f_\theta, \mathcal{D})) \cdot y(D).
\end{aligned} \tag{2.17}$$

In Step 1 we used the Equation 2.11.In this equation, the document selection and position biases are clearly visible. The ranks are weighted by the probability of observance at the given position in a ranking, and each ranking is weighted according to the probability

of being shown by the logging policy π. We can see that, although the clicks are related to relevance, they cannot be used directly. What we really want is the expression shown in Equation 2.4.

To remove biases, an inverse propensity scoring estimator can be used. The idea is to estimate the probabilities that cause the bias and use the estimation to cancel out the effect. In the following estimator, we estimate only the position bias for each ranking. This estimator is called a policy-oblivious estimator [4]:

$$\hat{\Delta}_{oblivious}(f_\theta, \mathcal{D}, c, \pi) := \frac{\sum_{D \in \mathcal{D}} \lambda(rank(D|f_\theta, \mathcal{D})) \cdot c(D)}{P(o(D) = 1|\bar{R}, D)}. \tag{2.18}$$

We get an unbiased estimate when we calculate the same expected value of this estimator:

$$\begin{aligned}
&\mathbb{E}_{o,\bar{R}}\left[\hat{\Delta}_{oblivious}(f_\theta, \mathcal{D}, c, \pi)\right] \\
&\quad = \mathbb{E}_{o,\bar{R}}\left[\sum_{D \in \mathcal{D}} \frac{\lambda(rank(D|f_\theta, \mathcal{D})) \cdot c(D)}{P(o(D) = 1|\bar{R}, D)}\right] \\
&\quad = \mathbb{E}_{o,\bar{R}}\left[\sum_{D \in \mathcal{D}} \frac{o(D) \cdot \lambda(rank(D|f_\theta, \mathcal{D})) \cdot y(D)}{P(o(D) = 1|\bar{R}, D)}\right] \\
&\quad = \sum_{D \in \mathcal{D}} \mathbb{E}_{o,\bar{R}}\left[\frac{o(D) \cdot \lambda(rank(D|f_\theta, \mathcal{D})) \cdot y(D)}{P(o(D) = 1|\bar{R}, D)}\right] \\
&\quad = \sum_{D \in \mathcal{D}} \mathbb{E}_{\bar{R}}\left[\frac{P(o(D) = 1|\bar{R}, D) \cdot \lambda(rank(D|f_\theta, \mathcal{D})) \cdot y(D)}{P(o(D) = 1|\bar{R}, D)}\right] \\
&\quad \stackrel{1}{=} \sum_{D \in \mathcal{D}} \lambda(rank(D|f_\theta, \mathcal{D})) \cdot y(D) \\
&\quad = \Delta(f_\theta, \mathcal{D}, y).
\end{aligned} \tag{2.19}$$

Step 1 assumes $P(o(D) = 1|\bar{R}, D) > 0$. Only relevant documents contribute to the estimate since, in the sum, the non-relevant documents are 0. As long as the following condition is met, the estimate is unbiased:

$$\forall \bar{R} \in \pi(\cdot|q), \forall D \in \mathcal{D} : y(D) = 1 \implies P(o(D) = 1|\bar{R}, D) > 0. \tag{2.20}$$

For every relevant document, we need a non-zero probability of being examined in every ranking $\bar{R}$ that can be shown for a query Q. A convenient property of this estimator is that the logging policy π does not have to be known, which is why it is called policy-oblivious. To use this method, we must know each clicked document's propensities. Propensities in the real world are unknown, so we need to estimate them. The estimation method is presented in further sections.

This condition cannot be satisfied when we show the user only the top-k documents. There could be more than k relevant documents, and since we cannot show more than k, we end up with at least one document that is not visible, so the probability is 0. When there are fewer relevant documents than k for each query, the previously deployed ranker must be near optimal, which is unlikely. As a result, the $\hat{\Delta}_{oblivious}$ estimator is biased in the top-k scenario. We use a policy-aware estimator $\hat{\Delta}_{aware}$ estimator to account for this.

Across multiple top-k rankings, showing all the relevant documents is possible. Therefore, a non-static logging policy could show each relevant document in a ranking. As a result, the probability of examination across multiple rankings is not 0. The document may not be visible in some rankings, but it is not a problem as long as it is visible in at least one ranking. The idea is to relax the per-ranking condition and apply a per-query condition. The probability of examination over all rankings can be calculated as an expectation over the logging policy [4]:

$$\begin{aligned} P(o(D)=1|D,\pi) &= \mathbb{E}_{\bar{R}}\left[P(o(D)=1|\bar{R},D)\right] \\ &= \sum_{\bar{R}\in\pi(\cdot|q)} \pi(\bar{R}|q)\cdot P(o(D)=1|\bar{R},D). \end{aligned} \tag{2.21}$$

The policy-aware estimator is defined as:

$$\hat{\Delta}_{aware}(f_\theta,\mathcal{D},c,\pi) := \sum_{D\in\mathcal{D}} \frac{\lambda(rank(D|f_\theta,\mathcal{D}))\cdot c(D)}{P(o(D)=1|D,\pi)}. \tag{2.22}$$

The estimator $\hat{\Delta}_{aware}$ weighs clicks more heavily when a document appears in only a single ranking compared to another document that appears in all rankings. Thus, correct for situations where a document is absent in some rankings and cannot be clicked. The expected value of the estimator is [4]:

$$\begin{aligned} &\mathbb{E}_{o,\bar{R}}\left[\hat{\Delta}_{aware}(f_\theta,\mathcal{D},c,\pi)\right] \\ &\quad= \mathbb{E}_{o,\bar{R}}\left[\sum_{D\in\mathcal{D}} \frac{\lambda(rank(D|f_\theta,\mathcal{D}))\cdot c(D)}{P(o(D)=1|D,\pi)}\right] \\ &\quad= \mathbb{E}_{o,\bar{R}}\left[\sum_{D\in\mathcal{D}} \frac{o(D)\cdot\lambda(rank(D|f_\theta,\mathcal{D}))\cdot y(D)}{P(o(D)=1|D,\pi)}\right] \\ &\quad= \sum_{D\in\mathcal{D}} \mathbb{E}_{o,\bar{R}}\left[\frac{o(D)\cdot\lambda(rank(D|f_\theta,\mathcal{D}))\cdot y(D)}{P(o(D)=1|D,\pi)}\right] \\ &\quad= \sum_{D\in\mathcal{D}} \mathbb{E}_{o,\bar{R}}\left[\frac{o(D)\cdot\lambda(rank(D|f_\theta,\mathcal{D}))\cdot y(D)}{\sum_{\bar{R}'\in\pi(\cdot|q)}\pi(\bar{R}'|q)\cdot P(o(D)=1|\bar{R}',D)}\right] \\ &\quad= \sum_{D\in\mathcal{D}} \mathbb{E}_{\bar{R}}\left[\frac{P(o(D)=1|\bar{R},D)\cdot\lambda(rank(D|f_\theta,\mathcal{D}))\cdot y(D)}{\sum_{\bar{R}'\in\pi(\cdot|q)}\pi(\bar{R}'|q)\cdot P(o(D)=1|\bar{R}',D)}\right] \\ &\quad= \sum_{D\in\mathcal{D}} \frac{\sum_{\bar{R}\in\pi(\cdot|q)}\pi(\bar{R}|q)\cdot P(o(D)=1|\bar{R},D)\cdot\lambda(rank(D|f_\theta,\mathcal{D}))\cdot y(D)}{\sum_{\bar{R}'\in\pi(\cdot|q)}\pi(\bar{R}'|q)\cdot P(o(D)=1|\bar{R}',D)} \\ &\quad= \sum_{D\in\mathcal{D}} \lambda(rank(D|f_\theta,\mathcal{D}))\cdot y(D) \\ &\quad= \Delta(f_\theta,\mathcal{D},y). \end{aligned} \tag{2.23}$$

Similarly, the equation holds as long as the following condition is satisfied:

$$\forall D\in\mathcal{D}: y(D)=1 \implies \sum_{\bar{R}\in\pi(\cdot|q)} \pi(\bar{R}|q)\cdot P(o(D)=1|\bar{R},D) > 0. \tag{2.24}$$

Every relevant document has a non-zero probability of examination in at least one ranking $\bar{R}$ with a non-zero probability of being shown. Therefore, as long as this condition is satisfied, the $\hat{\Delta}_{aware}$ estimator is unbiased.

The $\hat{\Delta}_{aware}$ estimator is a generalization over the $\hat{\Delta}_{oblivious}$ estimator since Condition 2.20 implies Condition 2.24. But the other implication is not true. The policy-oblivious estimator could be seen as a special case of the policy-aware estimator where the logging policy is static, and every relevant document is shown in the ranking.

There can be a large number of logging policies that satisfy Condition 2.24, such as a policy that performs a random permutation of the rankings or a less invasive one that swaps a random document with a random document at the top-k. Condition 2.24 can also be satisfied directly from the models deployed when the clicks were collected. A disadvantage is that we have to know or estimate the logging policy π otherwise, we cannot use this method. The next disadvantage is that historical interactions were logged using a logging policy, so we cannot choose a different one. We are also unable to validate the conditions. We do not know which documents are relevant and if the documents were observed. We assume that only the clicked documents are relevant and that every clicked document had been observed before it was clicked.

An important point to note here is that we start with the click c on the document, then we use the estimators to get to where we use the y of the actual relevance of the document, so we claim that the document is really relevant.

So far, we have assumed that we have no click noise. This means that we assumed that the following condition is true:

$$c(D) = \mathbb{1}_{y(D) \wedge c(D)}. \tag{2.25}$$

for every document D. However, the inverse propensity scoring approaches still work as long as the following condition is true [8]:

$$y(D_i) > y(D_j) \Leftrightarrow P(c(D_i) = 1 | q, \bar{R}, D_i) > P(c(D_j) = 1 | q, \bar{R}, D_j). \tag{2.26}$$

It means that as soon as a document D_i is more relevant than another document D_j, the probability of clicking on D_i is greater than clicking on D_j. The other implication states that if clicking on the document, D_i is larger than clicking on D_j, the document D_i is more relevant than D_j. So, as long as this condition holds, the estimates are unbiased under click noise. This condition is reasonable to assume in our case.

To better understand the biases, we present the same example as in [4] that contrasts the two approaches. First, we show a situation where there is a relevant document that appears in all the rankings. Subsequently, we offer a situation where another document is beyond the top-k rankings. We will see that in the first situation, both estimates are unbiased. In the second situation, the policy-oblivious estimator cannot account for the ranking where the document is missing compared to the policy-oblivious that provides an unbiased estimate.

► **Example 2.7.** Suppose that we have a query Q and a logging policy π that shows only two rankings $\bar{R}_1$ and $\bar{R}_2$:

$$\pi(\bar{R}_1 | q) > 0 \wedge \pi(\bar{R}_2 | q) > 0 \wedge \pi(\bar{R}_1 | q) + \pi(\bar{R}_2 | q) = 1. \tag{2.27}$$

In the first situation, there is a relevant document D_A that appears in both rankings, so the examination probabilities are positive for each ranking:

$$P(o(D_A)|\bar{R}_1, D_A) > 0 \wedge P(o(D_A)|\bar{R}_2, D_A) > 0. \tag{2.28}$$

We can write an expected value for a generic estimator $\hat{\Delta}$ as:

$$\begin{aligned}
&\mathbb{E}_{o,\bar{R}}\left[\hat{\Delta}(f_\theta, \mathcal{D}, c, \pi)\right] \\
&= \mathbb{E}_{o,\bar{R}}\left[\sum_{D\in\mathcal{D}} \frac{\lambda(rank(D|f_\theta,\mathcal{D}))\cdot c(D)}{\rho(o(D)|D,\pi,\bar{R})}\right] = \lambda(rank(D_A|f_\theta,\mathcal{D}))\cdot y(D_A)\cdot \\
&\left(\frac{\pi(\bar{R}_1|q)\cdot P(o(D_A)=1|\bar{R}_1,D_A)}{\rho(o(D_A)|D_A,\pi,\bar{R}_1)} + \frac{\pi(\bar{R}_2|q)\cdot P(o(D_A)=1|\bar{R}_2,D_A)}{\rho(o(D_A)|D_A,\pi,\bar{R}_2)}\right).
\end{aligned} \tag{2.29}$$

The ρ could be replaced by the policy-aware 2.22 or policy-oblivious 2.19 estimator. For a policy-oblivious estimator, the propensities are positive, so we can write:

$$\begin{aligned}
&\mathbb{E}_{o,\bar{R}}\left[\hat{\Delta}_{oblivious}(f_\theta, \mathcal{D}, c, \pi)\right] = \lambda(rank(D_A|f_\theta,\mathcal{D}))\cdot y(D_A)\cdot \\
&\quad\left(\frac{\pi(\bar{R}_1|q)\cdot P(o(D_A)=1|\bar{R}_1,D_A)}{P(o(D_A)=1|\bar{R}_1,D_A)} + \frac{\pi(\bar{R}_2|q)\cdot P(o(D_A)=1|\bar{R}_2,D_A)}{P(o(D_A)=1|\bar{R}_2,D_A)}\right) \\
&= \left(\pi(\bar{R}_1|q) + \pi(\bar{R}_2|q)\right)\cdot\lambda(rank(D_A|f_\theta,\mathcal{D}))\cdot y(D_A) \\
&=\lambda(rank(D_A|f_\theta,\mathcal{D}))\cdot y(D_A) = \Delta(f_\theta,\mathcal{D},y).
\end{aligned} \tag{2.30}$$

and, for a policy-aware estimator, the propensities are also positive, so we can write:

$$\begin{aligned}
&\mathbb{E}_{o,\bar{R}}\left[\hat{\Delta}_{aware}(f_\theta, \mathcal{D}_i, c_i, \pi)\right] = \lambda(rank(D_A|f_\theta,\mathcal{D}))\cdot y(D_A)\cdot \\
&\quad\left(\frac{\pi(\bar{R}_1|q)\cdot P(o(D_A)=1|\bar{R}_1,D_A) + \pi(\bar{R}_2|q)\cdot P(o(D_A)=1|\bar{R}_2,D_A)}{\pi(\bar{R}_1|q)\cdot P(o(D_A)=1|\bar{R}_1,D_A) + \pi(\bar{R}_2|q)\cdot P(o(D_A)=1|\bar{R}_2,D_A)}\right) \\
&=\lambda(rank(D_A|f_\theta,\mathcal{D}))\cdot y(D_A) = \Delta(f_\theta,\mathcal{D},y).
\end{aligned} \tag{2.31}$$

Note that the policy-oblivious estimator applies the propensity per single ranking, and the policy-aware estimator corrects with propensity per query so the propensities are the same for both rankings. Both estimates are unbiased for this situation since both Condition 2.20 and Condition 2.24 are satisfied.

Now, the second situation in which we have a different document D_B that is present in the ranking R_1 and is not present in the ranking R_2 is the following:

$$P(o(D_B)|\bar{R}_1, D_B) > 0 \wedge P(o(D_B)|\bar{R}_2, D_B) = 0. \tag{2.32}$$

As a result, no click will ever be received in the ranking R_2, so we have to consider only R_1 in the estimation. The generic expected value is:

$$\begin{aligned}
&\mathbb{E}_{o,\bar{R}}\left[\hat{\Delta}(f_\theta, \mathcal{D}, c, \pi)\right] = \mathbb{E}_{o,\bar{R}}\left[\sum_{D\in\mathcal{D}} \frac{\lambda(rank(D|f_\theta,\mathcal{D}))\cdot c(D)}{\rho(o(D)|D,\pi,\bar{R})}\right] \\
&\quad=\lambda(rank(D_A|f_\theta,\mathcal{D}))\cdot y(D_A)\cdot\left(\frac{\pi(\bar{R}_1|q)\cdot P(o(D_A)=1|\bar{R}_1,D_A)}{\rho(o(D_A)|D_A,\pi,\bar{R}_1)}\right).
\end{aligned} \tag{2.33}$$

Condition 2.20 is not satisfied, so the policy-oblivious estimate provides a biased estimate:

$$\begin{aligned}
&\mathbb{E}_{o,\bar{R}}\left[\hat{\Delta}_{oblivious}(f_\theta, \mathcal{D}, c, \pi)\right] \\
&=\lambda(rank(D_A|f_\theta, \mathcal{D})) \cdot y(D_A) \cdot \frac{\pi(\bar{R}_1|q) \cdot P(o(D_A) = 1|\bar{R}_1, D_A)}{P(o(D_A) = 1|\bar{R}_1, D_A)} \\
&=\pi(\bar{R}_1|q) \cdot \lambda(rank(D_A|f_\theta, \mathcal{D})) \cdot y(D_A).
\end{aligned} \tag{2.34}$$

On the other hand, Condition 2.24 for the policy-aware estimator is fulfilled. It implies that Equation 2.23 is true and the estimate is unbiased, so this estimator can account for the situation where a document is not shown in a ranking as soon as the document is shown in some other rankings:

$$\begin{aligned}
&\mathbb{E}_{o,\bar{R}}\left[\hat{\Delta}_{aware}(f_\theta, \mathcal{D}, c, \pi)\right] \\
&= \lambda(rank(D_A|f_\theta, \mathcal{D})) \cdot y(D_A) \cdot \frac{\pi(\bar{R}_1|q) \cdot P(o(D_A) = 1|\bar{R}_1, D_A)}{\pi(\bar{R}_1|q) \cdot P(o(D_A) = 1|\bar{R}_1, D_A)} \\
&= \lambda(rank(D_A|f_\theta, \mathcal{D})) \cdot y(D_A) = \Delta(f_\theta, \mathcal{D}, y).
\end{aligned} \tag{2.35}$$

The policy-aware estimator gives more weight to cases where the document is rarely shown.

The intuition behind the estimators can be the following. The estimator tries to convert the previous ranking to the following case. The case is when a ranking of all matched documents is shown to the user. The user is a robot that clicks on a document if and only if a document is relevant to the user based on a given query. So, the user goes through the ranking and examines all the documents. Even when there is a large number of those documents. This behavior would provide an unbiased estimate in our model. Unfortunately, this is not the case in the real world. The estimators try to remove the position bias and document selection bias. Estimators try to convert the biased data to the ideal situation by weighting the clicks.

Since we do not know the propensities, we have to estimate them. In Section 2.2.4, we show how to estimate position bias, and in Section 2.2.5 we generalize this method and try to estimate document selection bias.

2.2.3.3 Learning a model

So far, we have shown how to evaluate model performance using estimators to obtain an unbiased estimate of a metric using historical user interactions. We can use an estimator $\hat{\Delta}$ to directly learn a model. We will use this approach to learn a linear model that we will then compare with the *LambdaMART* approach. Although we can use the click labels to learn a model as in $\hat{\Delta}_{naive}$ estimator we can also use the unbiased estimators. If we use the naive approach, we learn a model that optimizes a biased metric, so the performance can be worse than when using the other estimators. The model that learns using bare clicks is called a biased model [9]. We want to find the parameters θ for a model f to obtain an optimal value of a metric [3]:

$$\theta = \arg\min_{\theta'} \frac{1}{|\mathcal{Q}|} \sum_{Q_i \in \mathcal{Q}} \hat{\Delta}(f_{\theta'}, \mathcal{D}_i, c_i, \pi). \tag{2.36}$$

We would use gradient descent, but the function is not differentiable due to the *rank* function. We will solve this problem by optimizing a differentiable upper bound $\overline{rank}$ of the *rank* function. If we can find a tight upper bound for the *rank* function, then optimizing that upper bound can also help us find the optimal value of the function. In the model, we will optimize the following upper bound:

$$\begin{aligned} rank(D|f_\theta, D) = \sum_{D' \in \mathcal{D}} \mathbb{1}_{f_\theta(D) \leq f_\theta(D')} \leq \sum_{D' \in \mathcal{D}} max(1 - (f_\theta(D) - f_\theta(D')), 0) \\ = \overline{rank}(D|f_\theta, \mathcal{D}). \end{aligned} \tag{2.37}$$

On the right side of the inequality, there are three cases in the *max* function. The score of the document D' is higher, so the difference in parentheses is negative, and thus the first argument is greater than 1. For each document D' that should be ranked before D, we add a number greater than 1. The second case is when the document D' has a lower score than D. In this case, the difference in parentheses is positive. Thus, we add a number between 0 and 1 due to the *max* function. The last case is when the scores of the document are the same. We do not know how to order the documents with the same score, so we place them all in the last position, so we add 1. We can see that the inequality holds for all three cases. The $\overline{rank}$ function is differentiable. So, the function we will optimize looks like this:

$$\frac{1}{|\mathcal{Q}|} \sum_{Q_i \in \mathcal{Q}} \hat{\Delta}(f_\theta, \mathcal{D}_i, c_i, \pi) = \frac{1}{|\mathcal{Q}|} \sum_{Q_i \in \mathcal{Q}} \sum_{D \in \mathcal{D}_i} \frac{\lambda(\overline{rank}(D|f_\theta, \mathcal{D}_i)) \cdot c(D)}{\rho(o(D)|D, \pi, \bar{R})}. \tag{2.38}$$

This is an example of a listwise loss function. Note that we pass the document D to the model as an argument, but when we learn the model, we do not use the concrete document, but we use the document features that are presented in further sections. The pseudocode for learning a linear model is the following:

Algorithm 1: *Learning the baseline model*

Input : Model of position bias B,
a training set of queries $\mathcal{Q}$,
number of iterations n,
learning rate η

Output: Parameters of the model θ

Initialize parameters θ' to 0.

for $i \leftarrow 1$ **to** $n + 1$ **do**
  for $Q_i \in \mathcal{Q}$ **do**
    Compute the propensity $\rho(o(D)|D, \pi, \bar{R})$ using B for each $D \in \mathcal{D}_i$
    Calculate the gradient: $\Delta_\theta \left[\sum_{D \in \mathcal{D}_i} \frac{\lambda(\overline{rank}(D|f_\theta, \mathcal{D}_i)) \cdot c(D)}{\rho(o(D)|D, \pi, \bar{R}_i)} \right]$
    Update the parameters θ' by subtracting the calculated gradient and η
  end
end
return θ';

In the algorithm, we iterate n times throughout the dataset, and for each query, we compute the propensity. The function ρ can be replaced according to the naive, policy-oblivious, or policy-aware estimators. Once we have the propensities, we use

them to calculate the gradient of the loss function with respect to the parameters θ. Then we subtract the gradient using a learning rate parameter η. Subtracting is valid only for the *DCG* and *ARP* metrics. The gradient update is performed for each query. Parameters n and η are chosen using validation data.

2.2.4 Position bias estimation

As described in the previous sections, we address two types of bias in the thesis. The biases are unknown, so to get an unbiased estimate, we need to know the propensities.

One of the biases is the position bias. We will show the estimation in this section. The estimation of selection bias is a generalization and will be shown in Section 2.2.5.

The simple idea of calculating the position bias as the number of clicks in a position and dividing it by the total document recommendations in the position is wrong. In our model, the probability of a click is:

$$P(c(D) = 1|q, \bar{R}, D) = P(o(D) = 1|\bar{R}, D) \cdot P(y(D)|q, D). \tag{2.39}$$

The rankings that were shown to the users were sorted by estimated relevance. So, the most relevant documents were at the top. If we do the naive calculation, then the propensities would be biased by relevance. One way to eliminate bias would be to perform a random permutation of the documents in the rankings before showing the ranking to a user. Then there would be the same number of relevant documents at each position on average, so the naive estimate would not be biased for large data. Unfortunately, we cannot change the logging policy of the historical data. And even if we can, the randomization of the result causes a bad user experience.

We will simplify our problem by assuming that the observation is dependent only on the position. A ranking and a document do not influence the probability of observation, except for the position of the document in the ranking:

$$P(o(D) = 1|\bar{R}, D, k) = P(o(D) = 1|k). \tag{2.40}$$

The position bias estimation algorithm we present is based on [6]. It is an Expectation-Maximization (*EM*) algorithm that iterates between an expectation step and a maximization step. The procedure finds parameters of the user click model that maximize the log-likelihood of the data. For the sake of brevity, we denote probabilities as:

$$P(c(D) = 1|q, D, k) = \underbrace{P(o(D) = 1|k)}_{\theta_k} \cdot \underbrace{P(y(D)|q, D)}_{\gamma_{q,D}}. \tag{2.41}$$

Parameters θ_k and $\gamma_{q,D}$ are the partially observed parameters of the model that we want to estimate. There is one θ_k for each position k and one $\gamma_{q,D}$ for each unique query string q and document D pair. Let $\{\theta_k\}$ be the set of parameters for all positions and let $\{\gamma_{q,D}\}$ be the set of all parameters for all query–document pairs. The log-likelihood of the data in the model is:

$$\sum_{Q_i \in \mathcal{Q}} \sum_{D_j \in \mathcal{D}_i} c_i(D_j) \cdot \log \theta_k \gamma_{q,D_j} + (1 - c_i(D_j)) \cdot \log(1 - \theta_k \gamma_{q,D_j}). \tag{2.42}$$

We first present the standard *EM* algorithm and then a regression-based version that better handles sparse data. We can compute the following probabilities using our parameters.

$$
\begin{aligned}
&P(o(D)=1, y(D)=1|c(D)=1, q, D, k)=1, \\
&P(o(D)=1, y(D)=1|c(D)=0, q, D, k)=0, \\
&P(o(D)=1, y(D)=0|c(D)=0, q, D, k)=\frac{\theta_k(1-\gamma_{q,D})}{1-\theta_k\gamma_{q,D}}, \\
&P(o(D)=0, y(D)=1|c(D)=0, q, D, k)=\frac{(1-\theta_k)\gamma_{q,D}}{1-\theta_k\gamma_{q,D}}, \\
&P(o(D)=0, y(D)=0|c(D)=0, q, D, k)=\frac{(1-\theta_k)(1-\gamma_{q,D})}{1-\theta_k\gamma_{q,D}}.
\end{aligned} \tag{2.43}
$$

From these probabilities, we can compute the marginals:

$$
\begin{aligned}
&P(o(D)=1|c(D)=0, q, D, k)=\frac{\theta_k(1-\gamma_{q,D})}{1-\theta_k\gamma_{q,D}}, \\
&P(y(D)=1|c(D)=0, q, D, k)=\frac{(1-\theta_k)\gamma_{q,D}}{1-\theta_k\gamma_{q,D}}, \\
&P(o(D)=1|c(D)=1, q, D, k)=1, \\
&P(y(D)=1|c(D)=1, q, D, k)=1.
\end{aligned} \tag{2.44}
$$

In the expectation step in iteration $t+1$ we use the parameters $\{\theta_k^{(t)}\}$ and $\{\gamma_{q,D}^{(t)}\}$ from iteration t and estimate the joint distribution of the variables $o(D)$ and $y(D)$ using equations 2.43. Subsequently, we compute the marginal distribution using equations 2.44.

In the maximization step in iteration $t+1$, we use the marginals 2.44 from iteration t to obtain a better estimate of the hidden variables $\{\theta_k^{(t+1)}\}$ and $\{\gamma_{q,D}^{(t+1)}\}$ using the data as:

$$
\theta_k^{(t+1)} := \frac{\sum_{Q_i\in\mathcal{Q}}\sum_{D_j\in\mathcal{D}_i} P(o(D_j)=1|c(D_j), q, D_j, k)\cdot \mathbb{1}_{\bar{R}_i[k]=D_j}}{\sum_{Q_i\in\mathcal{Q}}\sum_{D_j\in\mathcal{D}_i}\mathbb{1}_{\bar{R}_i[k]=D_j}}, \tag{2.45}
$$

$$
\gamma_{q,D}^{(t+1)} := \frac{\sum_{Q_i\in\mathcal{Q}}\sum_{D_j\in\mathcal{D}_i} P(y(D_j)=1|c(D_j), q, D_j, k)\cdot \mathbb{1}_{q_i=q\wedge D_j=D}}{\sum_{Q_i\in\mathcal{Q}}\sum_{D_j\in\mathcal{D}_i}\mathbb{1}_{q_i=q\wedge D_j=D}}, \tag{2.46}
$$

where the indicator function $\mathbb{1}_{\bar{R}_i[k]=D_j}$ is 1 if and only if the document D_j is in a position k in a ranking $\bar{R}_i$ logged for a query Q_i. Otherwise, the function is 0. Equation 2.45 divides the data by positions. On the other hand, Equation 2.46 divides the data into query–document pairs. Note that the maximization step uses the marginal probabilities from equations 2.44.

We start with an initial estimate of the parameters that could be, for example, random. At each step, the parameters are corrected using the data. In a subsequent iteration, the parameters are used to obtain an even better estimate. Therefore, the log-likelihood converges to a local optimum. A complete explanation of why the *EM* algorithm optimizes the likelihood can be found in Paper [10]. We stop iterating once the convergence condition is satisfied.

Now, we present the regression-based *EM* algorithm [10]. The motivation to not use the standard algorithm is the sparsity of the data. We need the parameter $\gamma_{q,D}$ for each query–document pair we want to rank. There is a problem with new queries and new documents. There could be a lack of data even for the query–document pairs we have in the training data to get a good estimate. The new approach differs in the maximization step, which uses a feature vector $\boldsymbol{x}$ that represents the query–document pair. The feature vector $\boldsymbol{x}$ used can be the same as the ranking features used in the ranking system. Using a feature vector instead of a specific query–document pair allows the algorithm to generalize beyond single query document pair and thus handles the data sparsity. A function g is then used to represent the relevance $\gamma_{q,D} = g(\boldsymbol{x}_{q,D})$. To obtain the function g to estimate relevance, the problem is converted to a classification problem. After the expectation step, a label r is sampled from $P(y(D)|c(D), q, D, k)$ for each query–document pair q and D. The label represents relevance. We get a set of feature vectors $\boldsymbol{x}_{q,D}$ and a binary label r for each vector. Then we learn a binary classifier that is capable of predicting $P(r = 1|\boldsymbol{x}_{q,D}) = g(\boldsymbol{x}_{q,D})$ [6]. Here is a pseudocode of the regression-based *EM* algorithm.

Algorithm 2: *Regression-based EM*

Input : Set of documents $\mathcal{D}_i$ and set of rankings $\bar{R}_i$ for queries $Q_i \in \mathcal{Q}$
Output: θ_k, $\gamma_{q,D}$
$\theta_k^{(0)} \leftarrow 0$ // or random
$\gamma_{q,D}^{(0)} \leftarrow 0$ // or random
$t \leftarrow 0$
repeat
  $S \leftarrow \{\}$
  for $Q_i \in \mathcal{Q}$ **do**
    for $D_j \in \mathcal{D}_i$ **do**
      Sample r from $P(y(D)|c_i(D_j), q_i, D_j, pos(D_j, \bar{R}_i))$
      $S \leftarrow S \cup (r, \boldsymbol{x}_{q_i,D_j})$
    end
  end
  Fit classifier g with S
  $\gamma_{q,D}^{(t+1)} \leftarrow g(\boldsymbol{x}_{q,D})$
  Create parameters $\theta_k^{(t+1)}$ based on Definition 2.45
  $t \leftarrow t + 1$
until *converged*;
return set of $\theta_k^{(t)}$ and set of $\gamma_{q,D}^{(t)}$

Expression $pos(D_j, \bar{R}_i)$ returns the position of D_j in the ranking $\bar{R}_i$. When the document is not present in the ranking, the probability is defined as 0. Regression-based *EM* replaces just the Equation 2.46 by sampling a distribution and then learning a model. The convergence criterion could be arbitrary. For example, a fixed number of iterations or a minimal difference in log-likelihood between iterations. In the next chapter, we present the exact implementation of the algorithm.

2.2.5 Selection bias estimation

In this section, we estimate the document selection bias in a way similar to that of Section 2.2.4. We found no document selection bias estimation method in the literature, so we created our own using the generalization of position bias estimation. In Paper [3] they present only the document selection bias, but without any estimation method. They create semi-synthetic data where the bias is predetermined and thus is known. This is not our case.

The goal is to estimate the value of:

$$\sum_{\bar{R} \in \pi(\cdot|q)} \pi(\bar{R}|q) \cdot P(o(D) = 1|\bar{R}, D), \tag{2.47}$$

for each query and for each document from the candidate set. The expression sums over all rankings with a non-zero probability of appearance for a given query. The probability of the appearance of a ranking $\bar{R}$ is multiplied by the probability of the observance of document D in the ranking $\bar{R}$. A document may have a 0 probability of observance in a concrete ranking since the document is not visible. Similarly, as in Expression 2.21 we condition the click probability on the logging policy π instead of the concrete ranking and get:

$$\begin{aligned} P(c(D) = 1|q, D, \pi) &= \mathbb{E}_{\bar{R}} \left[P(c(D) = 1|q, \bar{R}, D) \right] \\ &= \mathbb{E}_{\bar{R}} \left[P(o(D) = 1|\bar{R}, D) \cdot P(y(D)|q, D) \right] \\ &= \underbrace{P(y(D)|q, D)}_{\gamma_{q,D}} \overbrace{\sum_{\bar{R} \in \pi(\cdot|q)} \pi(\bar{R}|q) \cdot \underbrace{P(o(D) = 1|\bar{R}, D)}_{\theta_k}}^{\theta_{q,D}}, \end{aligned} \tag{2.48}$$

the concrete rankings $\bar{R}$ are replaced by the logging policy. The probability of clicking now depends only on the given query and document. The click probability also depends on the logging policy, but the logging policy is fixed. The data are already collected using a logging policy that we cannot change, so we do not use it in the notation. The expression marked $\theta_{q,D}$ is the observation probability of a document for a query without the dependence on a ranking. The probability of clicking in Equation 2.48 can be seen as the probability of clicking on a document D for a query string q over all possible rankings. There is one parameter $\theta_{q,D}$ per each query–document pair. Note that the parameter $\theta_{q,D}$ is the same as Expression 2.47.

The log-likelihood of the model is similar to the Expression 2.42 but the θ_k is replaced by $\theta_{q,D}$:

$$\sum_{Q_i \in \mathcal{Q}} \sum_{D_j \in \mathcal{D}_i} c_i(D_j) \cdot \log \theta_{q,D_j} \cdot \gamma_{q,D_j} + (1 - c_i(D_j)) \cdot \log(1 - \theta_{q,D_j} \gamma_{q,D_j}). \tag{2.49}$$

Similarly, as for equations in 2.43, we can calculate the following joint probabilities using

the parameters:

$$\begin{aligned}
&P(o(D)=1, y(D)=1|c(D)=1, q, D, \pi)=1, \\
&P(o(D)=1, y(D)=1|c(D)=0, q, D, \pi)=0, \\
&P(o(D)=1, y(D)=0|c(D)=0, q, D, \pi)=\frac{\theta_{q,D}(1-\gamma_{q,D})}{1-\theta_{q,D}\gamma_{q,D}}, \\
&P(o(D)=0, y(D)=1|c(D)=0, q, D, \pi)=\frac{(1-\theta_{q,D})\gamma_{q,D}}{1-\theta_{q,D}\gamma_{q,D}}, \\
&P(o(D)=0, y(D)=0|c(D)=0, q, D, \pi)=\frac{(1-\theta_{q,D})(1-\gamma_{q,D})}{1-\theta_{q,D}\gamma_{q,D}}.
\end{aligned} \tag{2.50}$$

From these probabilities, we can compute the marginals:

$$\begin{aligned}
&P(o(D)=1|c(D)=0, q, D, \pi)=\frac{\theta_{q,D}(1-\gamma_{q,D})}{1-\theta_{q,D}\gamma_{q,D}}, \\
&P(y(D)=1|c(D)=0, q, D, \pi)=\frac{(1-\theta_{q,D})\gamma_{q,D}}{1-\theta_{q,D}\gamma_{q,D}}, \\
&P(o(D)=1|c(D)=1, q, D, \pi)=1, \\
&P(y(D)=1|c(D)=1, q, D, \pi)=1.
\end{aligned} \tag{2.51}$$

In the maximization step, we use the marginal probabilities of the previous iteration to calculate the parameters $\{\gamma_{q,D}\}$ and $\{\theta_{q,D}\}$. The formula for $\gamma_{q,D}^{(t+1)}$ is the same as in Definition 2.46. To estimate $\theta_{q,D}^{(t+1)}$ for a query string q and a document D, we use the following formula:

$$\theta_{q,D}^{(t+1)}=\frac{\sum_{Q_i\in\mathcal{Q}}\sum_{D_j\in\mathcal{D}_i} P(o(D_j)=1|c(D_j), q, D_j, pos(D_j, \bar{R}_i))\cdot \mathbb{1}_{q_i=q\wedge D_j=D}}{\sum_{Q_i\in\mathcal{Q}}\sum_{D_j\in\mathcal{D}_i}\mathbb{1}_{q_i=q\wedge D_j=D}}, \tag{2.52}$$

where the probability $P(o(D_j)=1|c(D_j), q, D_j, pos(D_j, \bar{R}_i))$ is 0 when the document D_j is not present in the ranking $\bar{R}_i$. The probability is not estimated by this model, so we use the estimate in Section 2.2.4. We iterate over all query strings, and for each query where the document D is in the candidate set, we sum the probability that the document D was observed in the ranking. This sum is divided by the total number of queries in which the document D is in the set of candidates. According to the way an *EM* algorithm works, the parameter $\theta_{q,D}^{(t)}$ should be increasingly closer to the true value of $\theta_{q,D}$.

The *EM* algorithm is mostly the same as in Section 2.2.4. We will use the regression-based version again for the same reason: data sparseness. Similarly, as for Equation 2.46, we do not calculate $\theta_{q,D}$ for each pair, but use a feature vector $\boldsymbol{x}_{q,D}$ to train a model that outputs probabilities. To obtain the labels for the vectors, we sample binary labels from probability $P(o(D)=1|c(D), q, D, pos(Dj, \bar{R}))$ for each document D in a ranking $\bar{R}$. A label equal to 1 represents that the document was observed. To estimate the document selection bias, we need the position bias estimate. In the algorithm, we train two binary classifiers instead of one in Algorithm 2.

Note that these models also estimate the relevance of documents. Relevance can be used for ranking: documents from the candidate set can be sorted by relevance in

descending order. This is an example of the pointwise approach, where the probability of relevance is trained simply by using document features and labels without dependence on other documents. These models will also be evaluated and compared with the other approaches.

2.2.6 Item/User K-Nearest-Neighbors (*KNN*)

In this section, we use the term *item* as a synonym for *document* as it is a common naming convention on this topic. The *Item/User KNN* algorithms are very popular and effective models in recommendation systems. We will use the algorithm to create personalized features for each item in the candidate set assigned to a query that a user sends. In the next chapter, we show how. This section describes only the algorithms [11].

The *UserKNN* and *ItemKNN* algorithms belong to the category of collaborative filtering algorithms. Collaborative filtering is based on the assumption that people tend to like things that people with similar tastes and preferences also like. There are two types, user-based and item-based. The basic idea of the user-based is that groups of users are interested in the same content. Once we identify which groups a user belongs to, we can recommend the items with which the users in the group interacted. The item-based approach finds items similar to those with which the user has interacted in the past and recommends them. The advantage of collaborative filtering is that there is no need for user or item attributes. The only data needed are the user preferences on the items. Preferences are usually collected using user feedback. Feedback could be explicit or implicit. Explicit feedback could be collected, for example, from users using a form in which users can rate an item from 0 to 5 stars. The implicit feedback was already presented. Since we do not have explicit feedback, we will use implicit feedback in the form of clicks. Collaborative filtering differs from the content-based approach, where the item or user attributes are used to get the recommendation. Content-based methods will not be discussed.

Both algorithms work with a matrix called *rating matrix* or *interaction matrix*. We denote the matrix as

$$\mathbf{R} \in \mathbb{R}_?^{|U| \cdot |I|}, \tag{2.53}$$

where $|U|$ is the size of the set of users U and $|I|$ is the size of the set of items I. The $\mathbb{R}_?$ is the set of real numbers with the symbol ?:

$$\mathbb{R}_? := \mathbb{R} \cup ?. \tag{2.54}$$

The question mark represents an unknown or unobserved value. We will use the users $u \in U$ and the items $i \in I$ to index the matrix. Let $r_{i,u} \in \mathbb{R}_?^{|U| \cdot |I|}$ represent an element in the matrix $\mathbf{R}$, and let $r_{u:}$ and $r_{:i}$ represent a vector of user u and item i, respectively. Each row of the matrix represents a user and contains the user's interaction with the items. Each matrix column represents an item and contains users' interactions with the item. The matrix is huge when there are a large number of items and users, but the matrix is very sparse, which means that most of the cells are ? or 0. A rating matrix $\mathbf{R}$ is shown in Figure 2.5. We can see that in the rows are the vectors $r_{u:}$ representing users, and in the columns are the $r_{:i}$ vectors representing items.

From the feedback, we want to create the matrix. We can generally have multiple interactions for each user-item pair and possibly of different types. Each cell $r_{i,j}$ is just

one real number or symbol ?. We need to aggregate the interaction into a single number. The problem is solved using an aggregation function that maps the set of all possible interactions to $\mathbb{R}_?$. We will present the aggregation function used in the next chapter. An example of this function could be a function that gives each type of interaction a weight, and then for all the interactions in the set, the weights are summed according to the interaction type. When the set is empty, the value is ?.

Before using the matrix, the matrix cells containing ? must be replaced by a real number. Therefore, we need a function m:

$$m : \mathbb{R}_?^{|U|\cdot|I|} \rightarrow \mathbb{R}^{|U|\cdot|I|}. \tag{2.55}$$

The function m could be a rating prediction model. Rating prediction is a standard task in recommendation systems, but is beyond the scope of this thesis. From now on, we will use the following simple function m that applies to each cell:

$$m(r_{i,j}) := \begin{cases} 0 & \text{if } r_{i,j} =?, \\ r_{i,j} & \text{otherwise.} \end{cases} \tag{2.56}$$

All symbols ? are replaced by 0. The other cells are left unchanged. Now, we present pseudocodes of the algorithms.

Algorithm 3: *UserKNN*

Input : Rating matrix $\mathbf{R} \in \mathbb{R}_?^{|U|\cdot|I|}$,
a user u,
how many neighbors to consider k
Output: Vector representing items with relevancy scores
$Res[1, \ldots, |I|] \leftarrow 0$
Find set of k most similar users N for user u using similarity sim
for $u_n \in N$ **do**
| $Res \leftarrow Res + sim(u, u_n) \cdot r_{u_n:}$
end
return Res

We find the k most similar users N to the user u. Then for each similar user u_n, we aggregate the rating vector $r_{u_n:}$ of the matrix $\mathbf{R}$ to Res weighted by similarity sim between user u and user u_n. The output is a vector of length $|I|$, and each vector element represents an item. The value of the vector element represents a score for how the item is relevant to the user u.

Algorithm 4: *ItemKNN*

Input : Rating matrix $\mathbf{R} \in \mathbb{R}_?^{|U| \cdot |I|}$,
a user u,
how many neighbors to consider k
Output: Vector representing items with relevancy scores
$Res[1, \dots, |I|] \leftarrow 0$
Find set of all interacted items $\bar{I}$ for user u
for $\bar{i} \in \bar{I}$ **do**
  Find set of k most similar items N for item $\bar{i}$ using similarity *sim*
  for $i_n \in N$ **do**
    $Res[i_n] \leftarrow Res[i_n] + r_{u,\bar{i}} \cdot sim(i_n, \bar{i})$
  end
end
return *Res*

For each interacted item $\bar{i}$, we find the k most similar items N using similarity *sim*. We sum the ratings $r_{u,\bar{i}}$ weighted by *sim* over all similar items i_n. The algorithms are very similar, we just use rows or columns from the rating matrix.

The similarity function *sim* returns a real number that represents the similarity of two vectors. The similarity function uses the rating matrix. In this thesis, we use the cosine similarity defined as follows:

$$cosine(u, v) := \frac{u \cdot v}{||u|| \cdot ||v||}, \tag{2.57}$$

where u and v are some non-zero vectors of real numbers of the same length. The expression $u \cdot v$ is a standard scalar product:

$$u \cdot v := \sum_i u_i \cdot v_i, \tag{2.58}$$

and the expression $|| \cdot ||$ is the Euclidean norm:

$$||u|| := \sqrt{\sum_i u_i^2}. \tag{2.59}$$

So, the similarity between users $u_1, u_2 \in U$ is defined as:

$$sim(u_1, u_2) := cosine(R_{u_1:}, R_{u_2:}), \tag{2.60}$$

and similarly for items $i_1, i_2 \in I$:

$$sim(i_1, i_2) := cosine(R_{:i_1}, R_{:i_2}). \tag{2.61}$$

The cosine similarity is bounded by -1 and $+1$, where higher means more similar. Cosine similarity is suitable for sparse vectors since only non-zero vector elements can change the result value.

Calculating the k nearest neighbors could be very computationally demanding, so the neighbors for each element are usually precomputed.

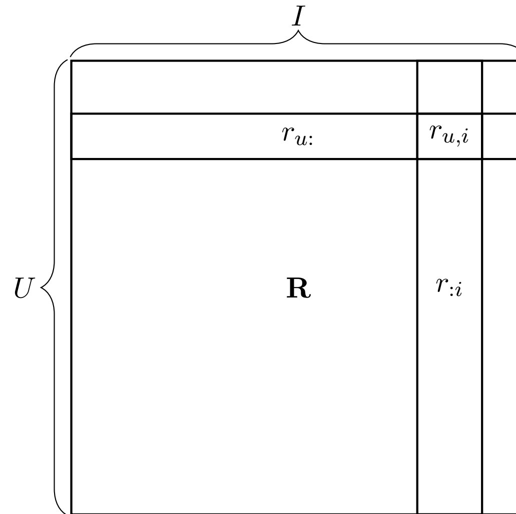

Figure 2.5 A rating matrix $\mathbf{R}$ denoting the important elements

2.2.7 *RankNet, LambdaRank, LambdaMART*

Now we present the basic ideas behind the state-of-the-art pairwise model in Learning to Rank. We show how to train an unbiased pairwise model using only interaction data. We will use only binary relevance labels, although the idea holds with general labeled data where the labels can be, for example, a real number between 0 and 1 [12].

The first approach was *RankNet*, and other approaches are based on this model. Despite the name *RankNet*, the underlying model can be any differentiable model, but the original paper used neural networks [13]. Let relation $\rhd_Q$ denote that document D_i is more relevant (and thus should be ranked before) D_j for a query Q:

$$D_i \rhd_Q D_j, \tag{2.62}$$

when we are writing about a single query, we use just $\rhd$ without the subscript. From now on, we will use only one query, so the Q will be omitted. A document D_i should be ranked before another D_j because the labels differ. Document D_i could have 5 stars relevance label, while document D_j has only 3. Let $P(D_i \rhd D_j)$ denote the probability that $D_i \rhd D_j$ for query Q. For now, suppose that we have a trained model f that accepts a vector $\boldsymbol{x}$ of document features and query-related features of size m and outputs a real number:

$$f : \mathbb{R}^m \to \mathbb{R}. \tag{2.63}$$

Let $s_i = f(\boldsymbol{x}_i)$ and $s_j = f(\boldsymbol{x}_j)$, where $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are the features of the document D_i and D_j, respectively. Note that we do not use the exact document identifier, but we use the document features. With this model trained, we create the ranking R of a set of documents $\mathcal{D}$ by sorting the documents by the model output in descending order.

Now, we show how to train the model. We estimate the probability $P(D_i \rhd D_j)$ that D_i should be ranked before D_j for query Q using:

$$\hat{P}_{i,j} \coloneqq \frac{1}{1 + e^{-\sigma(s_i - s_j)}}. \tag{2.64}$$

The parameter σ determines the shape of the sigmoid function. We do not have the relevance labels in our data, but we have the clicks. For now, assume that click means relevance and non-click means irrelevance. We will deal with the gap later. Let

$$S_{i,j} := \begin{cases} 1 & \text{if } c(D_i) > c(D_j), \\ 0 & \text{if } c(D_i) = c(D_j), \\ -1 & \text{if } c(D_i) < c(D_j). \end{cases} \tag{2.65}$$

So $S_{i,j}$ indicates whether document D_i is considered more relevant, the same relevant, or less relevant for a query Q. For the sake of brevity, we denote:

$$\alpha = \sigma(s_i - s_j). \tag{2.66}$$

The cross-entropy loss function $C(i,j)$ we will use for a single pair of documents is the following:

$$C(i,j) := -\frac{1}{2}(1 + S_{i,j}) \cdot \ln \hat{P}_{i,j} - (1 - \frac{1}{2}(1 + S_{i,j})) \cdot \ln(1 - \hat{P}_{i,j}), \tag{2.67}$$

which could be rewritten as:

$$\begin{aligned} C(i,j) &= -\frac{1}{2}(1 + S_{i,j}) \cdot \ln \hat{P}_{i,j} - (1 - \frac{1}{2}(1 + S_{i,j})) \cdot \ln(1 - \hat{P}_{i,j}) \\ &= -\frac{1}{2}(1 + S_{i,j}) \cdot \ln \frac{1}{1 + e^{-\alpha}} - (1 - \frac{1}{2}(1 + S_{i,j})) \cdot \ln(1 - \frac{1}{1 + e^{-\alpha}}) \\ &= \frac{1}{2}(1 + S_{i,j}) \cdot \ln(1 + e^{-\alpha}) - (1 - \frac{1}{2}(1 + S_{i,j})) \cdot \ln \frac{e^{-\alpha}}{1 + e^{-\alpha}} \\ &= \frac{1}{2}(1 + S_{i,j}) \cdot \ln(1 + e^{-\alpha}) - (1 - \frac{1}{2}(1 + S_{i,j})) \cdot (\ln e^{-\alpha} - \ln(1 + e^{-\alpha})) \\ &= \frac{1}{2}(1 - S_{i,j})\alpha + \ln(1 + e^{-\alpha}) = \frac{1}{2}(1 - S_{i,j})\sigma(s_i - s_j) + \ln(1 + e^{-\sigma(s_i - s_j)}). \end{aligned} \tag{2.68}$$

The partial derivative of $C(i,j)$ with respect to s_i and s_j is as follows:

$$\frac{\partial C(i,j)}{\partial s_i} = \sigma \left(\frac{1}{2}(1 - S_{i,j}) - \frac{1}{1 + e^{\sigma(s_i - s_j)}} \right) = -\frac{\partial C(i,j)}{\partial s_j}. \tag{2.69}$$

Suppose that θ is a parameter of the model f with a vector of parameters $\boldsymbol{\theta}$. We update the parameter using the partial derivative of θ for a single pair of documents:

$$\theta \leftarrow \theta - \eta \frac{\partial C(i,j)}{\partial \theta} = \theta - \eta \left(\frac{\partial C(i,j)}{\partial s_i} \frac{\partial s_i}{\partial \theta} + \frac{\partial C(i,j)}{\partial s_j} \frac{\partial s_j}{\partial \theta} \right), \tag{2.70}$$

where the parameter η is called the learning rate. The last expression is rewritten using the chain rule.

We learn the parameters using a gradient descent algorithm. The gradient update is done for each pair of documents with different labels in the default algorithm.

Let I_R be a set of pairs of documents D_i, D_j where D_i is ranked after document D_j and $D_i \rhd D_j$ in a ranking R:

$$I_R := \{(i,j) | D_i, D_j \in \mathcal{D} \wedge D_i \rhd D_j \wedge pos(D_i, R) > pos(D_j, R)\}. \tag{2.71}$$

The set I_R is a set of pairs, where $S_{i,j} = 1$ for every pair. There is a set I_R for each query ranking R. We will omit R from the notation since we will always use only a single ranking.

There is a high-level pseudocode of the *RankNet* approach:

Algorithm 5: *RankNet*

Input : Set of documents $\mathcal{D}_i$ and set of labels c_i for queries $Q_i \in \mathcal{Q}$, learning rate η
Output: Vector of model parameters
Initialize model parameters $\boldsymbol{\theta}$
repeat
  for $Q_i \in \mathcal{Q}$ **do**
    Create I_{R_i} by sorting the documents using model scores and the labels c_i
    for $(D_k, D_j) \in I_{R_i}$ **do**
      Calculate the gradient of $C(k,j)$ with respect to parameters $\boldsymbol{\theta}$
      Update model parameters using the gradient and η
    end
  end
until *converged*;
return $\boldsymbol{\theta}$

The algorithm iterates over a set of queries in a training set. The R_i is created by sorting the documents by model scores with respect to the current parameters. For each pair of documents with different labels, the algorithm uses the loss function $C(i,j)$ to update the model parameters. The gradient calculation and parameter update steps are symbolic and depend on the underlying model. For example, the convergence criterion can be when the metric difference between iterations is less than a threshold on the validation set. The parameters of the mode can be initialized, for example, randomly.

The *LambdaRank* approach presents two main contributions: a speedup in learning and direct optimization of a ranking metric. The *LambdaRank* algorithm performs parameter updates per query instead of *RankNet*. The total cost of a query we define as:

$$C := \sum_{(i,j)\in I} C(s_i, s_j). \tag{2.72}$$

The partial derivative of C with respect to a parameter θ is:

$$\begin{aligned}
\frac{\partial C}{\partial \theta} &= \sum_{(i,j)\in I} \frac{\partial C(s_i,s_j)}{\partial s_i}\frac{\partial s_i}{\partial \theta} + \frac{\partial C(s_i,s_j)}{\partial s_j}\frac{\partial s_j}{\partial \theta} \\
&= \sum_{(i,j)\in I} \sigma\left(\frac{1}{2}(1-S_{i,j}) - \frac{1}{1+e^{\sigma(s_i-s_j)}}\right)\cdot\left(\frac{\partial s_i}{\partial \theta} - \frac{\partial s_j}{\partial \theta}\right) \\
&= \sum_{(i,j)\in I} \lambda_{i,j}\left(\frac{\partial s_i}{\partial \theta} - \frac{\partial s_j}{\partial \theta}\right) \\
&= \sum_{(i,j)\in I} \lambda_{i,j}\frac{\partial s_i}{\partial \theta} - \lambda_{i,j}\frac{\partial s_j}{\partial \theta},
\end{aligned} \tag{2.73}$$

where we have defined

$$\lambda_{i,j} \coloneqq \sigma \left(\frac{1}{2}(1 - S_{i,j}) - \frac{1}{1 + e^{\sigma(s_i - s_j)}} \right) = \frac{\partial C(i,j)}{\partial s_i} = -\frac{\partial C(i,j)}{\partial s_j}. \tag{2.74}$$

If we use it with the set I where $S_{i,j} = 1$ for every element, we can write just:

$$\lambda_{i,j} = -\frac{\sigma}{1 + e^{\sigma(s_i - s_j)}}. \tag{2.75}$$

For each document D_i, we add the term $\lambda_{i,j} \frac{\partial s_i}{\partial w}$ if $(i,j) \in I$ and subtract the term if $(j,i) \in I$. Based on this observation, we can accumulate the sum per document. The sum can be rewritten as follows:

$$\frac{\partial C}{\partial w} = \sum_{D_i \in \mathcal{D}} \lambda_i \frac{\partial s_i}{\partial w}, \tag{2.76}$$

where we have defined:

$$\lambda_i \coloneqq \sum_{(i,j) \in I} \lambda_{i,j} - \sum_{(j,i) \in I} \lambda_{j,i}. \tag{2.77}$$

For example, suppose we have a ranking with only two documents D_1, D_2 and $D_1 \rhd D_2$. So, the set I has only one element $I = \{(1,2)\}$. The lambdas are:

$$\lambda_1 = \lambda_{1,2} = -\lambda_{2,1} = -\lambda_2.$$

The lambdas with a single subscript can be thought of as forces. We can attach an arrow to every document, and the length of the arrow denotes the force in which direction the document should move. The ranking has only two possible directions: move up or down. For each pair of documents (D_i, D_j), if a document D_i has $\lambda_{i,j}$ in the sum λ_i, then document D_j has $-\lambda_{i,j}$ in the sum λ_j. The *RankNet* optimizes pairwise errors. It is all well if it is the desired loss, but we often want to optimize a different measure, such as *NDCG*. Although *RankNet* can work well with these measures, there is a better approach. We multiply $\lambda_{i,j}$ by the difference in *NDCG* that we get by swapping only the documents D_i and D_j, leaving the others unchanged. The difference is denoted as $|\Delta_{NDCG}|$. So, we redefine the lambdas:

$$\lambda_{i,j} \coloneqq -\frac{\sigma}{1 + e^{\sigma(s_i - s_j)}} |\Delta_{NDCG}|. \tag{2.78}$$

Experimentally, this approach has been shown to optimize *NDCG* directly according to [14, 15]. So, the issue of optimizing a non-continuous measure like *NDCG* can be solved by this approach. In [16] they even proved that the *LambdaRank* approach optimizes an upper bound on *NDCG*. The *RankNet* cost function C is not the only one possible it is just the cost function that worked the best in the *LambdaRank* paper [12]. Even when we want to optimize a different Information Retrieval measure, such as Average Relevant Position (*ARP*), we can replace $|\Delta_{NDCG}|$ by $|\Delta_{ARP}|$ [14]. In Figure 2.6 we can see a ranking where the non-relevant documents are of turquoise color and the relevant documents are yellow. The arrows attached to the relevant document symbolize the accumulated lambdas. On the right are the lambdas that we get when using the described loss function without the multiplication by the *NDCG* difference. The arrow of

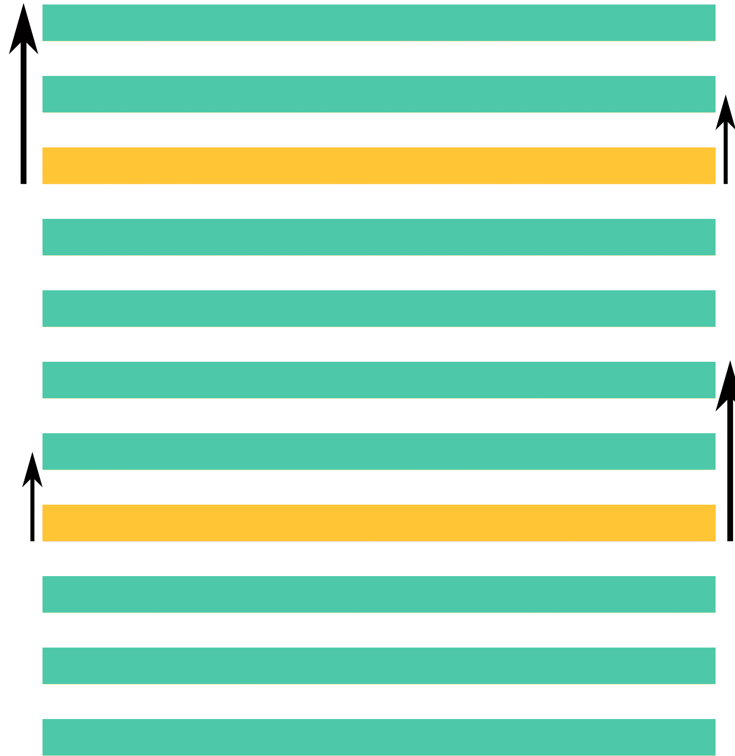

Figure 2.6 A ranking with depicted lambdas

the document at the lower rank is larger since there are more pairwise errors for the document compared to the other document. The arrows on the left side are the lambdas that we get when optimizing *NDCG*. In this diagram, only the relevant documents have arrows, but the other non-relevant documents also have lambdas that are not depicted for transparency. Now, we present a high-level pseudocode of the approach:

Algorithm 6: *LambdaRank*

Input : Set of documents $\mathcal{D}_i$ and set of labels c_i for queries $Q_i \in \mathcal{Q}$,
learning rate η,
ranking measure M
Output: Vector of model parameters
Initialize model parameters $\boldsymbol{\theta}$
repeat
  for $Q_i \in \mathcal{Q}$ **do**
    Create I_{R_i} by sorting the documents using model scores and the labels c_i
    Calculate $\lambda_{j,k}$ for each $(j,k) \in I_{R_i}$ and multiply by difference in M
    Calculate λ_j for each document D_j
    Calculate the gradient of C with respect to parameters $\boldsymbol{\theta}$
    Update model parameters using the gradient and parameter η
  end
until *converged*;
return $\boldsymbol{\theta}$

The algorithm is very similar to the *RankNet* algorithm. The model parameter up-

date uses a mini-batch update per query, but that does not give the important learning speedup. For now, we do not take the convergence loop and the query loop into account. The complexity of the *RankNet* algorithm depends on the number of pairs of documents in the set I. In the worst case, it could be quadratic. In Equation 2.76 we can see that the partial derivative computation is also quadratic for *LambdaRank*, since the computation of λ_i is linear. We compute λ_i and multiply it with $\frac{\partial s_i}{\partial w}$. The computation of $\frac{\partial s_i}{\partial w}$ can be a costly operation, for example, for a large neural network. In Equation 2.70 we have to compute the expensive operation per each pair of documents since the parameters of the models change every iteration. Thus we cannot store some results in memory. Computing a lambda in Equation 2.78 is a relatively inexpensive operation. In Equation 2.76, we perform the expensive operation only in the outer sum. This makes *LambdaRank*, which also has quadratic complexity, a much faster algorithm.

LambdaMART is a combination of *LambdaRank* and Multiple Additive Regression Trees (*MART*). The inner model used is *MART*. *MART* uses gradient-boosted decision trees for prediction [17]. *LambdaMART* has state-of-the-art ranking performance in pairwise supervised Learning to Rank [9]. We do not present the ideas behind *MART* since it is beyond the scope of this thesis.

2.2.7.1 Pairwise bias

In this section, we remove the gap between clicks and relevance labels. For now, we have assumed for pairwise loss in Section 2.2.7 that click means relevance and non-click means irrelevance. As we have seen in Section 2.17 the clicks are biased. Thus, click labels cannot be used interchangeably with relevance labels. Using click labels as relevance labels, we would train a biased ranker [9] that would give suboptimal results. In Article [9], they have not shown precisely why and how the ranker is biased in the context of the User Model used. The paper also removed only the position bias, not the document selection bias. Thus, we present our own derivation of the results. Until now, we defined the set I using $S_{i,j}$ where we used clicks c as true relevance labels. Now we redefine the expression $S_{i,j}$ using true relevance labels y as in Definition 2.65, and thus we also redefine the set I.

$$S_{i,j} := \begin{cases} 1 & \text{if } y(D_i) > y(D_j), \\ 0 & \text{if } y(D_i) = y(D_j), \\ -1 & \text{if } y(D_i) < y(D_j). \end{cases} \tag{2.79}$$

To show the bias, we used a naive estimator $C_{naive}(s_i, s_j)$ where we used the clicks as relevance labels again. To remove the bias, we create a new estimator $C_{aware}(s_i, s_j)$ that handles both position bias and document selection bias. The naive estimator is defined as follows:

$$C_{naive}(s_i, s_j) := c(D_i) \cdot (1 - c(D_j)) \cdot C(s_i, s_j). \tag{2.80}$$

Since the set I is now created using the true relevance labels, we use the click labels directly in the estimators. We rewrite Definition 2.72 for query cost as follows:

$$C_{naive} := \sum_{pos(D_i)<pos(D_j)} C_{naive}(s_i, s_j) = \sum_{pos(D_i)<pos(D_j)} c(D_i) \cdot c(1 - D_j) \cdot C(s_i, s_j). \tag{2.81}$$

The equation is rewritten using the definition of a set I for binary relevance labels. When a clicked document D_i is below a non-clicked document D_j, then the pair (D_i, D_j) is in

the set. In summation, we iterate on all pairs where D_i is below D_j in a ranking. Then in the sum, the only non-zero case is where D_i is clicked and D_j is not clicked. Again, we calculate the expected value of the cost function. We want to have the expected value equal to the value obtained when using the true relevance labels and thus unbiased.

$$\begin{aligned}
&\mathbb{E}_{o,\bar{R}}\, C_{naive}(s_i, s_j) \\
&\quad = \mathbb{E}_{o,\bar{R}}\left[c(D_i)(1 - c(D_j))C(s_i, s_j)\right] \\
&\quad = \mathbb{E}_{o,\bar{R}}\left[o(D_i)y(D_i)(1 - o(D_j)y(D_j))C(s_i, s_j)\right] \\
&\quad = y(D_i)C(s_i, s_j)\mathbb{E}_{o,\bar{R}}\left[o(D_i) - o(D_i)o(D_j)y(D_j)\right] \\
&\quad = y(D_i)C(s_i, s_j)(\mathbb{E}_{o,\bar{R}}\left[o(D_i)\right] - y(D_j)\mathbb{E}_{o,\bar{R}}\left[o(D_i)o(D_j)\right]) \\
&\quad \stackrel{1}{=} y(D_i)C(s_i, s_j)(\mathbb{E}_{o,\bar{R}}\left[o(D_i)\right] - y(D_j)\mathbb{E}_{o,\bar{R}}\left[o(D_i)\right]\mathbb{E}_{o,\bar{R}}\left[o(D_j)\right]) \\
&\quad = y(D_i)C(s_i, s_j)\mathbb{E}_{\bar{R}}\left[P(o(D_i) = 1|\bar{R}, D_i)\right]\left(1 - y(D_j)\mathbb{E}_{\bar{R}}\left[P(o(D_j) = 1|\bar{R}, D_j)\right]\right) \\
&\quad = y(D_i)C(s_i, s_j)\mathbb{E}_{\bar{R}}\left[P(o(D_i) = 1|\bar{R}, D_i)\right]\mathbb{E}_{\bar{R}}\left[1 - y(D_j)P(o(D_j) = 1|\bar{R}, D_j)\right] \\
&\quad = y(D_i)C(s_i, s_j)\mathbb{E}_{\bar{R}}\left[P(o(D_i) = 1|\bar{R}, D_i)(1 - y(D_j)P(o(D_j) = 1|\bar{R}, D_j))\right] \\
&\quad = C(s_i, s_j)\sum_{\bar{R}\in\pi(\cdot|q_i)} \pi(\bar{R}|Q_i)y(D_i)P(o(D_i) = 1|\bar{R}, D_i)(1 - y(D_j)P(o(D_j) = 1|\bar{R}, D_j)).
\end{aligned} \tag{2.82}$$

In Step 1 we used the independence of $o(D_i)$ and $o(D_j)$. The result is biased similarly to Equation 2.17. There is the same document selection bias and position bias. To remove biases, we again use inverse propensity scoring. The term in Equation 2.82 $(1 - y(D_j)P(o(D_j) = 1|\bar{R}, D_j)$ represents the biased non-relevance. A problem we have is that we do not know how to interpret non-click. A document is not clicked if it is unobserved at all or if the document is observed and not relevant. So, the case is different when a document is clicked. If a click occurs, we know that the document is relevant, as can be seen in Figure 2.4. An observation is:

$$P(c(D) = 0 \wedge y(D) = 0|q, \bar{R}, D) = P(y(D) = 0|q, D), \tag{2.83}$$

since every non-relevant document is not clicked. In other words: a set of non-clicked documents is a superset of non-relevant documents. The joint probability in Equation 2.83 can be rewritten:

$$P(y(D) = 0|q, D) = P(y(D) = 0|c(D) = 0, q, D) \cdot P(c(D) = 0|q, \bar{R}, D). \tag{2.84}$$

In the case where $y(D)$ is given in the probability $P(c(D) = 0|q, \bar{R}, D)$ we can write:

$$P(y(D) = 0|c(D) = 0, q, D) \cdot P(c(D) = 0|q, \bar{R}, D, y(D)) = (1 - y(D)). \tag{2.85}$$

The expression $c(D) = 0$ can be further decomposed in our User Model as follows:

$$c(D) = 0 \Leftrightarrow o(D) = 0 \vee o(D) = 1 \wedge y(D) = 0. \tag{2.86}$$

If we calculate the probability of not clicking:

$$
\begin{aligned}
&P(c(D) = 0|q, \bar{R}, D) \\
&\quad = P(o(D) = 0 \vee o(D) = 1 \wedge y(D) = 0|q, \bar{R}, D) \\
&\quad = P(o(D) = 0|q, \bar{R}, D) + P(o(D) = 1|q, \bar{R}, D) \cdot P(y(D) = 0|q, \bar{R}, D) \\
&\quad = 1 - P(o(D) = 1|q, \bar{R}, D) \cdot P(y(D) = 1|q, \bar{R}, D).
\end{aligned} \tag{2.87}
$$

If we assume that $y(D)$ is deterministically known, we can rewrite it as follows:

$$P(c(D) = 0|q, \bar{R}, D, y(D)) = 1 - P(o(D) = 1|q, \bar{R}, D) \cdot y(D). \tag{2.88}$$

That is enough to create a policy-aware estimator of pairwise loss $C_{aware}(s_i, s_j)$ for a pair of documents D_i and D_j:

$$C_{aware}(s_i, s_j) := c(D_i) \cdot (1 - c(D_j)) \cdot C(s_i, s_j) \cdot \frac{P(y(D) = 0|c(D) = 0, q, D)}{\sum_{\bar{R}' \in \pi(\cdot|q_i)} \pi(\bar{R}'|q_i) \cdot P(o(D) = 1|\bar{R}', D)}. \tag{2.89}$$

The query cost for the policy-aware estimator:

$$C_{aware} := \sum_{pos(D_i) < pos(D_j)} C_{aware}(s_i, s_j). \tag{2.90}$$

We again calculate the expected value with the new estimator. Following the last step in Equation 2.82 where $C_{naive}(s_i, s_j)$ is replaced by $C_{aware}(s_i, s_j)$ for the sake of brevity:

$$
\begin{aligned}
&\mathbb{E}_{o,\bar{R}}\, C_{aware}(s_i, s_j) \\
&\quad \stackrel{1}{=} C(s_i, s_j) \frac{P(y(D) = 0|c(D) = 0, q, D)}{\sum_{\bar{R}' \in \pi(\cdot|q_i)} \pi(\bar{R}'|q_i) \cdot P(o(D) = 1|\bar{R}', D)} \cdot \\
&\quad \sum_{\bar{R} \in \pi(\cdot|q_i)} \pi(\bar{R}|q_i) \cdot y(D_i) \cdot P(o(D_i) = 1|\bar{R}, D_i) \cdot P(c(D) = 0|q, \bar{R}, D, y(D)) \\
&\quad \stackrel{2}{=} (1 - y(D_j)) \cdot C(s_i, s_j) \cdot \frac{\sum_{\bar{R} \in \pi(\cdot|q_i)} \pi(\bar{R}|q_i) \cdot y(D_i) \cdot P(o(D_i) = 1|\bar{R}, D_i)}{\sum_{\bar{R}' \in \pi(\cdot|q_i)} \pi(\bar{R}'|q_i) \cdot P(o(D) = 1|\bar{R}', D)} \\
&\quad = \sum_{pos(D_i) < pos(D_j)} y(D_i) \cdot (1 - y(D_j)) \cdot C(s_i, s_j) \\
&\quad = \sum_{(i,j) \in I} C(s_i, s_j).
\end{aligned} \tag{2.91}
$$

In Step 1, we already used Equation 2.88, and in Step 2, we used Equation 2.85. After the last step, we have the desired unbiased output. We want to calculate $C(s_i, s_j)$ for each pair of documents D_i and D_j where D_i is below document D_j in a ranking and document D_i is relevant and D_j is not. To remove the bias, we have to estimate $P(y(D) = 0|c(D) = 0, q, D)$. If we had the true value of $P(y(D) = 0|c(D) = 0, q, D)$, we would know the probability of relevance for each document. In this case, it is useless to train a ranker since we would sort the document according to this probability of relevance. We do not know the exact probability value, but the equations 2.44 also estimate the probability. We can use this estimate to train a ranker since an estimation does not need to be optimal to provide better results than a ranker trained without the estimate.

Chapter 3

Implementation

In this chapter, we start by presenting the whole framework in which we conducted the experiments that we present in the next chapter. We begin by presenting the framework as a whole in the very next section and then describe the individual parts in more detail in each section alone. This chapter also contains some details on the implementation of the methods used.

3.1 Framework

In Figure 3.1 is a diagram of the framework depicted as a graph. The nodes with rounded corners represent an activity, and the rectangle nodes represent data sent between the activity nodes. We have data from a search engine of *VOD* and *e-commerce* platforms. We have the data presented in Section 2.2.0.1. Figure 2.1 denotes the data graphically. The data are: *Users*, *Documents*, *Historical rankings* $\bar{R}$ *and clicks* c, and *Search queries* $\mathcal{Q}$. The data are depicted in the graph. We describe the exact data in more detail in Section 3.3.

We start by sending the queries to a *Search engine* node that outputs a set of candidate documents based on a search engine metric from Section 2.1.1. Each candidate's score represents the match between the document and the query. All sets of candidates with scores for each query are denoted using the *Candidate documents* node.

The next activity node is *Feature factory*. For each user and document, we have some features. In this node, we assign to each candidate from the candidate set D the features of the document along with the score. We also assign each candidate in the candidate set the user features to each document. Each document in a candidate set for a particular query has the same user feature values. If there is the same query string q from a different user, the user features may differ between queries. Some different features are described in Section 3.6. Here, we also use another search engine to create additional features. The output is the *Dataset* node where we have a set of candidates with features partitioned by query $Q \in \mathcal{Q}$. This is also where the data are pre-processed as described in the following Section.

In the *Dataset split* node, we split the data into training, validation, and test. The data were divided by users to ensure each user belonged to a single set. The

training and validation datasets are used for training, and the test dataset is used only at the end of the process to get an estimate of the performance of the models. The training, test, and validation datasets were divided in a ratio of 70%, 15%, and 15%, respectively.

The *Bias estimation* node accepts the training data with the rankings and clicks. This is the node where we estimate the position and the document selection bias using the methods in Sections 2.2.4 and 2.2.5. The output of the node are *Bias estimators* which contains two models that can estimate position bias and document selection bias. These bias estimators can also predict a given document's relevance probability. We can use the probabilities to sort the documents, so the ranking performance of these models was also evaluated.

The next activity node is *Training of models*. This involves utilizing both the training and validation data to train the models and adjust their parameters using the validation data. In addition, unbiased rankers are trained using estimated bias, and models are trained using historical rankings and interactions. The node's output is composed of all the models we present in more detail in Section 3.5. The models are named *Baseline*, *Linear*, and *LambdaMART*. The model named *Baseline* is the model that ranks documents only by the score returned from the search engine. *Linear* model is presented in Section 2.2.3.3 and the *LambdaMART* is presented in Section 2.2.7.

The last activity node is *Counterfactual evaluation*. All models are evaluated and compared in this node using the apparatus in Section 2.2.3.1. The input of this node contains all the models we want to evaluate. Note that we use *Bias estimators* not only for bias estimation but also for ranking. Then we need the test dataset with the historical rankings and clicks.

3.2 Search engines

In the framework, we used two search engines. The first search engine is called *Search engine (Candidates)* and is used to obtain the set of candidates that can be relevant to the query. This engine used the Levenshtein distance to obtain the candidates. Both platforms wanted to use the measure to tolerate a typo in the query. The returned score was calculated as the maximum for all text fields in a document and, therefore, the maximum match for all text properties. Using the maximal possible score, the resulting score is normalized between zero and one. The maximum size limit for the candidate set was 750, and any documents that had a lower score and went beyond this limit were filtered out. Documents with scores lower than 10% of the maximum score were also filtered out. The search engine is case-insensitive. The search engine used was Recombee's search engine, which was also used for all the data we collected in the past. All sets of candidates passed on to the previous rankers were created using this search engine.

The second search engine used, called *Search engine (Features)* used the *BM25* scoring function. The parameters of the function were the most widely used: $k_1 = 1.2$ and $b = 0.75$. We used the engine to create additional document features. We indexed the documents in the search engine and then searched for the query per each document property. So, for each document, we have a *BM25* score for each text field. If we pass these scores as features to a ranker, the ranker can create weights for each text property

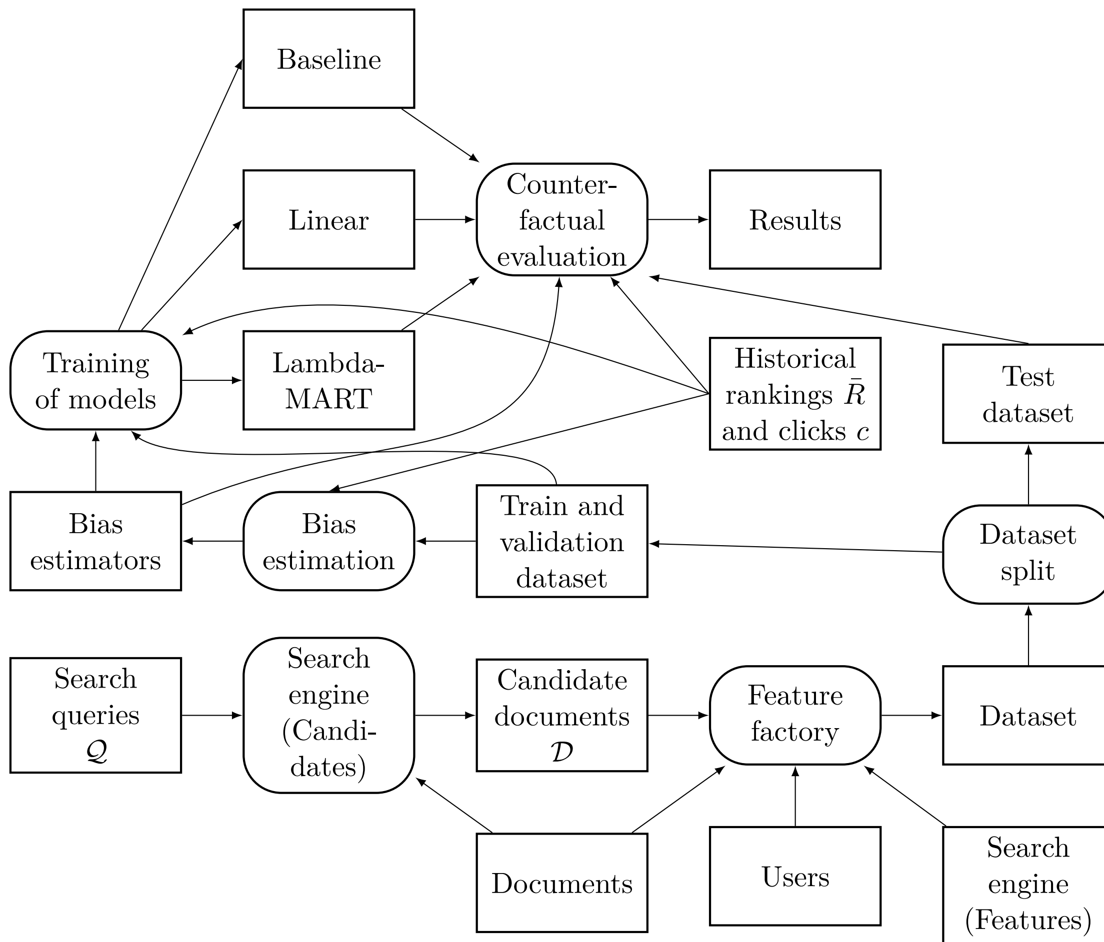

Figure 3.1 A graph of the framework in which we conduct experiments

of the document. For example, a title of a product is often more important than a description of the product. The **BM25** score was initially defined for a document D and not for a text property of the document. It can be envisioned as splitting one document with multiple text fields into multiple documents with only one field and then searching in those documents.

As a result, we have a global score of *Search engine (Candidates)*, which represents a general match between a query and a document. Then, we have a *BM25* score for each document text field. These values are then used in *Feature factory* in Figure 3.1. We used Elasticsearch as the implementation of *Search engine (Features)*.

3.3 Data description/preprocessing

As already stated, we have data from a search system of two web platforms: a *VOD* platform and an *e-commerce* platform. All data were supplied by Recombee, a cloud-based recommendation engine that functions as a service that offers recommendations based on user behavior and preferences. The input data files are not attached because they are private data that Recombee cannot disclose

Some data are already presented in Section 2.2.3.1. A Video on Demand (*VOD*) platform is a type of online streaming service that allows users to access video content on demand. This platform provides a library of movies, TV shows, and other video content that users can access from anywhere with an Internet connection. The *e-commerce* platform allows users to buy a product online. The platform we use is focused on the entertainment industry.

The usual scenario is that a user comes to the website and tries to find the desired content using a search box. The user is shown some results and decides which document to view in detail using a click. The user can click on multiple documents, not just one. The user can also decide not to click at all.

For the *VOD* case, there is an infinite scroll in which users can view all documents in the candidate set. The users are shown around 8 documents (depending on the screen size), and then they decide to click to see the next 8 documents. Therefore, Condition 2.20 is theoretically satisfied.

Whereas for the *e-commerce* platform, there is a typical top-k scenario where only the best 6 documents are shown to the user, and the user cannot view more documents. If a document relevant to the query and for the given user is not visible, then a more specific query must be issued to view other documents. Condition 2.20 cannot be true for the top-k scenario, as already stated in Section 2.2.3.1. We cannot validate if at least Condition 2.24 is satisfied. The condition states that if a document is relevant to a query, there must be a non-zero probability of observance across all rankings that can be shown to the query. We do not know if a document is relevant, and thus we cannot check if the document is shown in a ranking. There were multiple different models deployed. Some of the models included randomization of the results. There is also randomization caused by personalization of the ranking for different users or by A/B testing other models or parameters. Based on these reasons, we presume that the condition is satisfied.

For both platforms, we have the data from three consecutive days. We used only users who clicked on a document at least once. In other words, we filtered all users without a single document interaction. For the *e-commerce* platform data, we have 8040,

1660, and 1725 users in the training, test, and validation sets, respectively. For the *VOD* platform data, we have 22364, 4708, and 4807 users in the training, test, and validation sets, respectively. For the *e-commerce* platform, there are around 950000 recommendation queries, and 17000 were clicked, and for the *VOD* platform, there are around 300000 recommendation queries, and 32000 were clicked. The *e-commerce* platform data contain around 16000 unique queries, and the *VOD* platform data around 4000. What is very different is the number of recommended documents. The *e-commerce* data contain 26000 unique documents that were returned in rankings to users, whereas the *VOD* data contain only 350. For the *e-commerce* dataset, there are 26000 unique documents in 950000 different recommendation queries, whereas for the *VOD* platform, there are only 350 documents in 300000 unique recommendation queries. For a query in the *VOD* platform, there is usually a much smaller set of candidate documents, so there is a much smaller space for reordering of the documents. The presentation of the results and the number of documents that were returned for a query make the two datasets very different, which is good for comparison of the methods.

In addition to text fields of documents, the platforms also have their own properties that are not textual. Since we cannot present the exact properties, we present just how we pre-processed them. We need the features in numerical values to be used by the *Linear* model. Each numeric property is left as is. When a categorical document property has fewer than 50 unique values, the property is one-hot encoded. If a categorical property has more than 50 unique values, then the category is replaced by an integer. This causes an ordering of the values, but it is better than removing the entire property. If a document property is a set of values, then if there are less than 50 unique values, then the values are also one-hot encoded. If there are more than 50 values, the property is remoted. If a property after preprocessing is a constant value, then the property is removed. The textual properties are used in Section 3.2 and then removed. Then these document features are used in the *Feature factory* node in Figure 3.1. Then each feature is normalized between 0 and 1 throughout the whole dataset. We normalize the values using the training dataset. The validation and test datasets are also normalized, but using the values from the training dataset, therefore the values in test and validation datasets can be out of the interval.

When a user is writing a query, every few hundred milliseconds, the query is sent to the search system to give the user at least some results as the user is typing. We do not want to take these queries into account since the queries are never clicked (except the last). All queries five seconds apart in time at maximum are considered to be in a single search session. We are interested in the last query from the session. This is the point where the user decides whether to click or not. All other queries in the session are not important to us. When a user typed only the first part of the query and the desired document was shown, the user also finished the session.

Both platforms also have features for their users, such as gender, age, nationality, etc. These features could also be used to improve the quality of the ranking. For now, we assumed that the relevance of a document depends only on the query without dependence on a user. A document is either relevant or non-relevant to a query, and it is the same for all users. In the real world, a document relevant to one user may not be relevant to another user, even for the same string query. So, in addition to the presented theory, we assume that the relevance also depends on the user, and thus all the probabilities are

conditioned on a query and the user. Actually, we use the user features instead of the user identifier in the models and assume that the fitted model can generalize to other feature vectors. We use the assumption that users that are somewhat similar will have the same preferences for relevant items. Suppose that we have users where we have user age, nationality, gender, language, and maybe others. Now suppose that there are two users with completely the same features, but the age feature difference between the users is three. Furthermore, users have similar preferences or interact with similar documents (presented in Section 3.6). These users are likely to have the same preferences when the same query is given. The same is true for the features of the document also. If we want to personalize the rankings for a given user, then the models that provide the scores by which the documents are sorted must depend on the features of the user. We use user features in learning all models: the *Linear* ranker, estimating both biases and learning the *LambdaMART* model. We preprocess the user features in the same way as the document features, and the resulting vector that we use to learn a model is a simple concatenation of the vectors.

As a result, we have 310 features for the *e-commerce* dataset and 330 for the *VOD* dataset. Since the data are large, we had problems with memory while learning the models. To solve the problems, we did a feature selection to reduce the dimensionality of the data, and thus reduce memory usage. When a model learns using fewer features, it does not mean that performance will be worse. On the validation data, the performance can be even better due to the overfitting of the training data.

Since we use the *MART* algorithm in *LambdaMART* we can use the feature importances the model creates. We have not presented how the algorithm works, but it is an iterative algorithm that iteratively builds decision trees. Each tree is built by selecting the best splits by a feature based on a predefined criterion. When a feature is selected, the feature provides the best split among some other features. The goal of splitting by features in a decision tree is to create subsets of the data that are as homogeneous as possible with respect to the target variable, which helps later with predictions. A predefined criterion can be, for example, the Gini impurity or the information gain. When we aggregate the number of times a feature was used, we can interpret it as the importance of the feature for the model. The importances are only relative to other features. When the algorithm is fitted, we have for each feature the count of how many times the feature was used. We normalize the counts between 0 and 1 using the largest count. We remove all features with importance less than 0.0075 for both datasets. The threshold is very small, and it mainly removes features with an importance of 0. The threshold was selected mainly to remove the zero-importance features and the features with importance very close to 0. For the *e-commerce* we retained 69 features, and for the *VOD* dataset, we retained only 17 features.

3.4 Bias estimation

In this section, we present some of the implementation details of the bias estimation as presented in Sections 2.2.4 and 2.2.5.

To estimate position bias using the regression-based *EM* algorithm, we need a binary classifier capable of predicting probability. We need to estimate the probability of relevance. The classifier takes the ranking features as input (user features are also included).

The labels are sampled from a distribution. We used Gradient Boosted Decision Trees (*GBDT*) as the classifier from the *LightGBM* framework [18].

For each iteration of the Algorithm 2 we conducted five iterations of the *GBDT* model and then resampled the data and continued with the next iteration. In an iteration, we used all weak learners from previous iterations. We hand-tuned the learning rate parameter to 0.1 based on validation data. With this value, the log-likelihood of the User Model data converged in a reasonable time. We used the hand-tuning of the parameters more times since a proper grid search of the parameters would be very computationally demanding since the data are large. We used a simple convergence criterion to stop the algorithm, where we hand-tuned the number of iterations to nine since the value converged before that point for both datasets. We left all other parameters to the default values. The classifier used to estimate the document selection bias was exactly the same with the same parameters.

Since the classifier supports the weighting of the samples, we created the weights using the probability of each sample. In Algorithm 2 we sample the labels from the distribution:

$$P(y(D)|c_i(D_j), q_i, D_j, pos(D_j, \bar{R}_i)). \tag{3.1}$$

Each data point has a probability of being relevant and non-relevant, and we used the probability as the weight in the classifier. The classifier prefers to successfully classify the input variables with higher weights. Intuitively, a document that we are more confident in assessing relevance should receive higher weight in the classifier than a document for which we are less confident. We also use this technique in document selection bias estimation. This is the only improvement we did to the default algorithm. We present how it affected the performance in the following chapter.

We use a technique called *Bias clipping*. When learning an *unbiased* model using the estimated bias, there is a chance that the estimator has a high variance and predicts a very small probability. When the propensity is applied to a data point, the weight is very large compared to the size of the dataset. The ranker then can be overfitted to the point and is not able to generalize well. To solve the problem, we set a threshold on the minimal probability. This produces a biased model, but then the variance of the model is smaller, which can result in better performance. This is an example of *Bias-variance tradeoff*. We use clipping only on training data and never on test or validation data. We use the value of the parameter 0.001.

As already stated in Chapter 2 the model that estimates the document selection bias used the estimation of position bias of the second model.

3.5 Training of models

In this node, we train all ranking models without those already trained from the *Bias estimation* node. We start by describing the *Baseline* model.

3.5.1 Baseline

As a baseline, we use the bare search engine scores. It is a minimal setting in which a website could provide a way to search for documents. It is completely not personalized,

and the only feature used is the match between a text property of a document and a query. The provided scores are from *Search engine (Candidates)* where we used the Levenshtein score. The *Baseline* model can also be seen as a model that uses just the global score search score property and performs a sort. This model is not trained.

3.5.2 Linear

The second model is named *Linear*. This approach was presented in Section 2.2.3.3 where we directly optimized an estimator loss. As the name suggests, we used a regularized linear regression model using the L_2 regularization of the model coefficients. For this model, we have not used any library and implemented our own gradient descent algorithm to fit the coefficients. We briefly present the linear model [19]. We mark a feature vector of a document as:

$$\mathbf{x} = (x_1, x_2, \ldots, x_m, 1), x_i \in \mathbb{R}, \tag{3.2}$$

where m is the number of features and 1 at the end is the intercept. We mark the set of coefficients/parameters/weights as follows:

$$\mathbf{w} = (w_1, w_2, \ldots, w_m, w_{m+1}), w_i \in \mathbb{R}. \tag{3.3}$$

The w_{m+1} is the weight of the intercept. In this model, we assume that there is a real score for the document Y that is linearly dependent on the features and thus can be calculated as:

$$Y = w_1 x_1 + w_2 x_2 + \ldots + w_{m+1} + \epsilon = \mathbf{w} \cdot \mathbf{x}^\top + \epsilon, \tag{3.4}$$

where ϵ is a random variable and we assume $\mathbb{E}\ \epsilon = 0$. The intercept with weight w_{m+1} represents a starting value when all other weights are zero and ensures that $\mathbb{E}\epsilon = 0$. We estimate the score $\hat{Y}$ of a document based on the features as:

$$\hat{Y} = \mathbb{E}\ Y = \mathbf{w} \cdot \mathbf{x}^\top. \tag{3.5}$$

We also add a term that penalizes the weights of the model with L_2 regularization. It creates a biased estimate, but it helps us with the colinearity problem and can help with overfitting.

$$\hat{Y} = \mathbf{w} \cdot \mathbf{x}^\top + \lambda \sum_{i=1}^{m} w_i^2. \tag{3.6}$$

The parameter λ controls the weight of the regularization. The score $\hat{Y}$ is then used in the $\overline{rank}$ function.

We start by initializing all the coefficients to 0. Initializing the parameters at random provided the same results. All of the following parameters were manually tuned on the basis of validation data. We selected the parameter $\lambda = 0.0001$. Note that all features are normalized between zero and one, and thus have the same weight. The learning rate parameter in the first iteration is $\eta = 0.05$. We also used a learning rate decay, in which, at each iteration, we used the learning rate of the previous iteration multiplied by 0.95. It should ensure that the gradient step is not too large and thus converges better. Since we update the gradient per each query in mini-batches, we used a gradient

momentum technique. It helps to overcome the issues of oscillations and getting stuck in local minima or saddle points. Momentum is inspired by the concept of momentum in physics, where a moving object tends to continue moving in the same direction due to its inertia. In the context of gradient descent, momentum is introduced by adding a fraction of the previous update to the current update. This helps to smooth out the optimization path and accelerates convergence. We decided to keep the last three gradients, and each gradient has an exponentially decaying weight of 0.4. Therefore, the weights for the previous gradients are 0.4, 0.16, and 0.064. We selected the number of iterations $n = 100$ because higher values have not provided better results even for different decay values of the learning rate.

3.5.3 *LambdaMART*

We used the *LambdaMART* implementation of the *LightGBM* framework [18]. We use the parameter *objective* set to *lambdarank* in *LGBMRanker* and the default optimized metric is $NDCG@k$. The parameter k is set by *lambdarank_truncation_level* of *LGBMRanker*. We present the values of this parameter in the following sections. We used a learning rate of 0.4 with a learning rate decay of 0.995 with 500 iterations. All other parameters used the default values.

The *LGBMRanker* supports the setting of weights for the documents, but the interpretation of the weights is different from what we want in Section 2.2.7.1. The model gives weights to the lambdas in Equation 2.77. We want to give weight to a pair of documents as in Equation 2.89. So, we implemented support for these weights in the ranker source code. We changed the meaning of the weights. The framework is open source, so it was possible. The code is shown in the enclosed media.

3.6 Feature engineering

We have already presented most of the features used for the ranking or document. We start by summarizing the features and then presenting the new ones.

We presented the document properties and the user properties. Then we also used the search score per document and then search scores per text property of the document. In addition to these properties, we also used the length of the search query since information on how long the query is can be important. The only personalized features are the user features. Now, we present two other personalized features using the methods presented in Section 2.2.6.

We must specify how we aggregate the interaction in the rating matrix $\mathbf{R}$. Since we only use the clicks on the documents, we aggregate all the clicks to the rating matrix by summing all the interactions between a user and a document in the cell representing the user and document. A value in a cell represents the user's preference for the item. Since there may be users who interact with a single document many times, we limit the maximum number in cells to ten. Without the limitation, a cell can contain a very large number that can outweigh different cells. So, every cell in the rating matrix has a maximum value of ten. We start by presenting how we used the *ItemKNN* algorithm and then show how we used the *UserKNN*.

Suppose we have a user and want to recommend some documents to the user based on a user query. *ItemKNN* algorithm iterates over all items with which the user interacted. For each interacted item, a set of similar items is found. As a result, we have a vector of document scores, and the score is calculated as shown in Section 2.2.6. Scores represent possible user preferences for the document. We set the scores created by *ItemKNN* to the candidate documents returned for the search query. This makes a personalized feature for the given user that respects individual preferences.

The case for *UserKNN* is very similar. The Algorithm 3 output is also a vector of document scores. So we use it precisely the same way. As a result, we have two personalized features of user preference for the documents.

One problem is with evaluation when we create the interaction matrix. We cannot create the matrix for all users and items before evaluation since, in this case, we would have the data from the future, which we obviously do not have in a real application. Meaning that when we are evaluating a model and creating a ranking for a historical query we cannot use user interaction for the query. At this time, we do not know which document the user will click on. We know it since we have historical data, but we do not know it in a production environment. So, to estimate how the model would behave in production, we must handle this problem.

We solve it in the following way. We create the rating matrix for all training users. We aggregate all of their interactions into the matrix. The test and validation users are also in the matrix but have no interactions in the cells. The rows for validation and test users are zero vectors. Each query $Q \in \mathcal{Q}$ is a triplet (q, t, u). The triplet has a string query q, timestamp t, and user u as already presented in Section 2.2.0.1. When creating these two features for a query Q, we remove from a copy of the filled matrix $\mathbf{R}$ all the user interactions that occurred at a timestamp equal to or greater than t. When creating the features, we only have the interaction that the users already did in the past since we removed all current and future interactions. This is a very similar situation to that in the production environment. When we create the features for the test and validation user queries, we do not remove the interactions from a copy of the matrix $\mathbf{R}$, but instead, we add the interactions to the matrix. When creating the features for a query Q of a test/validation user, we added all the interactions of the user before the query was issued. This ensures that we do not use interactions of users in the test or validation sets while creating the features.

3.7 Counterfactual evaluation

In the node *Counterfactual evaluation* we evaluate the trained models using the unbiased estimators presented in Section 2.2.3.1. We can use the policy-oblivious estimator for the *VOD* platform data since the preconditions are satisfied. For the *e-commerce* dataset, we must use the policy-aware estimator.

We compare the methods using the *DCG* ranking measure. It is important to note that optimizing $DCG@k$ and DCG is the same as long as there are fewer or the same number of relevant documents than k. In this case, the ranking of optimal $DCG@k$ and optimal DCG is the same. If there are more relevant documents than k, then $DCG@k$ is not sensitive to the ranks of relevant documents above the threshold k. Optimizing $DCG@k$ and $NDCG@k$ is the same, as $NDCG@k$ is normalized using a constant value.

It is a difficult task for counterfactual evaluation to estimate the normalization constant [4, 20]. We do not have the true relevance labels, so we do not know which and how many documents are relevant to create the ideal *DCG*. In our model, we only know that the clicked documents are relevant, but there may be some others. Therefore, we use *DCG@k* for the evaluation instead of *NDCG@k*, which has a better interpretation.

We replace λ in the Algorithm 1 for the *Linear* model with *DCG*. So, we optimize the global *DCG*. We assume that a situation with more relevant documents than k is very rare. Thus, optimizing *DCG* and *DCG@k* is mostly the same. The *LambdaMART* implementation optimizes the *NDCG@k* measure. Since the method is initially meant for supervised Learning to Rank, and we use the clicks as relevance labels, the implementation creates the normalization constant using clicks. It is not a problem, since it is just a normalization constant that does not change the optimal values of the loss function. The bias estimators that we also use for the ranking evaluation do not optimize *DCG*, since they optimize a pointwise probability of relevance.

In addition to the models presented, we can also compare the average performance of the previous solution deployed when the data were collected. We mean average since there were multiple rankers, and we evaluate them all together as one ranker. We will call it the *Legacy* model. We have the rankings $\bar{R}$ of the previous *Legacy* model, so we use it for ranking evaluation.

Performance evaluation of a search system is not straightforward. There is no one-size-fits-all system that performs well across diverse datasets and users. Every platform has different documents to search for and users with different preferences. Each platform can have different business rules that often go against some evaluation measures. A proper evaluation of a search system's performance is always in a production environment using some A/B testing. However, often it is hard to get a model to production due to several reasons such as slow prediction, a lot of programming, low traffic, etc.

It is often easier to evaluate the model offline, even when the accuracy is worse than in production. When the performance in the offline evaluation is good enough, then start the deployment to production. It is important to use some evaluation measures that will correlate with the performance measures of the search system in the online environment.

We will use different evaluation measures that test the search system from different perspectives. We will evaluate using the *DCG@6* metric for the *e-commerce* platform, since only the top-6 documents are shown to the user, and we are not interested in ranking documents above rank 6. For the *VOD* platform, we are interested in the whole ranking, so we will optimize the *DCG*. Based on this, we set the parameter *lambdarank_truncation_level* to 6 for *e-commerce* and 750 for *VOD*, which is the total number of candidates that we use. We will use mainly this measure to compare the newly fitted models with the *Lagacy* model. The other measures are not completely suitable for this comparison as described further.

The next measure we will use is *recall@k*. This measure is invariant to the ranks of the documents. In this measure, only click labels are used to estimate performance. Although clicks do not indicate relevance and are biased, they provide an indicator of relevance that we use in this metric. Calculating an unbiased estimate of *recall@k* using true relevance labels is not as straightforward as for the previous measures. This measure is unsuitable for comparing newly created models with the ranker *Legacy*, as

clicks are biased by the presentation of the document. Therefore the *Legacy* ranker has an advantage. For example, a *Legacy* ranker for the *e-commerce* platform has the $recall@6$ equal to 1. The reason is that a user can click only on the first 6 documents. The $recall@k$ is robust to an unbalanced dataset, as we have. Most of the items retrieved from the search engine are not clicked. We want as much as possible from the clicked item in the top-k items. An optimal value of $DCG@k$ for a single ranking implies an optimal value for $recall@k$, but the other implication does not hold. So optimization of $DCG@k$ can be seen as optimization of $recall@k$ in the binary relevance case. Bias estimators do not optimize the $DCG@k$ directly. We will experiment with different values of k as we present in the next chapter.

The last metric we use is the *Jaccard index*. According to [5], the metric strongly correlates with online performance, so we use it also in model evaluation.

Chapter 4

Experiment design

In this chapter, we present the experiments we performed and how we evaluated them. We applied the experiments to both datasets. In the following chapter, we show the results and describe the findings.

In the following experiments, we are interested in these ranking quality metrics: *DCG@k*, *ARP*, *recall@k*, and *J@k*. We are mainly interested in *DCG@k*. The measure assigns to a higher position more weight compared to a lower position. It is exactly what we want since, for the user experience, the top positions are the most important. For both platforms, we evaluated only the *DCG* metric with k equal to the values for which the models are optimized. For the *VOD* dataset, we optimized the global k. It could be seen as infinite k. For the *e-commerce* platform only the top-6 documents are visible, so we are interested only in *DCG@6*. We have defined *DCG@k* (in Section 2.2.0.2) with a minus, so lower values mean better and the metric is non-positive for all values. All other metrics are positive, so we always show the absolute value to be consistent with other metrics. Therefore, larger values mean better.

We evaluated *recall@k* as well as *J@k* for the following values for the *VOD* dataset: 1, 3, 6, 15, and 30. For the *e-commerce* dataset, we evaluated 1, 3, and 6. Higher values did not provide significant differences and, therefore, no information.

We also evaluated the global *ARP* for both datasets. This metric is very sensitive to a relevant document placed in a high position compared to *DCG* and, therefore, can provide complementary information. It should be noted that, for the metric *ARP*, a lower value indicates better performance than all other metrics. All experiments were performed on the test data unless otherwise stated.

The training of the models and the evaluation of *DCG* and *ARP* were done using the policy-aware estimator unless otherwise noted. The policy-oblivious estimator cannot be used for the *e-commerce* data. The evaluation using policy-oblivious and policy-aware estimators is the same for the *VOD* dataset.

4.1 Expectation-Maximization experiments

First, we show how we improved the convergence of the *EM* algorithms. In addition to the default algorithm as presented in [6], we used some improvements. We presented the

improvement in Section 3.4. We ran the algorithm with and without improvement to see the difference. For both runs, we monitored the log-likelihood of the data presented in Equations 2.42 and 2.49 in addition to the ranking metrics. The ranking metrics were evaluated using the policy-aware estimator for both datasets. We used all the personalization features. We first performed this experiment because we used the created models that performed better in the next experiments. We call the model that estimates only position bias a *Position* model and the model that estimates document selection bias a *Selection* model.

4.2 Personalization experiments

We are interested in how personalization improves the ranking performance from the perspective of the metrics used. We divided the personalization features into two parts. The first one that we named *Content-based* personalization is created by the user features. On the other hand, we named *Collaborative* personalization the features presented in Section 3.6 using collaborative filtering algorithms. We are interested in how each approach influences performance and how performance changes when applied together compared to a version where the personalization features are missing.

For *Collaborative* personalization, we are also interested in how the two features perform with a different value of k. Therefore, we performed the following experiments.

For *Content-based* personalization, we trained and evaluated the rankers with and without *Content-based* user features.

For *Collaborative* personalization, we tried these values of k for the *e-commerce* platform: 1000, 500, 250, 125, 67, 33, 16, 8, 4, 2, and 1. For the *VOD* platform, we tested only the following values of k: 250, 125, 67, 33, 16, 8, 4, 2, and 1. The algorithm is slower for higher values of k, so it makes sense to check if there is a threshold above which the performance is not better. When trying the parameter k, the *Content-based* features were omitted.

Finally, we tested *Content-based* personalization and *Collaborative* personalization together with the best-performing parameter of k.

In addition, we will show the feature importances of the personalization features created by the models as presented in Section 3.3.

We used the policy-aware estimator for these experiments to train and evaluate the models. In other experiments, we used both *Collaborative* and *Content-based* features. The selection of the parameter k to be used in other experiments will be discussed and done in the next chapter.

4.3 Unbiased rankers experiments

We tested the influence of learning an unbiased ranker for the *Linear* model presented in Section 2.2.3.3 and for the *LambdaMART* model presented in Section 2.2.7. We are interested in different bias estimators including the *naive* estimator. We are also interested in how the estimation of:

$$P(y(D) = 0 | c(D) = 0, q, D), \tag{4.1}$$

will improve the performance of the *LambdaMART* model as presented in Section 2.82. We conducted the following experiments: For both datasets, we learned both models using their estimators. So, for *Linear* we fitted three models, and for *LambdaMART* we fitted also three models. The third does not estimate the probability in Equation 4.1. For the *e-commerce* platform, we omitted the policy-oblivious estimator for *Linear* model since the Condition 2.20 is not satisfied.

4.4 Comparison of rankers

We trained all rankers using the parameters that provided the best ranking performance and compared them with each other using the metrics presented. In addition to these models, we also compared the *Legacy* model, which is the ranker that was in production at the time the data were collected for both platforms. Each platform used a different *Legacy* model. We also compared a linear model called *Random* similar to *Linear* that only randomly initialized the weights when fitting to the data (note that all the feature values are normalized between 0 and 1). This model serves as a verification check that we have not trained some trivial models. Another advantage is that the model can tell if there is room for reranking of the documents. When the *Random* model performs well, for example, in *recall@k* metrics, it means that only a few documents can be reordered and there is not much room for reordering. The model serves as a comparison between the *e-commerce* and *VOD* datasets. The last model we compare is the *Baseline* model, which ranks documents only based on the scores of the documents returned from the search engine. From this comparison, we will check whether the trained models improve performance compared to a bare search engine.

In addition to this, we also show the feature importances created by the *LambdaMART* model to see if the features designed are important or not for the ranking of the documents.

Chapter 5

Discussion of the experimental results

In this chapter, we show and describe the results and the outputs of the experiments presented in Chapter 4.

5.1 Expectation-Maximization experiments results

In Figure 5.1 there are plots that denote the dependence of the log-likelihood on the number of iterations. In the first row, we have the plots for the *e-commerce* dataset. On the left is the improved version and on the right is the original version of the *EM* algorithm. The second row is the *VOD* dataset. In each graph, there are two *EM* models: the *Position* model and the *Selection* model. For each model, there is a measurement for the training and test data.

The weight improvement of the samples helped the *Selection* model the most. Convergence with the improved version was faster for both datasets. All models converged to the 9th iteration. The *Position* model fits the data better than the *Selection* model in both datasets. The curves for both test and training data are identical, indicating that the models do not suffer from overfitting to the training data.

In Figure 5.2 is a comparison of the ranking performance of the *Position* model and the *Selection* model for the *e-commerce* platform. Each model has a version with and without weight improvement. The ranking performance of the *Position* model and the *Selection* model is very similar.

For the $DCG@6$ metric, there is an improvement in the version with weights for both models. The ARP metric is worse for the version with weights. There is an improvement in the $DCG@6$ metric and a decrease in the ARP metric. It means that relevant documents to position 6 are ordered better, but relevant documents above position 6 are ordered worse compared to the original version.

The differences between the original version and the version with weights are insignificant in all other metrics compared to the metrics ARP and $DCG@6$.

A similar graph is shown in Figure 5.3 for the VOD platform. As already stated, due to the limited number of recommendable documents compared to the *e-commerce*

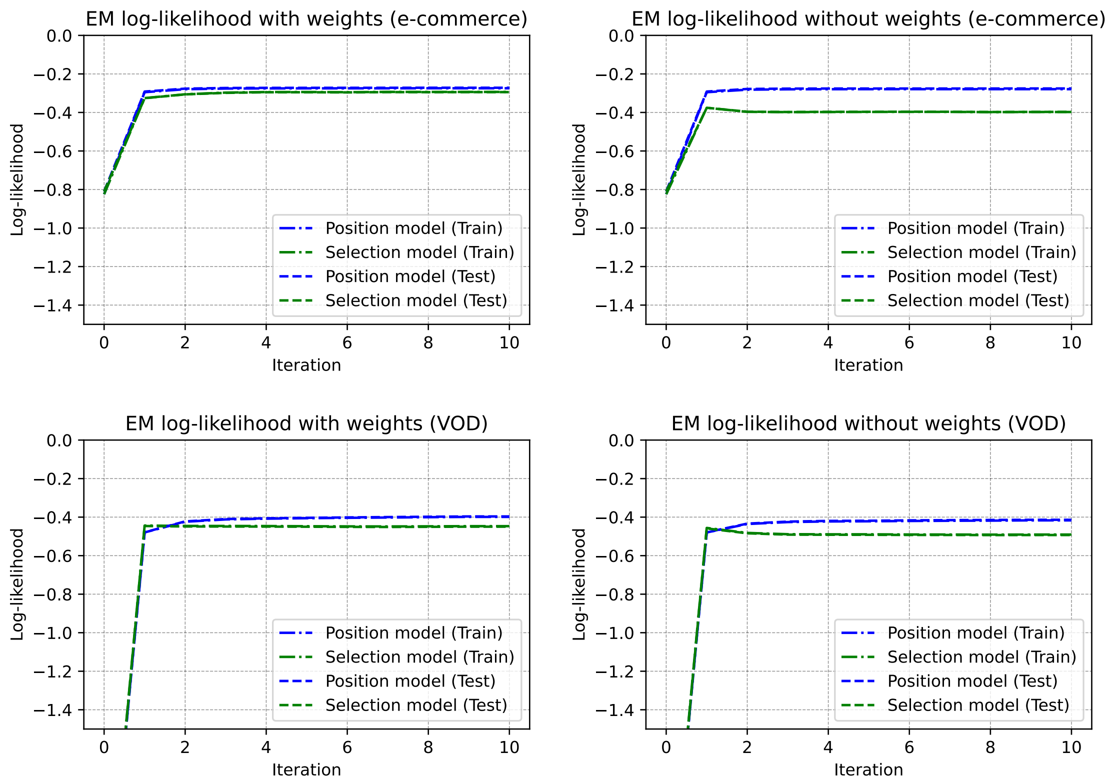

Figure 5.1 Comparison of data log-likelihood progression with and without weight enhancement

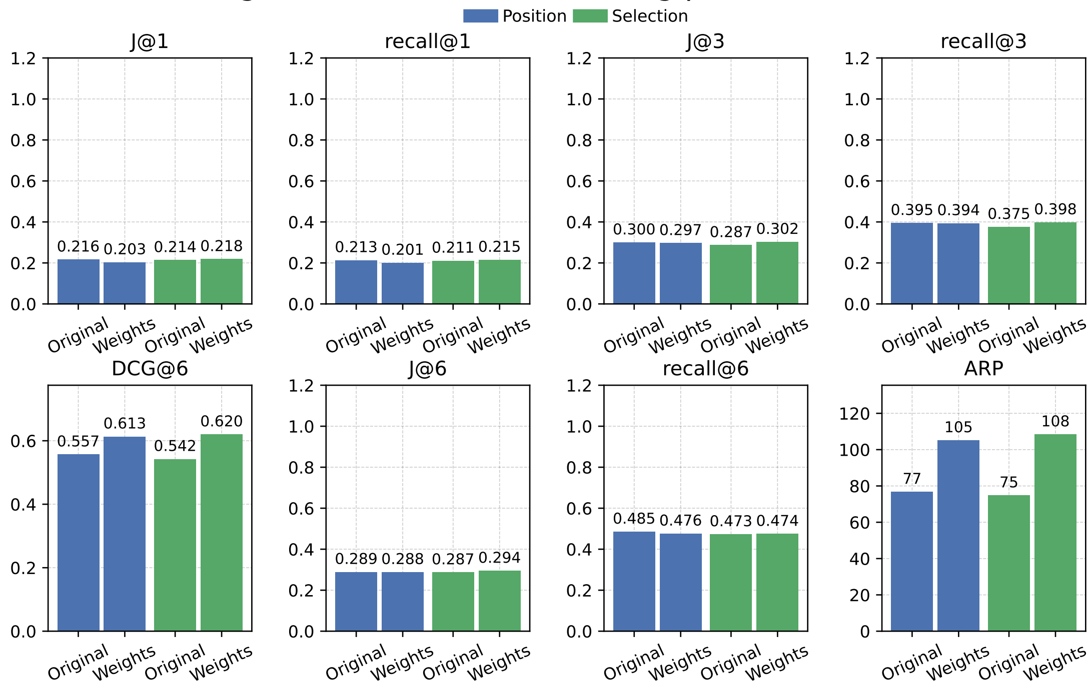

Figure 5.2 Comparison of the *EM* algorithms with the improvement on the ranking performance for the *e-commerce* platform

platform, there is not the same room for the reordering of the documents. The situation is very similar to that in Figure 5.2. There is also an improvement in *DCG* and a decrease in *ARP*.

For all other experiments, we used the version with the weights, since the convergence is better and the ranking performance of the *DCG* metrics is better in which we are more interested compared to *ARP*.

5.2 Personalization experiments results

In Figure 5.4 is shown the evolution of *DCG*@6 on the parameter k used in *ItemKNN* and *UserKNN* for the *e-commerce* platform. The parameter k represents the number of neighbors used in the algorithms. In the graph, the x-axis has a logarithmic scale except the interval between 0 and 1, where the scale is linear. We will use the label λ-*MART* as a synonym for *LambdaMART*.

There are 4 curves for the four models that we trained. The most significant impact of the parameter k is on the *Position* and *Selection* models. The most significant improvement is for values lower than 4. Then the performance is very similar. Each model used randomization in the learning phase, so there are some deviations.

The *Linear* model improved the metric for values of k less than 4. For larger values, *DCG*@6 is the same with some deviations. The impact of the parameter k is less significant than the other rankers.

The same situation is for the *LambdaMART* model where the improvement is mainly

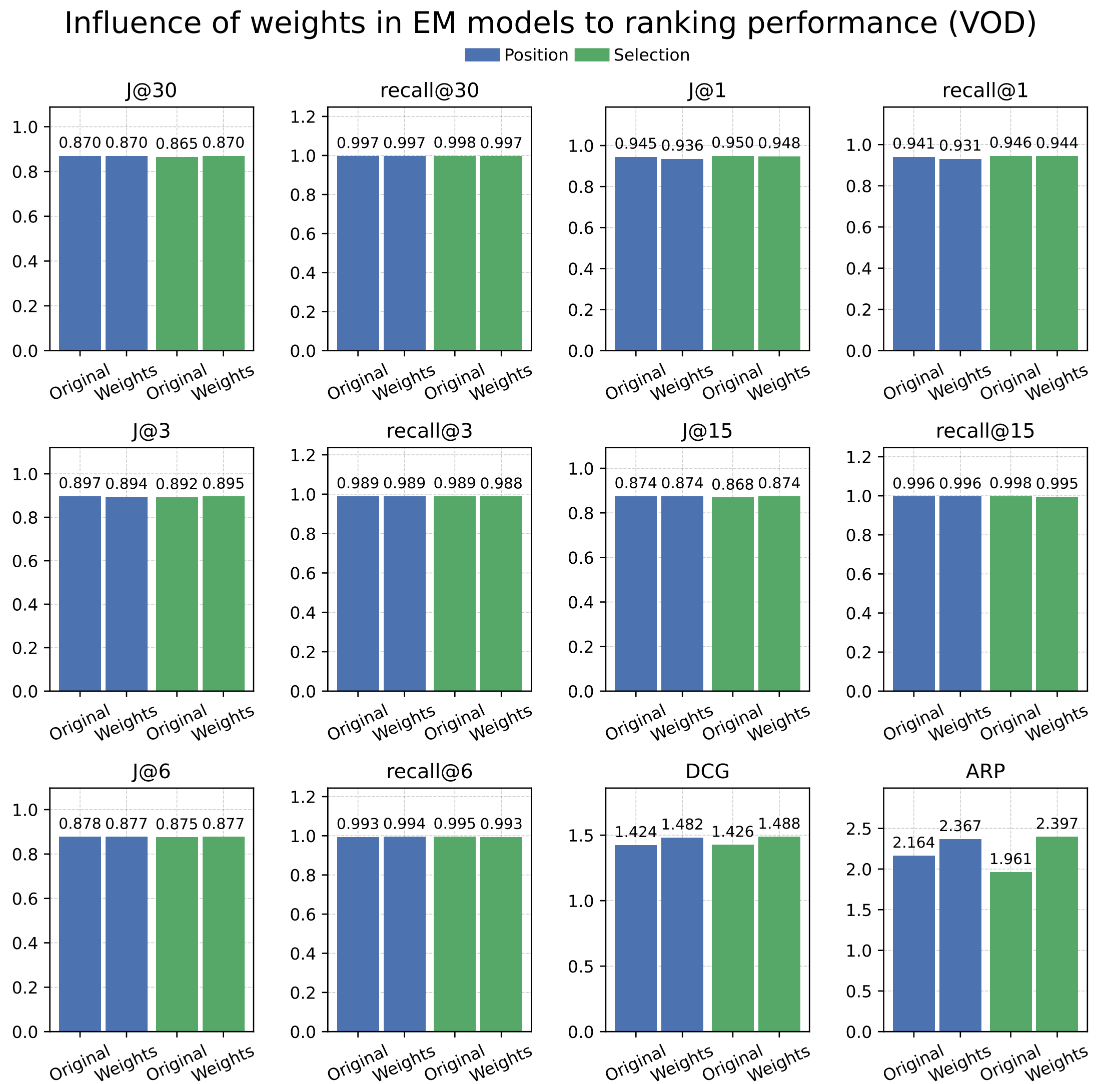

Figure 5.3 Comparison of the *EM* algorithms with the improvement on the ranking performance for the *VOD* platform

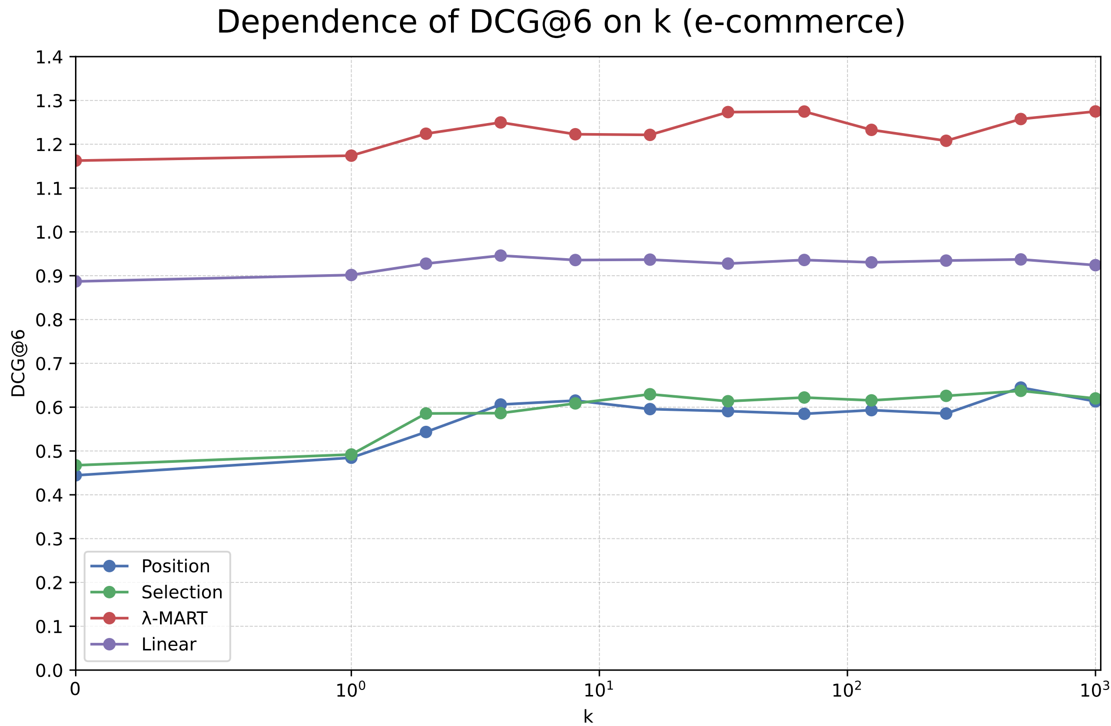

Figure 5.4 Dependence of *DCG*@6 on parameter k used in collaboration-filtering algorithms for *e-commerce* platform

for the values of the parameter k lower than 4. The deviations are larger compared to the other rankers.

The same graph is shown in Figure 5.5 for the *VOD* platform. There is an improvement for values of k less than 4. Again, there are some deviations. The improvement is lower compared to the *e-commerce* platform.

For both platforms, there is a correlation between the deviations in the performance of the models for a particular value of k. Models are trained independently, so the performance variance created by learning randomization should not be correlated. It indicates that there are values of parameter k greater than 4 that do not provide the same performance. That is, the features (created by the collaborative filtering algorithms that used the parameter k) do not explain the target variable the same for different values of k greater than 4. The ranking metric is not a non-decreasing function of the parameter k, which is expected since the larger the parameter k, the more information the features should contain.

In other experiments, we set the value of the parameter k at 1000 for the *e-commerce* platform and 250 for the *VOD* platform, as it provided the best performance among all models. Setting a higher value is not necessary since the performance of the models will be the same, but the computational resources will increase.

In Figures 5.6 and 5.9 are comparisons of the ranking performance of different personalization versions for different models. The *KNN* is a version in which only *Collaborative* features are used without user property features with values of k set to 1000 and 250 for the *e-commerce* and *VOD* platforms, respectively. The *Content* version is the opposite.

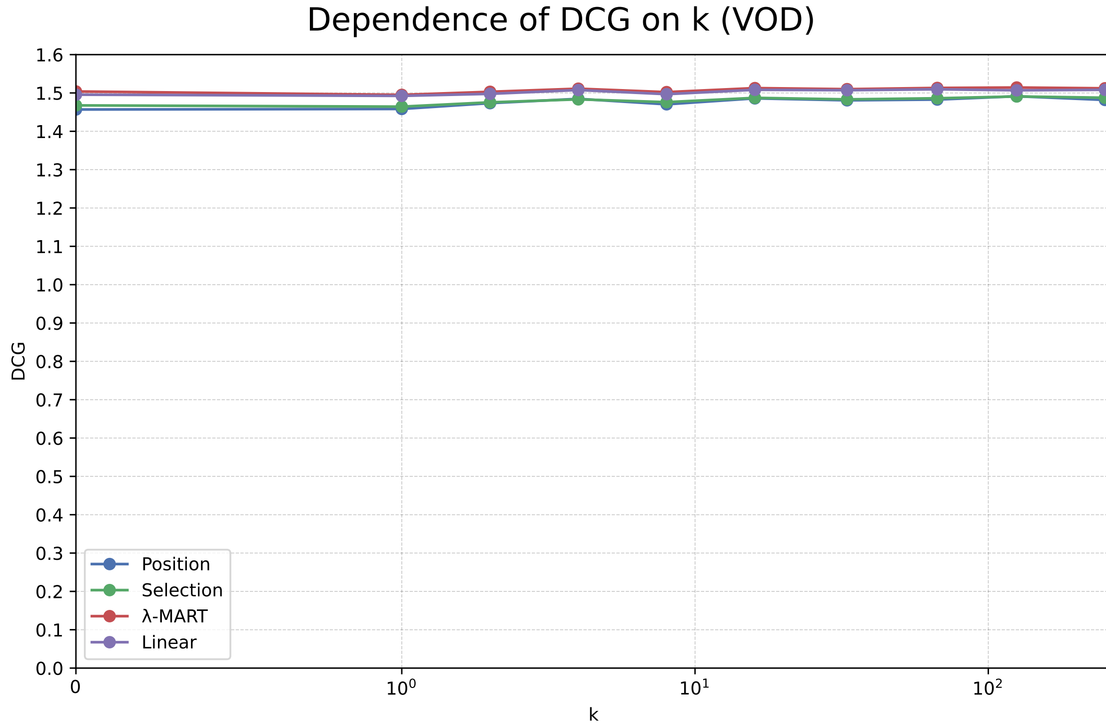

Figure 5.5 Dependence of *DCG* on parameter k used in collaboration-filtering algorithms for *VOD* platform

Only the *Content-based* features are used. Then the *Both* version uses both types of features. The *Without* version does not use the personalization features at all.

In Figure 5.6 is a comparison for the *e-commerce* platform. Using only user properties does not provide any additional performance to the *Without* version. An exception is the *ARP* metric for the *Selection* model, where the performance is better. It can be caused by the high variance of the model in this metric.

Using both types of features at the same time also does not provide additional performance to the *KNN* version.

The only features that significantly improve performance are the *Collaborative* features. The greatest improvement is for the *Selection* and *Position* models. The smallest improvement is for the *Linear* model, which indicates that there is a non-linear relationship between the ranking score and the *Collaborative* and other document features.

Since the *Selection* and *Position* models (presented in Section 3.4) are *GDBT*, we can show the importance of the features. In Figures 5.7 and 5.8 are the importance of the features created by the models for the *Both* version. The feature importances are divided by the maximum importance, and thus the maximum importance is 1. Features with importance lower than 3.4% after normalization are filtered out. Since we cannot give the exact name of the properties, we use the following naming convention. The document or user properties that we can present have the same name. The document text properties that we search in and create a score are numbered from zero (presented in Section 3.2) and then the name of the feature is *Search score* $\{NUMBER\}$. The user properties are also numbered from zero, and the name is *User feature* $\{NUMBER\}$. The

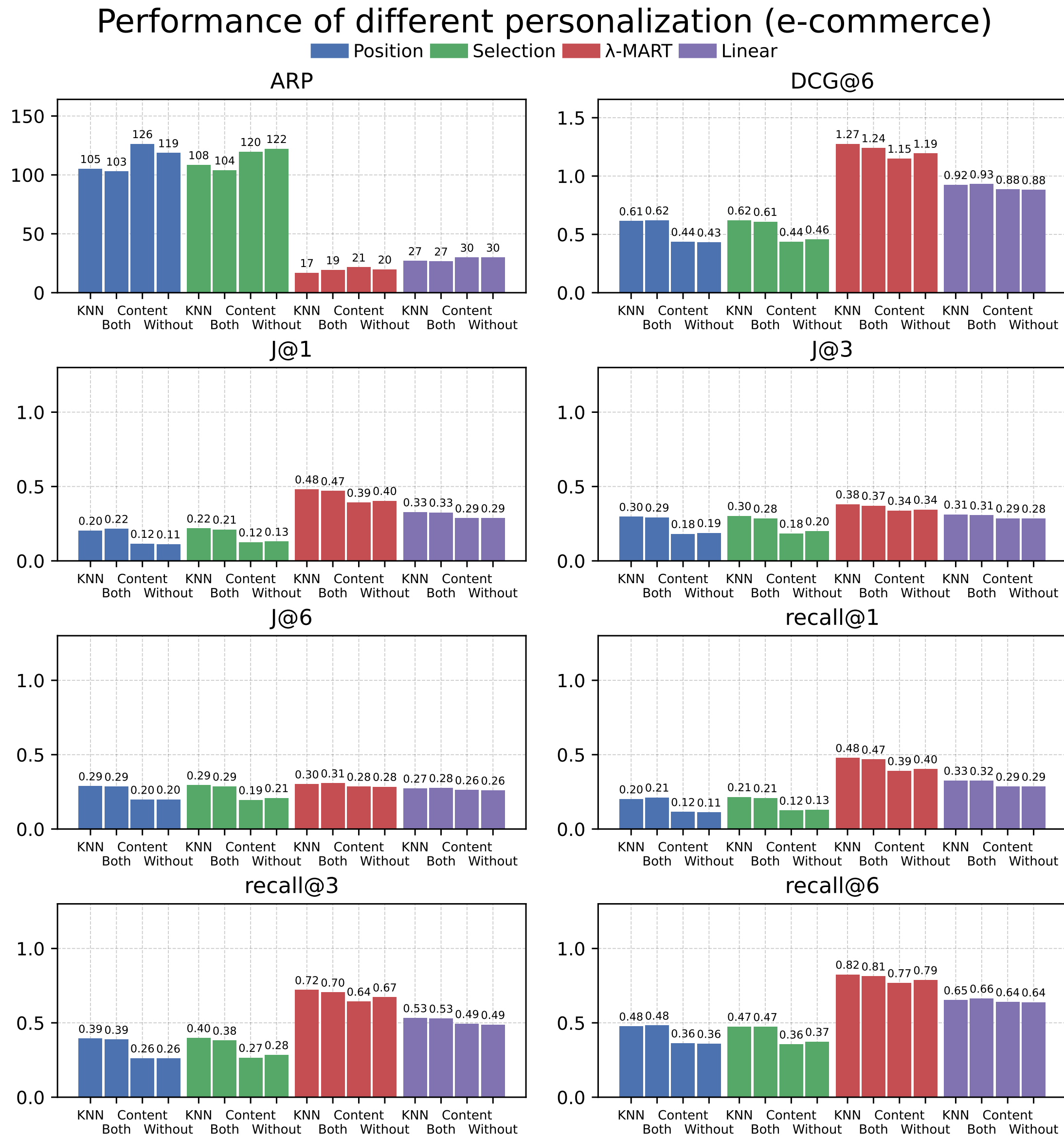

Figure 5.6 Comparison of different personalization approaches to the ranking performance for the *e-commerce* platform

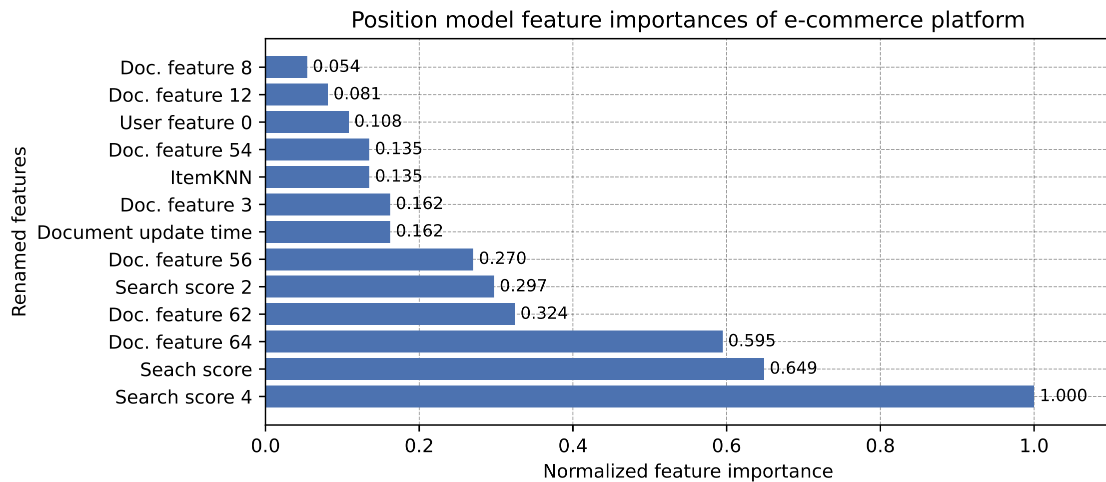

Figure 5.7 Feature importance of the *Position* model for the *e-commerce* platform

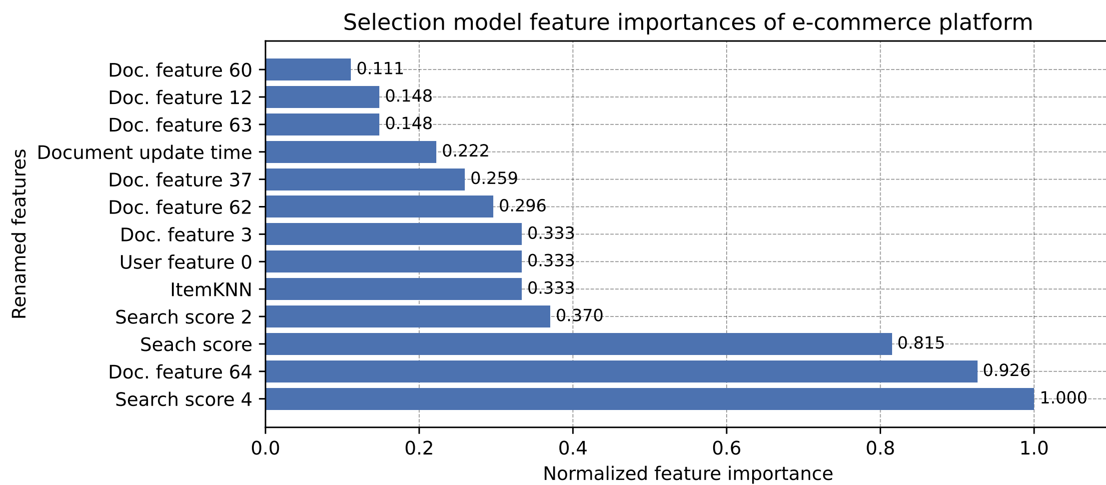

Figure 5.8 Feature importance of the *Selection* model for the *e-commerce* platform

other features of the documents are also numbered from zero, and the name is then *Doc. feature* $\{NUMBER\}$. The *Search score* without number is the global search score, collaborative filtering features are named *ItemKNN* and *UserKNN*.

The *Position* model in Figure 5.7 contains only one *Content-based* feature of importance 10.8%, the *ItemKNN* feature is of importance 13.5%. The *UserKNN* feature is filtered since the importance is lower than the threshold.

For the *Selection* model, the situation is slightly different. The importance of personalization features is greater. Both have the importance of 33.3%. There are several more features that are more important, but adding the *ItemKNN* feature significantly improved performance. Although the importance of the feature *User feature 0* is the same as for *ItemKNN* the ranking performance of the model did not change when the user properties were used alone.

In Figure 5.9 is again the ranking comparison of different versions of personalization for different models for the *VOD* platform. As we have seen, the performance improve-

ment is not as significant for the *e-commerce* platform. Using only user properties alone does not improve ranking metrics with the exception of *Position* model where each metric for *Content* is slightly better compared to *Without*. As we can see in Figure 5.10 there is no significant *Content-based* feature and the user data for the *e-commerce* platform contain only a single user feature. Based on this observation, the better performance is just a variance of the model caused by the randomization in learning.

Using both types of features at the same time also does not provide additional performance to the *KNN* version. Again, compared to all other versions, the greatest improvement is for the *KNN* version.

In Figures 5.10 and 5.11 are again the feature importances for *Position* and *Selection* models, respectively. Both the *ItemKNN* and *UserKNN* importances are above the threshold for both models, but there are still features with significantly higher importance.

In Figures 5.19 and 5.20 are the feature importances created by the *LambdaMART* model for the *e-commerce* and *VOD* platforms, respectively. For the *e-commerce* platform, *ItemKNN* is more important than *UserKNN*. A personalization feature that is even more important is *User update time*. There are also other user property features with numbers 4 and 33.

Figure 5.20 for the *VOD* platform shows that there is a *User update time* personalization feature that is the most important, and all other features are relatively much less significant. The second most important feature is *ItemKNN*.

The intuition why there are some *Content-based* features that are more important compared to *ItemKNN* or *UserKNN* but do not provide the same ranking performance improvement could be that the information in the user property features is also spread out in the document features. When user *Content-based* features are added, they do not provide any additional information needed to rank the documents better. Other features that are already present provide the information. When the *ItemKNN* or *UserKNN* features are added, the features contain more information and the target probability or score is better explained as a function of those features.

In general, the feature created by *ItemKNN* seems very important to the ranking performance. The feature created by *UserKNN* is much less important. Although some of the user property features are more important compared to *ItemKNN* from the model's perspective, they do not appear to be very important in the ranking performance. For parameter k greater than 4, the performance is very similar for all models that use the features. Although there appear to be some values of the parameter k greater than 4 that do not provide the same performance, the selection of the right parameter k is important for a concrete model. These observations hold for data from both platforms. The importance of the other features will also be discussed in the following sections.

5.3 Unbiased rankers experiments results

In Figure 5.12 is a comparison of rankers trained with different estimators. There are three versions of the *LambdaMART* model. The first one named *Biased* is the model trained using pure clicks without debiasing (the naive estimator is used). The second *Selection* is trained using debiasing with the policy-aware estimator. The last *Neg. biased* is trained using an estimator in which only the clicked documents are debiased,

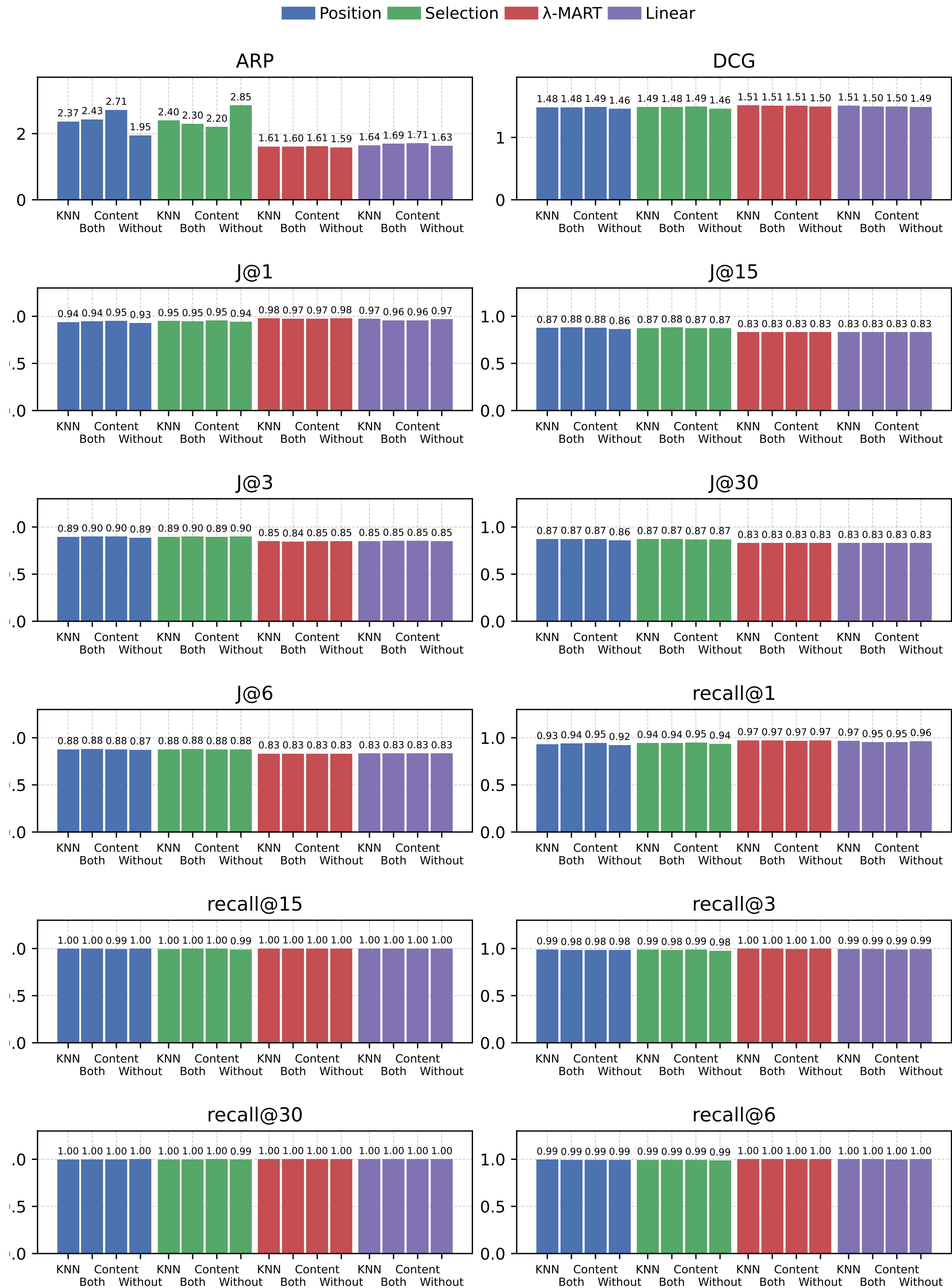

Figure 5.9 Comparison of different personalization approaches to the ranking performance for the *VOD* platform

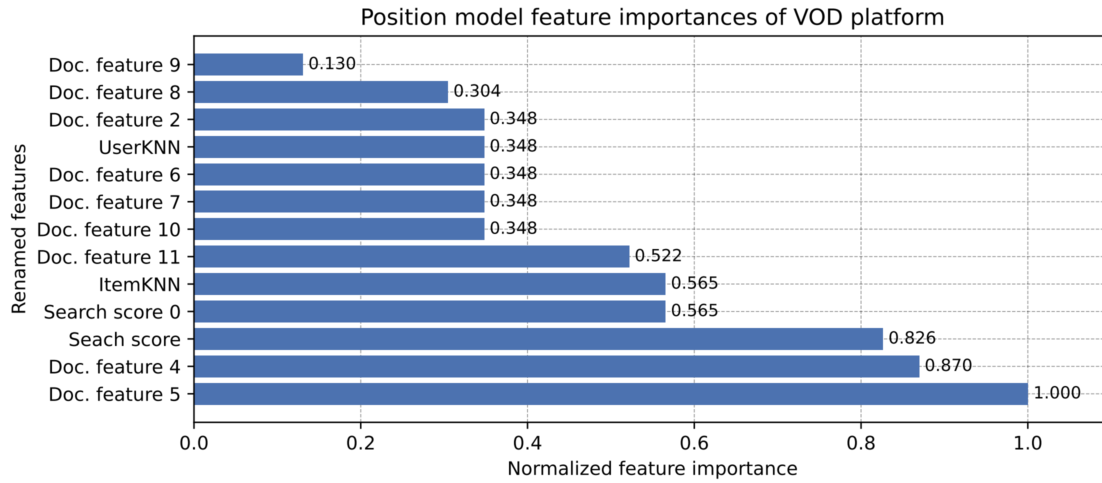

Figure 5.10 Feature importance of the *Position* model for the *VOD* platform

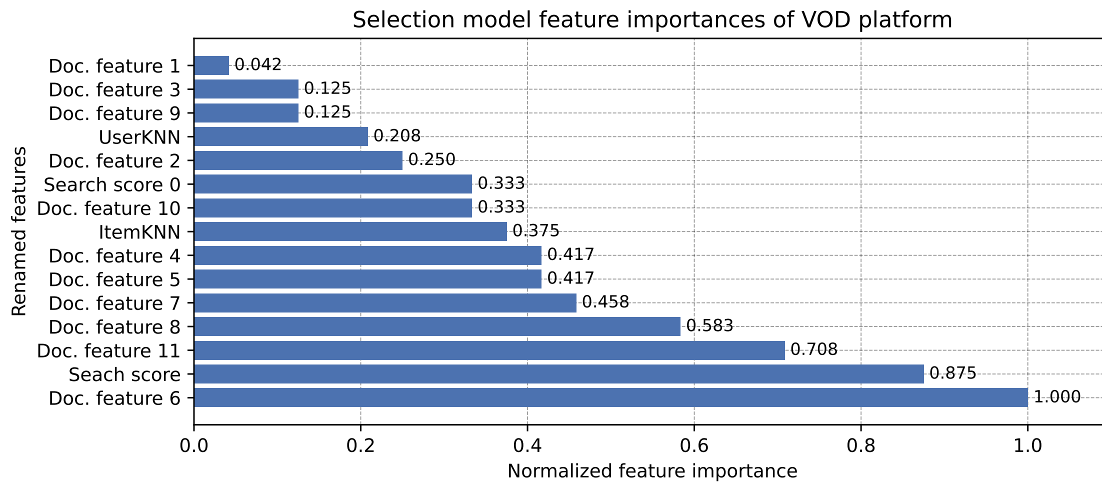

Figure 5.11 Feature importance of the *Selection* model for the *VOD* platform

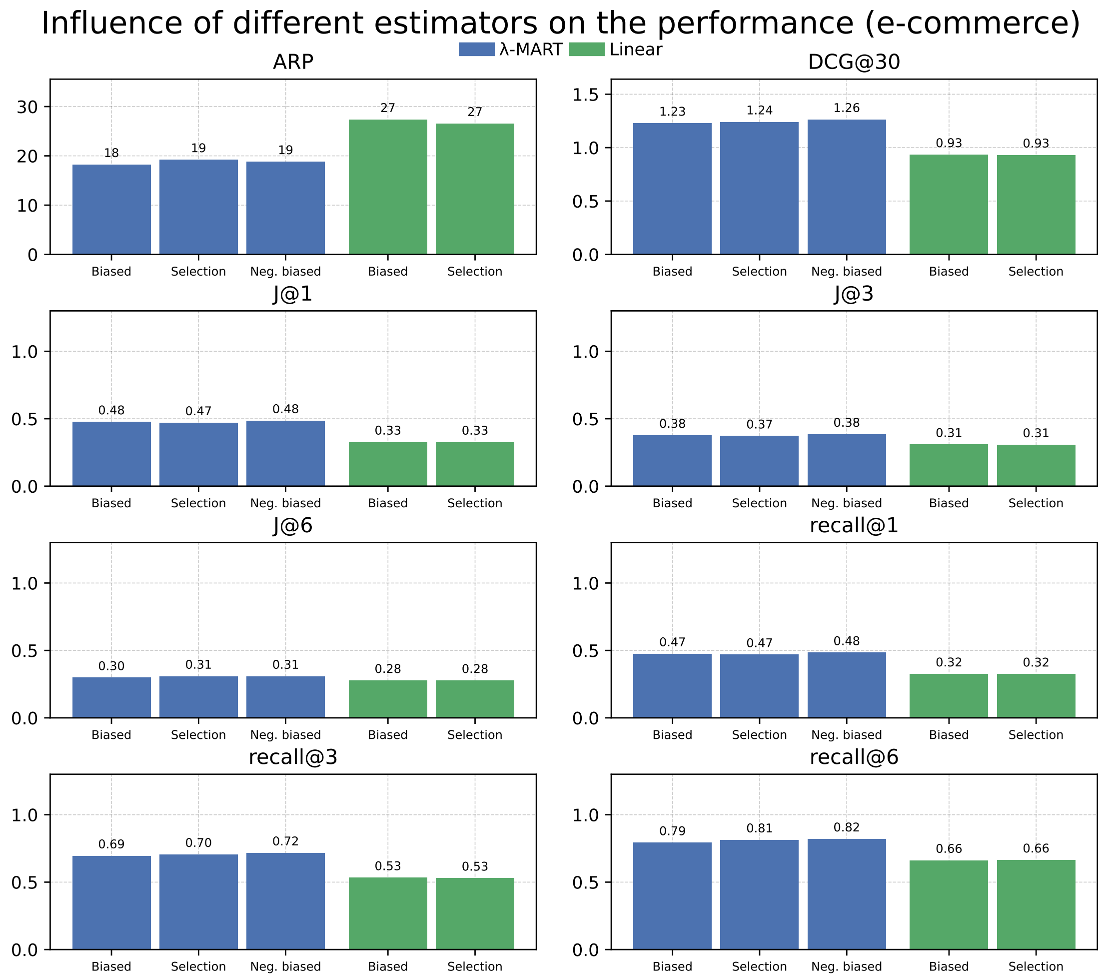

Figure 5.12 Comparison of different debiasing approaches on the ranking performance of the models for the *e-commerce* platform

as presented in Section 4.3. The *Linear* model is trained only using the naive estimator and the policy-aware estimator. The policy-oblivious estimator is omitted because the condition for using this estimator is not satisfied for this platform, as discussed in Section 3.3.

The performance of the models does not change much with different estimators for all metrics. For most metrics, the version *Neg. biased* is slightly better than the other versions. We estimated the probability in Equation 4.1 using the *Selection* model. The estimate may not be accurate enough and thus misleading. Omitting the estimate and replacing the expression with 1 seems to provide slightly better performance. There are other small differences, but they are most likely caused by randomization when fitting models. The reason why a biased ranker has the same performance as an unbiased ranker may be because training a biased ranker does not mean that the ranker is unable to learn important relationships compared to the unbiased ranker, as stated in [21].

Figure 5.13 is the same graph but for data from the *VOD* platform. Since the condition to use the policy-oblivious estimator is satisfied, there is also the *Position*

version for the *Linear* model for completeness. In this graph, there does not seem to be any pattern caused by the different versions of the models.

The different estimators used while learning the model do not have a significant influence on the performance of the model. Even a *Biased* version provides the same performance as the unbiased rankers. The only exception is *Neg. biased* version, which provides slightly better performance in some metrics, but only for the *e-commerce* platform.

5.4 Comparison of rankers results

A comparison of the learned model with the best-performing parameters is shown in Figure 5.14 for the *e-commerce* platform. There is the performance of the 7 rankers. The best model in almost all metrics is the *Legacy* model that was deployed in production while the data we used were collected. The *recall@6* is equal to 1 which is the optimal performance. The reason is that at most 6 documents were shown to the user, and the user could click only on these documents and no others.

The λ-*MART* model outperforms all other models in each metric. For the *recall@1* metric, λ-*MART* also outperforms the *Legacy* model, which means that the model is better at placing clicked documents at the first position. The performance of *recall@3* is also very competitive to that of *Legacy*.

The second best model that we trained is the *Linear* model in which we optimize an upper bound of the global *DCG*. The model is still better than the models primarily used to estimate the biases.

The *Position* and *Selection* models are the worst of all trained rankers in all metrics except the $J@3$ and $J@6$ metrics, where performance is comparable to other models.

The *Baseline* ranker provides a decent performance. All trained models are better in almost all metrics. An exception is the *ARP* metric, where the *Baseline* model is better than the *Position* and *Selection* models.

The *Random* model is presented here only for comparison. The performance of the *Random* model is very poor, so there is significant room for ranking performance improvement.

The same comparison for the *VOD* platform is shown in Figure 5.15. One big difference is that the *Legacy* model is outperformed by all trained models on most metrics. For the most important *DCG* metric, the *Legacy* model is outperformed.

The λ-*MART* model seems slightly better than the other trained models. One exception is the Jaccard index metrics.

The *Position* and *Selection* models provide a very similar performance to the *Linear* model. For the *J@k* metrics, bias estimation models are better and should therefore be preferred to the *Linear* model.

The *Baseline* model is again outperformed by trained models even in the *ARP* metric. The *Random* model has significantly better performance for the *DCG* metric compared to the *e-commerce* platform. This indicates that there is less space to reorganize the documents. It is expected since the *VOD* platform contains many fewer documents that can be recommended.

For completeness, Figure 5.16 evaluates the *DCG* and *ARP* metric using policy-oblivous estimator for the *VOD* platform. The relative differences between the models

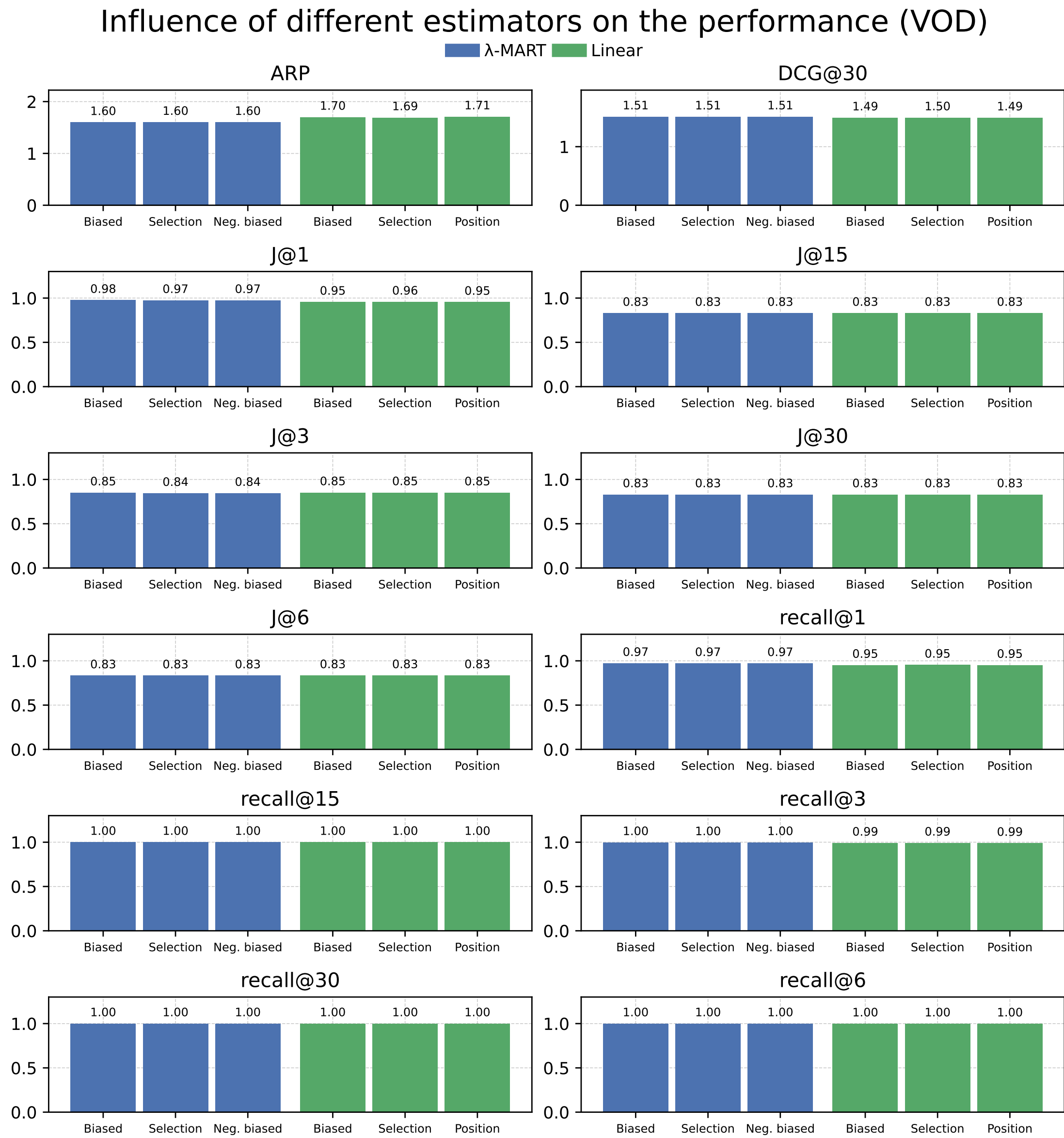

Figure 5.13 Comparison of different debiasing approaches on the ranking performance of the models for the *VOD* platform

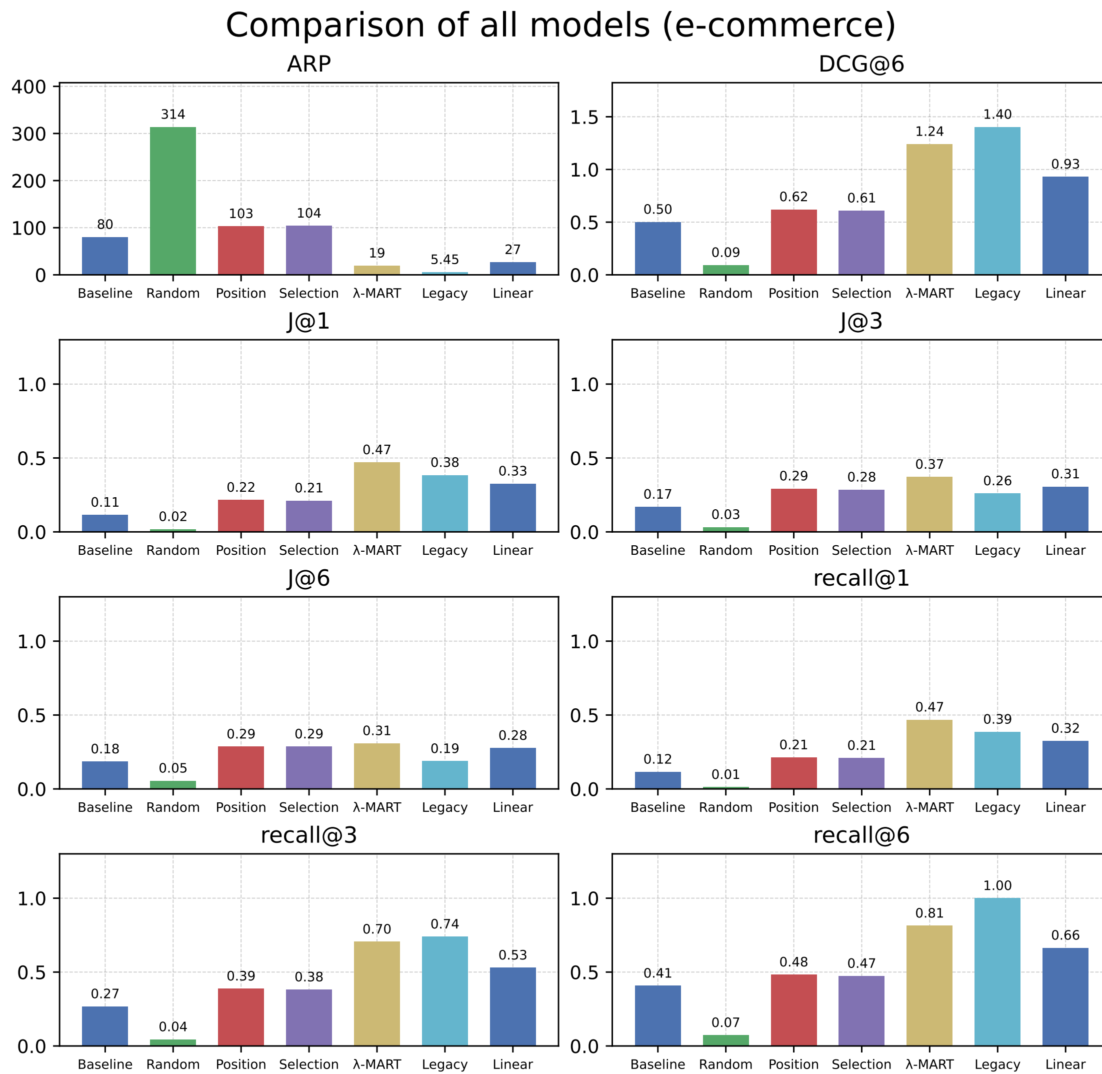

Figure 5.14 Comparison of different ranking approaches for the *e-commerce* platform

are very similar. The scale of the y-axis is different. The policy-oblivious estimator seems to give more weight to the metrics, and thus the scale is different. For this platform, the bias estimators should theoretically provide the same bias estimation. What is important is that the relative differences are preserved so the conclusion of which model is better stays the same.

The offline metrics are only indicators of the online performance of the model. For most metrics, the *Legacy* model is better than the other models, but this does not mean that the models would perform worse in production. Some preconditions may not be satisfied, the bias estimated may not be accurate enough, or the user model may be oversimplified, so it is not omniscient to compare the models to the model that was in production since the model is in a different position.

The reason why the performance of the bias estimation models is the worst among all the trained models could be that the models use the pointwise approach compared to the pairwise and listwise approaches, which generally have better ranking performance.

The performance of the bare search engine was beaten by all trained models, so it shows the advantage of using a Learning to Rank system.

Figure 5.17 denotes the dependence of $DCG@6$ on the learning phase. The algorithms that fit the models are iterative for all trained rankers. For each ranker, there is a different number of iterations. For each model after some fixed number of iterations, we measured the performance of the model. For each model, we measured the performance a different number of times, so we normalized the measurements into a learning phase in percentages using the least common multiple of the number of measurements. Each point in the graph is the performance measurement after some fixed number of training iterations. In the first row, there is the graph for training data, and the second row is the graph for test data.

The λ-*MART* model for iterations after the second measurement has the same ranking performance for the test data. The performance of the training data is still increasing with the number of iterations. The model is overfitting to the training data. It is not a problem as long as the performance on the test data does not decrease.

The curves for the *Linear* model are very similar, the model is not overfitted to the training data. The value of $DCG@6$ converged in the 6th measurement, there is no improvement in the next iterations.

The learning curves for both the *Position* and *Selection* models are mostly the same. The models have the best ranking performance from the perspective of $DCG@6$ in the first measurement. In further iterations, the performance decreases for both training and test data. In other experiments, we used the model from the third measurement, since it provided both good ranking performance and good log-likelihood of the data.

The same graph for the *VOD* platform is shown in Figure 5.18. The curves for the training and test data are the same. The λ-*MART* and *Linear* models converged to the same value of DCG. The *Position* and *Selection* models also have the same curves. For all models, the ranking performance is the same for all measurements after the second measurement.

There is no overfitting of the λ-*MART* model as for the data of the *e-commerce* platform. For the *Position* and *Selection* models, the performance does not decrease with the number of iterations.

In Figure 5.19 are the feature importances of the λ-*MART* model for the *e-commerce*

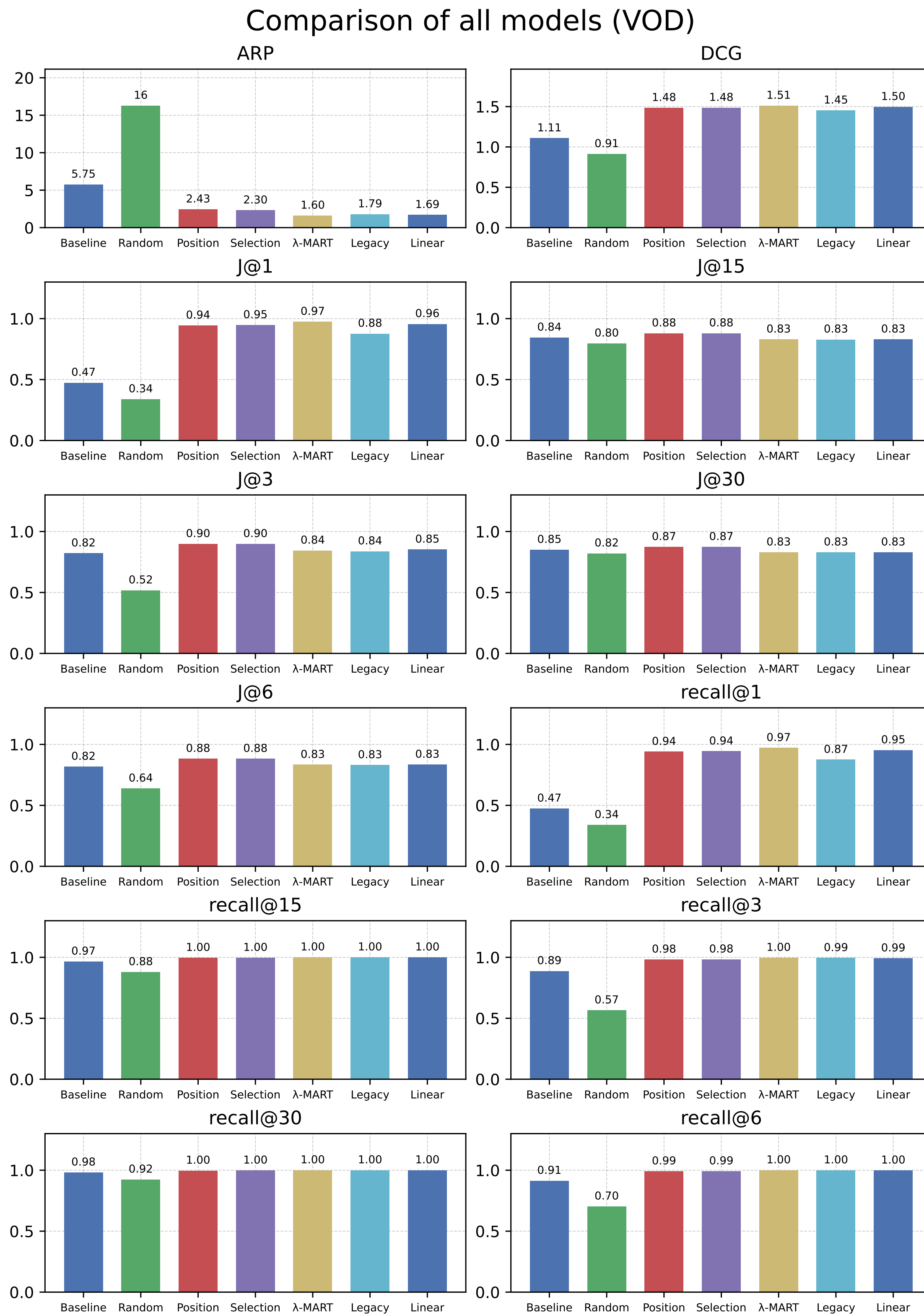

Figure 5.15 Comparison of different ranking approaches for the *VOD* platform

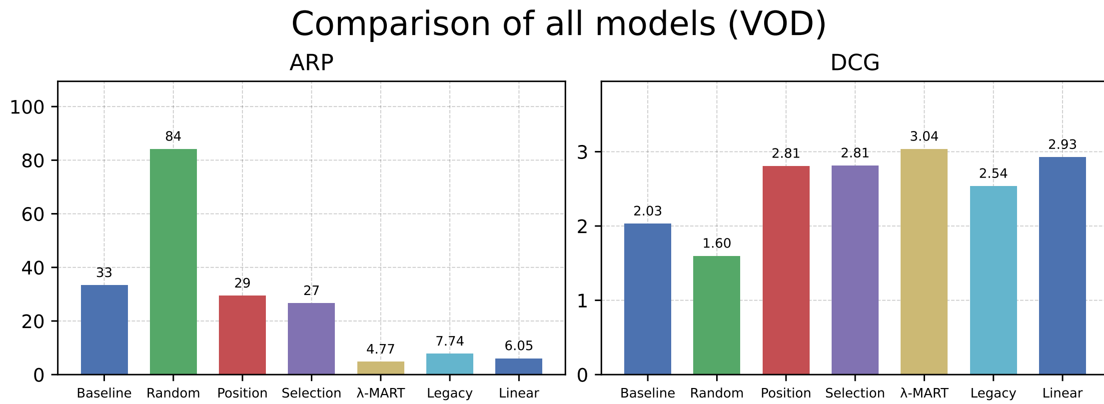

Figure 5.16 Comparison of different ranking approaches for the *VOD* platform with policy-oblivious estimator

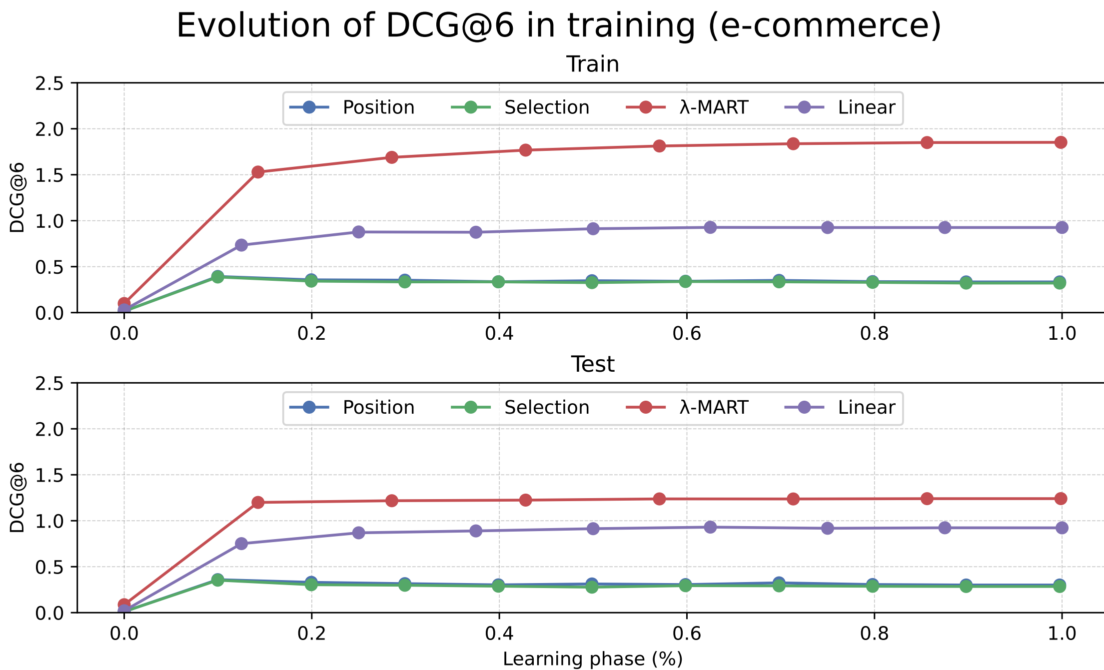

Figure 5.17 Dependence of *DCG*@6 on the number of iterations in the learning process for train and test data for the *e-commerce* platform

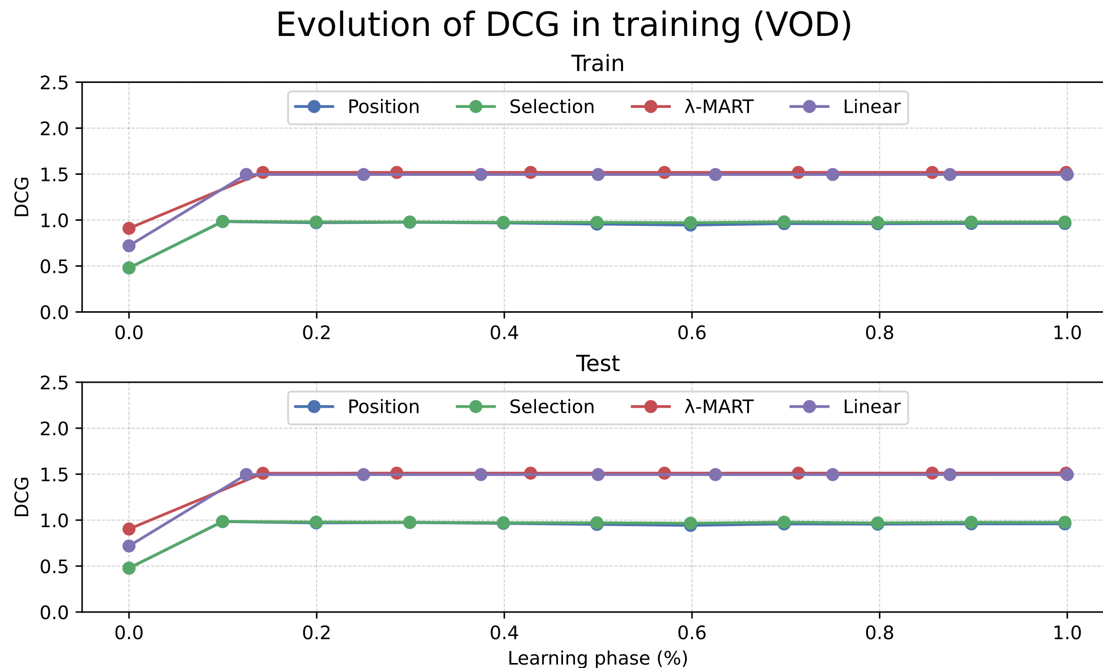

Figure 5.18 Dependence of *DCG* on the number of iterations in the learning process for training and test data for the *VOD* platform

platform. The graph structure was already presented in Section 5.2. The most important feature is *Document update time*. The update time can also be the time that the document was created. The cause may be that users click mainly on new or old documents in the platform, so it makes sense to recommend them in the first positions. The feature *User update time* is also very important for the model. The cause may be similar: new users are interested in specific types of documents. The global search score is the third most important feature. Search scores partitioned by the document text fields *Search score 4* and *Search score 2* are also very important.

A similar graph for the VOD platform is shown in Figure 5.20. By far, the most significant feature is *User update time*. The cause may be very similar to the *e-commerce* platform. The second most important feature is the *ItemKNN* personalization feature that slightly increased the ranking performance. Old or new users seem interested in a specific type of content. Search scores are slightly more important than most other document or user features.

In Figures 5.7 and 5.8 are feature importances of the bias models for the *e-commerce* platform. The feature importances between the *Selection* and *Position* models are similar, but they are very different from the *LambdaMART* feature importances. Some of the most important features for *LambdaMART* are not even present for the other two models, such as: *Doc. feature 49*, *Search score 4*, or *User update time*. This also holds on from the other side. The most important features for bias estimation models are *Document feature 64*, *Document feature 62*, or *User feature 0*. The bias estimation models have fewer features above the threshold. Similar observations hold for the *VOD* platform in Figures 5.10 and 5.11, but they use the same number of features.

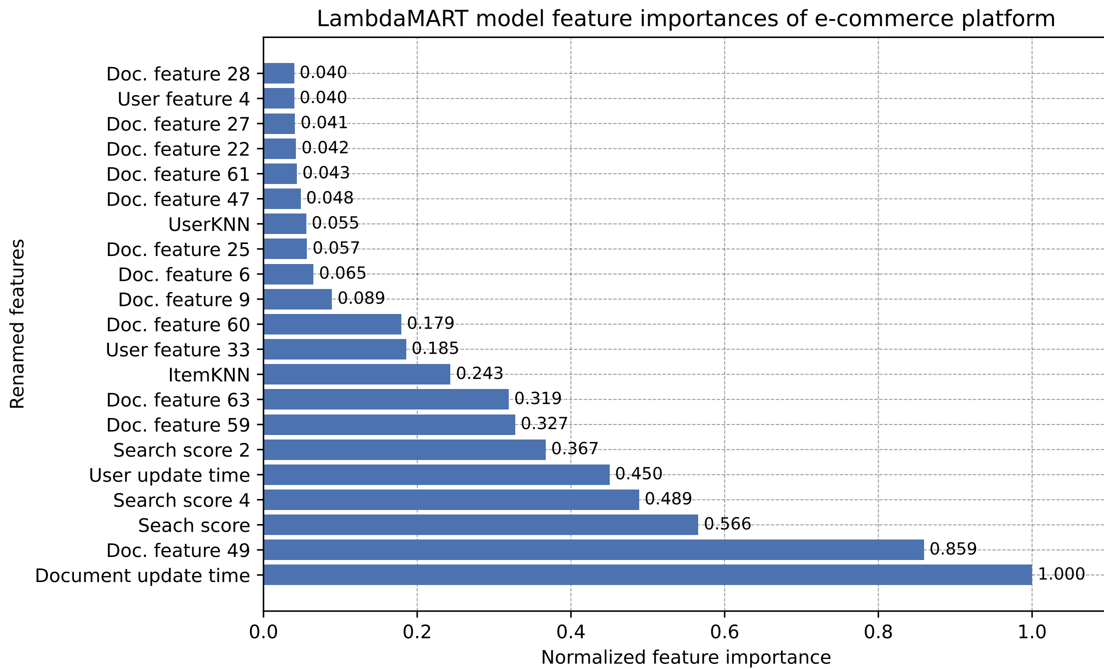

Figure 5.19 LambdaMART feature importances of *e-commerce* platform

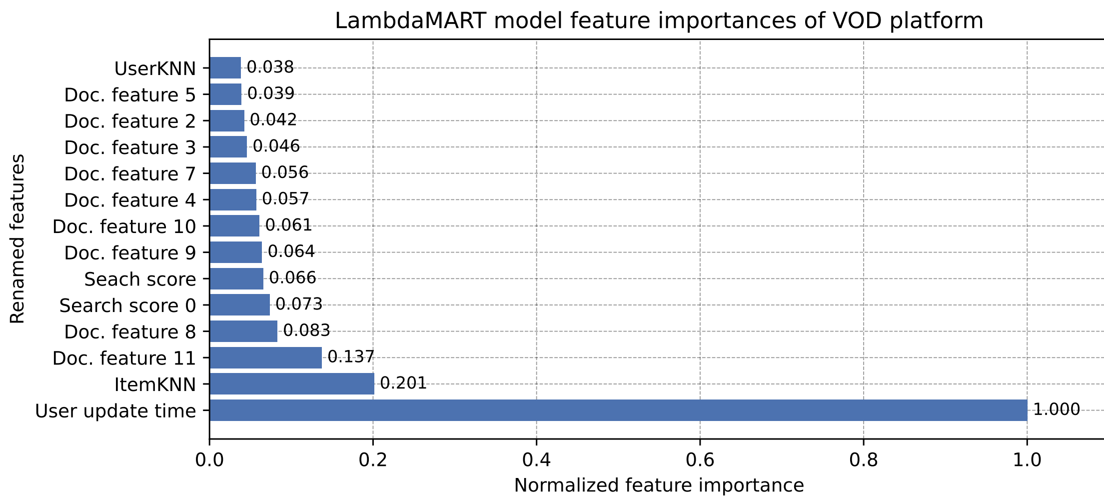

Figure 5.20 LambdaMART feature importances of *VOD* platform

Chapter 6

Conclusion

In this chapter, we discuss and conclude the findings of the previous chapter. In the end, we present the future work.

In the first experiment, we showed that adding the probabilities as weights to the model improves the convergence of the data log-likelihood. The weights also helped to improve the ranking performance from the perspective of the metrics used. An exception is the *ARP* metric which was worse than the original version. There was an improvement in the *DCG* metrics, which added more weight to the lower positions compared to the *DCG* metrics. The cause is that the models with the weights improvement rank better relevant documents in the top positions, but also some other relevant documents are moved deeper in the ranking.

In the second experiment, we tried different personalization methods. One approach used the properties of users. The other approach used created features using collaborative filtering algorithms. We showed that using the user properties did not improve the ranking performance, although the features had significant importance in the ranking algorithms. The cause may be that the information in the user properties is already present in the document features. For collaborative filtering features, we tried the parameter k that controls the number of neighbors to use. We found that using more than 4 neighbors does not provide a significantly better performance, although some values of the parameter negatively impact the performance of all models. The personalization features created by the collaborative filtering algorithms significantly improved ranking performance. The feature created by *ItemKNN* had considerably greater importance than the *UserKNN* feature. When both personalization approaches were used together, there was no performance improvement compared to the case where collaborative filtering features were used alone.

The third experiment showed that training an unbiased ranker does not significantly improve ranking performance compared to training a biased ranker. A slight improvement was for the version of the *LambdaMART* model, where we only estimated the bias of the clicked document. The estimation of the bias of a non-clicked document was probably misleading and thus decreased the performance of the completely unbiased ranker. The cause of such a minor improvement may be that learning a biased ranker does not mean that the ranker cannot learn the correct relationships compared to an unbiased ranker.

We compared the models and discussed the created features in the last experiment. From the models we trained, *LambdaMART* provided the best ranking performance among most metrics. This model also outperformed the previously deployed ranker for the *e-commerce* platform in the $recall@1$ metric. The performance of the $recall@3$ metric was also very competitive. For the *VOD* platform, all trained models outperformed the previously deployed model in all metrics. For the *e-commerce* platform, the *Linear* model outperformed the *Position* and *Selection* models. For the *VOD* platform, the ranking performance for these models was very similar, but the bias estimation models had better results from the perspective of Jaccard index metrics. All trained models surpassed the performance of the bare search engine, demonstrating the benefits of employing a Learning to Rank system. We showed using the feature importances of the models that the designed features are essential for the ranker as well as for the bias estimators. Both the personalization and search features are of relatively high importance compared to most other document features.

6.1 Future work

The next step that can be done with the results of this thesis is to deploy the best-performing models to the production of the platforms and evaluate the performance in the online environment.

Bibliography

1. ROBERTSON, Stephen; ZARAGOZA, Hugo. The Probabilistic Relevance Framework: BM25 and Beyond. *Found. Trends Inf. Retr.* 2009, vol. 3, no. 4, pp. 333–389. ISSN 1554-0669. Available from DOI: 10.1561/1500000019.
2. LIU, Tie-Yan. *Learning to Rank for Information Retrieval.* Vol. 3. Now Publishers, 2007. No. 3. Available from DOI: 10.1561/1500000016.
3. OOSTERHUIS, Harrie; JAGERMAN, Robin; RIJKE, Maarten de. Unbiased Learning to Rank: Counterfactual and Online Approaches. In: *Companion Proceedings of the Web Conference 2020.* ACM, 2020.
4. OOSTERHUIS, Harrie; RIJKE, Maarten de. Policy-Aware Unbiased Learning to Rank for Top-k Rankings. In: *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval.* ACM, 2020.
5. REHOREK, Tomas; BIZA, Ondrej; BARTYZAL, Radek; KORDIK, Pavel; POVALYEV, Ivan; PODSTAVEK, O. Comparing offline and online evaluation results of recommender systems. In: *In Proceedings of the REVEAL workshop at RecSyS conference (RecSyS 2018).* 2018.
6. WANG, Xuanhui; GOLBANDI, Nadav; BENDERSKY, Michael; METZLER, Donald; NAJORK, Marc. Position Bias Estimation for Unbiased Learning to Rank in Personal Search. In: *Proceedings of the 11th ACM International Conference on Web Search and Data Mining (WSDM).* 2018, pp. 610–618.
7. CHUKLIN, Aleksandr; MARKOV, Ilya; RIJKE, Maarten de. *Click Models for Web Search.* Morgan & Claypool, 2015. ISBN 9781627056489. Available from DOI: 10.2200/S00654ED1V01Y201507ICR043.
8. JOACHIMS, Thorsten; SWAMINATHAN, Adith; SCHNABEL, Tobias. *Unbiased Learning-to-Rank with Biased Feedback.* arXiv, 2016. Available from DOI: 10.48550/ARXIV.1608.04468.
9. HU, Ziniu; WANG, Yang; PENG, Qu; LI, Hang. *Unbiased LambdaMART: An Unbiased Pairwise Learning-to-Rank Algorithm.* 2019. Available from arXiv: 1809.05818 [cs.IR].

10. DEMPSTER, A.P.; LAIRD, N.M.; RUBIN, D.B. Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)* [online]. 1977, vol. 39, no. 1, pp. 1–38 [visited on 2023-03-22]. ISSN 00359246. Available from: `http://www.jstor.org/stable/2984875`.
11. SAMMUT, Claude; WEBB, Geoffrey I. (eds.). Collaborative Filtering. In: *Encyclopedia of Machine Learning*. Boston, MA: Springer US, 2010, pp. 189–189. ISBN 978-0-387-30164-8. Available from DOI: `10.1007/978-0-387-30164-8_138`.
12. BURGES, Christopher; RAGNO, Robert; LE, Quoc. Learning to Rank with Nonsmooth Cost Functions. In: SCHÖLKOPF, B.; PLATT, J.; HOFFMAN, T. (eds.). *Advances in Neural Information Processing Systems*. MIT Press, 2006, vol. 19. Available also from: `https://proceedings.neurips.cc/paper_files/paper/2006/file/af44c4c56f385c43f2529f9b1b018f6a-Paper.pdf`.
13. BURGES, Chris; SHAKED, Tal; RENSHAW, Erin; LAZIER, Ari; DEEDS, Matt; HAMILTON, Nicole; HULLENDER, Greg. Learning to Rank Using Gradient Descent. In: *Proceedings of the 22nd International Conference on Machine Learning*. Bonn, Germany: Association for Computing Machinery, 2005, pp. 89–96. ICML '05. ISBN 1595931805. Available from DOI: `10.1145/1102351.1102363`.
14. DONMEZ, Pinar; SVORE, Krysta; BURGES, Christopher. On the Local Optimality of LambdaRank. In: 2009, pp. 460–467. Available from DOI: `10.1145/1571941.1572021`.
15. YUE, Yisong; BURGES, Chris J.C. *On Using Simultaneous Perturbation Stochastic Approximation for Learning to Rank, and the Empirical Optimality of LambdaRank*. 2007-04. Tech. rep., MSR-TR-2007-115. Available also from: `https://www.microsoft.com/en-us/research/publication/on-using-simultaneous-perturbation-stochastic-approximation-for-learning-to-rank-and-the-empirical-optimality-of-lambdarank/`.
16. WANG, Xuanhui; LI, Cheng; GOLBANDI, Nadav; BENDERSKY, Michael; NAJORK, Marc. The LambdaLoss Framework for Ranking Metric Optimization. In: 2018, pp. 1313–1322. Available from DOI: `10.1145/3269206.3271784`.
17. BURGES, Christopher. From ranknet to lambdarank to lambdamart: An overview. *Learning*. 2010, vol. 11, no. 23-581, p. 81.
18. KE, Guolin; MENG, Qi; FINLEY, Thomas; WANG, Taifeng; CHEN, Wei; MA, Weidong; YE, Qiwei; LIU, Tie-Yan. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 3149–3157. NIPS'17. ISBN 9781510860964.
19. KLOUDA, Karel; PABLO MALDONADO LOPEZ, Juan; VAŠATA, Daniel. Lineární regrese - pokračování, Hřebenová regrese [presentation]. In: *FIT ČVUT Courses* [online]. Praha: ČVUT FIT v Praze, 2020 [visited on 2023-04-18]. Available from: `https://courses.fit.cvut.cz/BI-VZD/@B201/lectures/files/BI-VZD-07-cs-lecture.pdf`.

20. CARTERETTE, Ben; CHANDAR, Praveen. Offline Comparative Evaluation with Incremental, Minimally-Invasive Online Feedback. In: *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*. Ann Arbor, MI, USA: Association for Computing Machinery, 2018, pp. 705–714. SIGIR '18. ISBN 9781450356572. Available from DOI: 10.1145/3209978.3210050.

21. YANG, Tao; LUO, Chen; LU, Hanqing; GUPTA, Parth; YIN, Bing; AI, Qingyao. Can Clicks Be Both Labels and Features? Unbiased Behavior Feature Collection and Uncertainty-Aware Learning to Rank. In: *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*. Madrid, Spain: Association for Computing Machinery, 2022, pp. 6–17. SIGIR '22. ISBN 9781450387323. Available from DOI: 10.1145/3477495.3531948.

Content of the enclosed media

- `readme.txt` .... brief description of the media content
- `thesis.pdf` .... thesis text in PDF format
- `requirements.txt` .... the python packages that were used
- `Dockerfile` .... environment in which the experiments were conducted
- `src` .... directory of the source code files
  - `thesis` .... thesis source code in LaTeX format and bibliography file
  - `impl` .... python source code files used to conduct the experiments
    - `LightGBM` .... the LGBM framework with the updated code