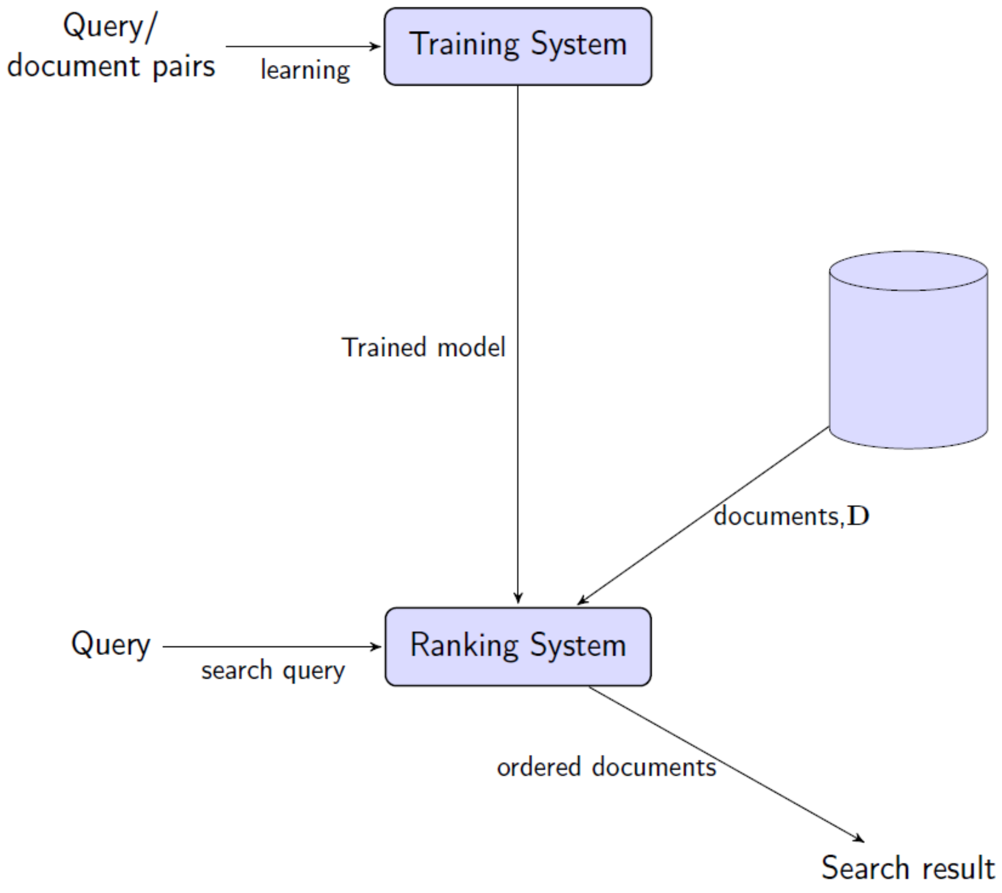
CHALMERS



Learning to rank, a supervised approach for ranking of documents

Master Thesis in Computer Science - Algorithms, Languages and Logic

KRISTOFER TAPPER

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering

Göteborg, Sweden, June 2015

Learning to Rank, a supervised approach for ranking of documents.

KRISTOFER TAPPER

© KRISTOFER TAPPER, June 2015.

Examiner: DEVDATT DUBHASHI
Supervisor: FREDRIK JOHANSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2015

Abstract

As available information gets more accessible everywhere and as the rate of new information grows very fast, the systems and models which retrieve this information deserves more attention. The purpose of this thesis is to investigate state-of-the-art machine learning methods for ranking known as learning to rank. The goal is to explore if learning to rank can be used in enterprise search, which means less data and less document features than web based search. Comparisons between several state-of-the-art algorithms from RankLib (Dang, 2011) was carried out on benchmark datasets. Further, Fidelity Loss Ranking (Tsai et al., 2007) was implemented and added to RankLib. The performance of the tests showed that the machine learning algorithms in RankLib had similar performance and that the size of the training sets and the number of features were crucial. Learning to rank is a possible alternative to the standard ranking models in enterprise search only if there are enough features and enough training data. Advise for an implementation of learning to rank in Apache Solr is given, which can be useful for future development. Such an implementation requires a lot of understanding about how the Lucene core works on a low level. **Keywords:** ranking, learning to rank, information retrieval, machine learning

Acknowledgements

I want to express my gratitude for all the support and helpful discussions from my advisor Per Ringqvist at Findwise and my supervisor Fredrik Johansson at Chalmers. Further I want to thank all the people at Findwise Gothenburg for great inputs, encouragement and for letting me do my thesis at their office. My examiner Devdatt Dubhashi also deserves a mention for his valuable insights.

Kristofer Tapper
Göteborg, Thursday 11th June, 2015

Contents

Chapter 1

Introduction

1.1 Background

As available information increases everywhere in a very fast pace, the systems and models which retrieves this information demands more attention. This constant growing stream of new information has both positive and negative aspects. A positive aspect is that the user can access information previously not available, information that earlier would have taken long time to access. Compare for example searching for information in a traditional library with searching for information in a digital online library, the latter takes considerably shorter time. But there are also negative aspects with too much information, often referred to as information overload. The term was coined in 1985 by Hiltz and Turoff (1985) and states that more information not necessarily is better. The increasing amount of information, not only on the Internet but everywhere, requires an increased precision in the process of retrieving this information. It puts more pressure on the actual algorithms that is retrieving the information to deliver relevant results. When there is more information one can also assume that there is more data that is irrelevant to the user. The term Information Retrieval is defined in *Introduction to Information Retrieval* by Manning et al. (2008) as:

"Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers)."

This large collection of information (often documents) should be ordered and ranked to satisfy the users demands. A search engine for example orders the search results based on the users input query. One complication is that the generated search result may not agree with what the user wants. Two users asking the same query may not be equally happy with the same search results, this subjectivity makes the task of ranking the search results particularly hard. The goal with a good ranking system is to improve the user satisfaction by generating relevant information.

The aim of this thesis is to investigate machine learning methods for ranking, often referred to as *learning to rank*. Learning to rank has been a hot topic in recent years and a lot of progress has been made in the field of information retrieval using different machine learning algorithms (Li, 2014).

This thesis is conducted in collaboration with Findwise Gothenburg and their motivation for this thesis is that their customers not always are fully satisfied with the implementation of search solutions. By researching learning to rank, Findwise wants to be on the front line together with the research community in information retrieval. Usually when the search results not are satisfying a number of parameters in the search software can be fine tuned to increase the relevance, but this is a rather complex and time consuming task. With learning to rank this time could be used for more meaningful work.

1.2 Purpose

Can machine learning improve standard ranking models by using learning to rank? The standard ranking models are: the BM25 algorithm (Robertson and Zaragoza, 2009), the Language Model for Information Retrieval (LMIR)(Robertson and Zaragoza, 2009) and the Vector Space Model (VSM) (Melucci, 2009). The hypothesis is that standard ranking models can be improved with learning to rank. When this knowledge is established the project advances to an implementation part where one of the researched learning to rank algorithms gets implemented. Further, an implementation of a learning to rank algorithm can expose possible weaknesses that can be eliminated from a standard ranking model. Advantages as well as disadvantages of learning to rank algorithms can be pointed out and guidelines on how to successfully implement a effective ranking model can hopefully be established.

There is a gap in the research as learning to rank is tested only in huge web based settings. By testing learning to rank with smaller datasets this gap can be filled and a new possible area for learning to rank could be identified: enterprise search. Enterprise search usually has smaller amount of data than web based search. By investigating if learning to rank works good on smaller datasets this thesis finds out if it is possible to get relevant search results in enterprise search using learning to rank. Furthermore, the number of features that can be extracted from the search software is limited. Often a large number of features is needed and this thesis will try to discover if the ranking turns bad when the number of features is limited.

1.3 Limitations

This thesis investigates already implemented methods with proven functionality and is not trying to reinvent the wheel. The methods are the top performers from different benchmark tests published in various research journals and books in the field of information retrieval.

1.4 Problem Specification

This master thesis project will give answers to the following questions:

- Is enterprise search a suitable area for learning to rank?
- Can learning to rank compete with standard search models in enterprise search?
- Is it possible to implement learning to rank in Apache Solr[1]?

1.5 Methods

The early stage of the project consist of an information gathering phase to get deeper knowledge of different approaches. This stage of the project tries to localize advantages (and disadvantages) of several different learning to rank algorithms. The information gathering focuses on recent research and algorithms with a proven foundation based on machine learning. This overall knowledge is valuable to be able to make correct decisions about which algorithms that are suitable for implementation. The implementation part helps to find important things to keep in mind when implementing a custom ranking model. To make the gathering part of the project successful it is essential to get knowledge about which specific algorithms that works particularly good combined with certain datasets. Potential datasets have been localized such as the Microsoft Learning to Rank Datasets from Microsoft

[1]`http://lucene.apache.org/solr/`

Research[2] and the Yahoo! Learning to Rank Challenge dataset (Chapelle et al., 2011). These datasets are rather big and by shrinking these sets stepwise and limit the numbers of features experiments tries to localize a suitable algorithm to implement on a "real" dataset. The search server software which is planned to be used in this project is Apache Solr. Solr is a open-source search server based on Apache Lucene[3], a popular search engine built entirely in Java.

[2]`http://research.microsoft.com/en-us/projects/mslr/`

[3]`http://lucene.apache.org/core/`

Chapter 2

Theory

This chapter is meant to give the reader an in depth explanation of how a number of common ranking models are implemented. This chapter will also get the reader familiar with learning to rank. More specifically, the three most common learning to rank approaches will be explained and strengths and weaknesses in each approach will be pointed out. Two terms essential to know as preliminaries when approaching literature about information retrieval are precision and recall. Precision is defined as the number of relevant documents retrieved out of the total number of documents retrieved for a query. Recall is the number of relevant documents retrieved out of the total number of relevant documents in a collection (for a specific query) (Carterette, 2009). Precision and recall are general performance measures of the information retrieval system (i.e. the search engine) as they measure the number of relevant results. To introduce the reader to information retrieval the theory section will begin with an example: the famous *PageRank* algorithm which is used in the worlds most common web search engine Google, to rank websites based on popularity (Langville and Meyer, 2006).

2.1 PageRank

The PageRank algorithm was developed by the founders of Google[1] Sergey Brin and Larry Page, the purpose of the algorithm is to rank web pages based on popularity. Brin and Page realised that web pages could be represented as graphs, with nodes as pages and edges as links between the pages. Each web node has two types of directed edges, in links and out links. If for example page A has a link to page B this is represented in the graph as a directed edge from node A to node B. A site with many in links is a popular site and

[1] `http://www.google.com/about/company/history/`

therefore the site should receive a high pagerank score. The score for each site in the graph is calculated iteratively and an in link from a popular site will contribute with a higher score than a site with only a few in links. The pagerank score $P(S_i)$ for website S_i is calculated with the following equation (where n is the current iteration):

$$P_{n+1}(S_i) = \sum_{S_j \in M(S_i)} \frac{P_n(S_j)}{L(S_j)}$$

where $M(S_i)$ is the set of nodes which have out links to S_i and $L(S_j)$ is the number of out links from node S_j. Initially all nodes gets the same rank score $S_i = 1/m$ where m is the total number of nodes. The algorithm iterates until a stable state has been reached and thereafter the web pages are ordered by rank score.

PageRank Example

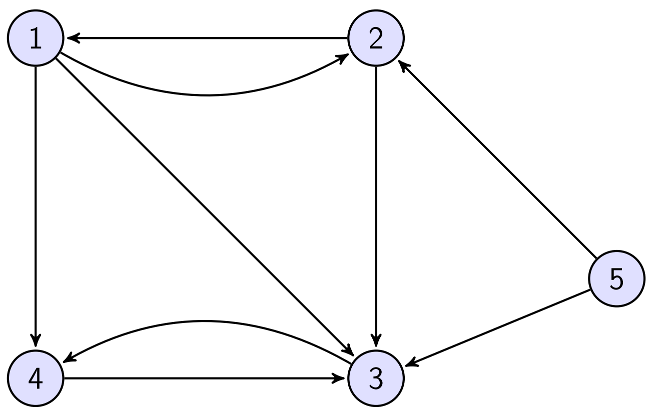


Figure 2.1: Small PageRank example

All nodes gets initialized with rank score $S_0 = 1/5$.

Node	Iteration 0	Iteration 1	Iteration 2	Rank after Iteration 2
1	1/5	1/10	1/12	3
2	1/5	5/30	1/30	4
3	1/5	13/30	23/60	2
4	1/5	4/15	14/30	1
5	1/5	0	0	5

Table 2.1: PageRank score for five nodes after two iterations.

2.2 The ranking model

The ranking problem is only explained for *document retrieval* but it still occurs in similar forms in several nearby areas such as *collaborative filtering* and *multi-document summarization.* The goal is to retrieve the most relevant documents ranked in order of relevance, based on the query q. The documents are retrieved from the collection D. The ranking model is pictured in figure 2.2.

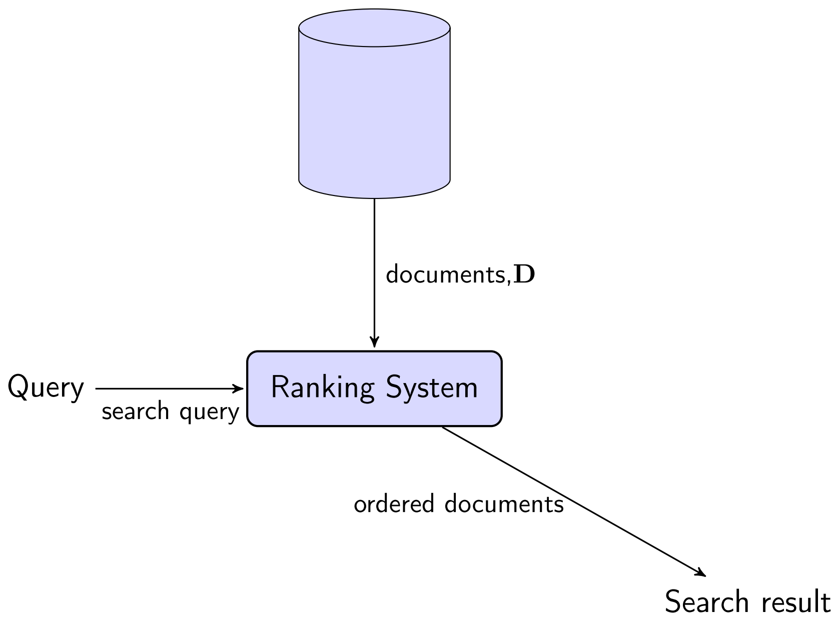


Figure 2.2: The ranking model for information retrieval.

Consider a collection of documents $D = d_1, d_2, ..., d_n$ and a query q, the goal is to let the ranking model $f(q, D)$ permute the order of the documents D ordered with the most relevant results ranked the highest. The top document can then be returned and shown as result to the user who executed the query. The ranking model $f(q, D)$ which is responsible for the ordering, can be created through many different approaches. The machine learning approach *learning to rank* train a model and learn a ranking function $f(q, D)$ (see figure 2.3). The training data consist of a number of query/document pairs with a relevance label (see sub section 2.4.1). This makes it possible to recognize patterns which can be utilized to create good ranking function.

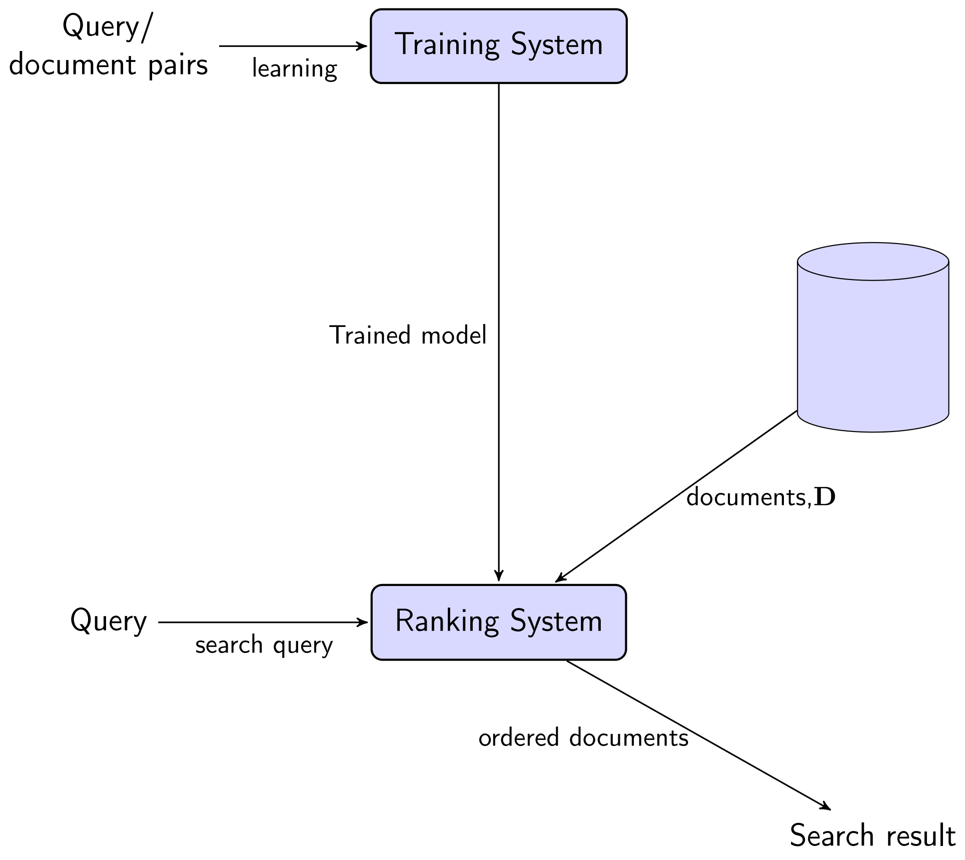


Figure 2.3: The ranking model in Learning to rank for information retrieval.

2.3 Conventional Ranking Models

This section will describe some of the most common standard ranking models, namely Okapi BM25, the Language Model for Information Retrieval (LMIR) and the Vector Space Model (VSM). These ranking models are popular today even though for example BM25 was developed more than 30 years ago.

2.3.1 Okapi BM25

The *Okapi BM25* ranking model (BM25) (Robertson and Zaragoza, 2009), where BM stands for best match is a probabilistic ranking model based on *term frequencies* (see section 5.2.1). BM25 gives each document d in a collection of documents $\mathcal{D}$ a distribution $P(r|q,d)$ given a query q, where r is a binary value 0 or 1; irrelevant or relevant. For a query q the documents with the highest probabilities to be relevant are returned and displayed to the user. The probability of relevance for a document given a query is the *Probability Ranking Principle* stated in "The Probability Ranking Principle in IR" by Robertson (1977) as:

"If retrieved documents are ordered by decreasing probability of relevance on the data available, then the system's effectiveness is the best that can be obtained for the data."

2.3.2 Language Model for Information Retrieval

The *Language model for Information Retrieval* (LMIR) (Manning et al., 2008) and (Robertson and Zaragoza, 2009) is based on the probability ranking principle stated above. The simplest language model is the *unigram* language model, it calculates a probability distribution over a corpus $P(w_1, ..., w_n)$, for each document. Every query has its own language model $M(d)$. Given a query, the language model $M(d)$ with the highest probability $P(q|M(d))$ is the language model of the most relevant document. LMIR returns the most relevant documents to the user by calculating the probabilities for all language models for all documents.

2.3.3 Vector Space Model

The *vector space model* (VSM) described in *Introduction to Information Retrieval* by Manning et al. (2008) and in "Vector-space model" by Melucci (2009) is a ranking model in which queries and documents are represented as weight vectors. The idea with VSM is to rank the documents *term vectors* (or more common *tfidf* vectors) (see sub section 5.2.1) proximity to the query term vector. The documents proximity to the query is based on the cosine between the document and the query in the vector space. The cosine similarity of two vectors is computed by first calculating the norm (l2 norm) of the vectors and then the dot product. The euclidean norm makes the length of all vectors equal and the cosine gives the similarity. A small angle means similar vectors, if for example two identical vectors are compared the cosine will be 0. The equation below shows how the similarity between a term vector of a document **D** and a term vector of a query **Q** is calculated.

$$cosine(\theta) = \frac{\mathbf{Q} \cdot \mathbf{D}}{|\mathbf{Q}||\mathbf{D}|} = \frac{\sum_{i=1}^{n} q_i \times d_i}{\sqrt{\sum_{i=1}^{n} (q_i)^2} \times \sqrt{\sum_{i=1}^{n} (d_i)^2}}$$

2.4 Learning to Rank

In the described models above the relevant documents could be retrieved by calculating probabilities given the query, the collection of documents and a corpus. The goal is to now by using different machine learning approaches achieve better ranking models than earlier. Learning to rank has three main branches, all of which take a *supervised approach* to solve the ranking problem. The meaning of a supervised approach is that the algorithms need to be trained and fed with sample data with relevance labels. This training must be done before the algorithms actually can rank documents without any surveillance. In the *unsupervised approach* training occurs without relevance labels. The three branches, namely *the pointwise approach*, *the pairwise approach* and *the listwise approach* will be described in the next sections. The next sections will also to point out strengths and weaknesses with each approach. Before the explanation of each approach a brief description of the learning- and test- data representation, used for learning to rank will be given.

2.4.1 Data Representation

Annotated data is needed to be able to train a learning to rank ranking model (see figure 2.3). Annotated data is search queries with retrieved documents labeled in accordance to their relevance. These labels comes from human judgement, how well each retrieved document matches what the user is searching for.

The training data consists of query-document pairs for a large number of queries, each with a label in a discrete interval. Most common is the interval $\{0, 4\}$ with 0 as an irrelevant document and 4 as a perfect match. In some cases the relevance labels are binary, $\{0, 1\}$ relevant or irrelevant. Each query is represented by an id and each query/document pair is represented by a feature vector consisting of n features. These features can for example be the documents term frequency, BM25 or PageRank score. Typically each document has many features, for example one of the Microsoft Learning to Rank Datasets[2] has 136 features for each query/document pair and the Yahoo dataset (Chapelle et al., 2011) has 600 features per query/document pair. Appendix A and B lists the features of the MQ2008 and HP2003 benchmark datasets.

A small toy example of the data representation of eight different query/document

[2]`http://research.microsoft.com/en-us/projects/mslr/`

pairs for two different queries with four dimensional feature vectors is shown in table 2.2. Typically the features are normalized.

label	qid:id	feature vector
1	qid:1	1:0.1 2:0.9 3:0.9 4:0.6
0	qid:1	1:0.1 2:0.8 3:0.9 4:0.6
3	qid:1	1:0.8 2:0.2 3:0.3 4:0.4
4	qid:1	1:0.9 2:0.3 3:0.1 4:0.7
3	qid:2	1:0.7 2:0.4 3:0.2 4:0.1
2	qid:2	1:0.1 2:0.8 3:1.0 4:0.3
2	qid:2	1:0.2 2:0.7 3:0.3 4:0.1
0	qid:2	1:0.4 2:0.6 3:1.0 4:0.2

Table 2.2: Typical data representation of a learning to rank dataset.

2.4.2 The Pointwise Approach

The pointwise approach described by Liu (2011) in *Learning to Rank for Information Retrieval* is considered the most basic learning to rank approach. In the Yahoo! Learning to Rank Challenge (Chapelle et al., 2011) the pointwise approach was outperformed by the pairwise and by the listwise approach. The approach handles each query/document pair separately and tries to predict the correct label for each pair. This is not beneficial, as the approach ignores the group structure.

As the pointwise approach treats each single query/document pair as a single object the ranking problem can be reduced, to classification, regression or ordinal-regression. The classification task solves the problem where the target values are the query/document pairs corresponding labels. This means that the classification algorithms try to classify the query/document pairs with correct discrete relevance label given their feature vector. An example of the pointwise approach using classification is Boosting Trees described in "McRank: Learning to Rank Using Multiple Classification and Gradient Boosting" by Li et al. (2008).

In regression each query/document pair is assigned with a continuous label. The goal is to minimize the error or *regret*. Examples of the pointwise approach using regression is "Subset Ranking using Regression" (Cossock and Zhang, 2006).

Ordinal-regression takes all the relevance labels $(1, 2, ..., k)$ in order, and tries to correctly classify the documents with a function f and thresholds $b_1 \leq b_2 ... \leq b_{k-1}$ where k is the number of labels. The output of the function gives a label to each document/query pair in a multi classification fashion. An example of ordinal regression is the PRank algorithm (Crammer and Singer, 2001) which uses *perceptrons*.

2.4.3 The Pairwise Approach

In the pairwise approach (Liu, 2011) two document/query pairs are compared, given a query one of the document is more relevant than the other. By doing this reduction to pairwise classification the approach focuses on the maximization of the number of correct classified document pairs. To have a ranking model that correctly can classify all pairs means that the model also can rank all the documents correctly. Examples of implementations with good performance using the pairwise approach for learning to rank is RankNet (Burges et al., 2005) which uses a neural network approach and RankBoost (Freund et al., 2003) which uses boosting.

2.4.4 The Listwise Approach

In the listwise approach (Liu, 2011) the input to the training system consists of batches of all the documents associated with a certain query. The listwise approach utilize the group structure of the training data. The ranking problem becomes an optimization task to generate a permutation of the input documents such that the permutation maximize the score function of choice. This can be seen as equal to minimizing a loss function. The listwise learning to rank algorithms are divided into two categories, *minimization of measure-specific loss* (maximization of a score function measure such that NDCG, MAP etc.) and *minimization of non-measure-specific loss* (minimization of a loss function). Adarank (Xu and Li, 2007) is a listwise approach in the minimization of non-measure-specific loss category.

Chapter 3

Related Work

3.1 RankLib

RankLib (Dang, 2011) is a library with several learning to rank algorithms implemented. RankLib is part of an open source project called The Lemur Project[1] which contains different search software. The Indri search engine is probably the best known from the Lemur Project. The Lemur Project is widely used in information retrieval research projects and in commercial products. RankLib is written in Java and the algorithms listed in table 3.1 are implemented in the current version (2.4).

Algorithm	Approach
MART (Gradient boosted regression tree)	Pointwise
Random Forests	Pointwise
RankBoost	Pairwise
RankNet	Pairwise
LambdaMART	Pairwise
AdaRank	Listwise
Coordinate Ascent	Listwise
ListNet	Listwise

Table 3.1: Algorithms implemented in RankLib version 2.4

In addition to these learning to rank algorithms RankLib contains different score functions and evaluation tools to test and validate trained models. The metrics for evaluation supported in the current version are: MAP, NDCG@k,

[1]`http://www.lemurproject.org/`

DCG@k, P@k, RR@k and ERR@k (see section 5.2 for a description of NDCG and DCG). K-fold cross-validation is available to predict the performance and features can be selected to specify which features to include for training of the model. Both models and scores can be saved to file and saved models can be loaded to re-rank new documents. The rest of this chapter will give brief explanations of the algorithms in RankLib version 2.4. For the interested reader further reading and implementation details of each algorithm can be found in the paper referred to in each section.

Figure 3.1: The Lemur Project logo

3.1.1 MART

The MART algorithm (Multiple Additive Regression Trees) is based on the theory in "Greedy Function Approximation: A Gradient Boosting Machine" by Friedman (2000). MART is a pointwise approach and it is also known as Gradient Boosting Regression Trees. MART is a boosting technique (it combines weak rankers to form a strong ranker) combined with gradient descent.

3.1.2 RankNet

RankNet is a pairwise approach described in "Learning to Rank Using Gradient Descent" by Burges et al. (2005). RankNet is using a neural network combined with gradient descent steps to control the learning rate in each iteration step. The neural network has two hidden layers and uses backpropagation to minimize a cost function to perform the pairwise ranking. The probabilistic framework in FRank (see chapter 4) is based on the probabilistic framework in RankNet.

3.1.3 RankBoost

RankBoost (Freund et al., 2003) is a pairwise technique based on boosting. RankBoost operates in rounds and choose the feature in each round that minimizes a loss function. FRank (chapter 4) is to a large degree based on RankBoost, as the additive model of weak rankers for learning to rank in FRank is taken from RankBoost. Further the threshold values used in FRank is also adapted from RankBoost.

3.1.4 AdaRank

AdaRank is based on the paper "AdaRank: A Boosting Algorithm for Information Retrieval" by Xu and Li (2007). AdaRank is another boosting technique which combines weak rankers to create the ranking function. The algorithm is inspired by AdaBoost or "Adaptive Boosting" (Schapire, 1999), a well recognized machine learning algorithm. AdaRank takes the listwise approach to the learning to rank problem and tries to minimize the performance measures directly instead of indirect minimization of a loss function as the similar algorithms above.

3.1.5 Coordinate Ascent

The Coordinate Ascent method is described as an optimization method in the paper "Linear Feature-Based Models for Information Retrieval" by Metzler and Bruce Croft (2007). The method optimize through minimization of measure-specific loss, more specifically the mean average precision (MAP). The Coordinate Ascent suffer from getting stuck in *local minimas*, when searching for the *global minima* of the MAP, but by doing a number of restarts (typically 10) this can be avoided.

3.1.6 LambdaMART

LambdaMART described in "Adapting Boosting for Information Retrieval Measures" by Wu et al. (2010) is an ensemble method consisting of boosted regression trees (MART) (Friedman, 2000) in combination with LambdaRank (Burges et al., 2006). LambdaRank is a neural network algorithm for learning to rank with the same basic idea as RankNet (backpropagation to minimize a loss function). LambdaRank's loss function is meant as an upgraded version of the loss function in RankNet with faster running time and better performance on measures. The authors of LambdaRank points out that their algorithm could be combined with boosted trees. The developers of the LambdaMART algorithm implemented an algorithm that does what Burges et al. (2006) did advice.

3.1.7 ListNet

ListNet is an algorithm in RankLib from the paper "Learning to Rank: From Pairwise Approach to Listwise Approach" by Cao et al. (2007). ListNet is using a neural network approach with gradient descent to minimize a loss function, similar to RankNet. ListNet differs from RankNet as the method uses the listwise approach instead of the pairwise approach taken in RankNet. In this way ListNet tries to utilize the benefits of the group structure of the training data.

3.1.8 Random Forests

The Random Forests (Breiman, 2001) is an ensemble of decision trees. In each decision tree a *random subset space* of the data is selected to create the tree. The left out data is then tested on the tree and the *out of bag* data error (OOB error) is calculated (similar to cross validation). The tree with the lowest error is then chosen as the ranking model. The left out data is usually around 1/3 of the data size and the subsets are chosen randomly for each new tree.

3.1.9 Linear Regression

It is not mentioned in the RankLib wiki but there is an implementation of linear regression in the source code of RankLib. The implementation is pointwise and the goal is to find a *regression line* that separates and classifies the documents. The way the RankLib implementation does this is through the *least-squares method* (Lawson and Hanson, 1995).

Chapter 4

Fidelity Loss Ranking

As RankLib is the best known learning to rank library the decision was made to support RankLib by addition of a yet unimplemented algorithm. The choice of algorithm to implement was based on which algorithms that are popular, have proven performance and yet is unimplemented in RankLib. The performance measures of the benchmark datasets in *Learning to Rank for Information Retrieval and Natural Language Processing* by Li (2014) and the Yahoo! Learning to Rank Challenge results (Chapelle et al., 2011) showed that one of the top algorithms from the comparisons was missing in RankLib, *Fidelity Loss ranking* (FRank) (Tsai et al., 2007).

FRank is a pairwise approach developed by Tsai et al. at Microsoft Research who claims in "FRank: A Ranking Method with Fidelity Loss" (Tsai et al., 2007) that *"The FRank algorithm performs effectively in small training datasets..."*. This was the main reason why this algorithm was chosen for implementation as the size of the training data often is limited in enterprise search solutions.

The theoretical approach of FRank is to combine the probabilistic framework of RankNet (Burges et al., 2005) and the additive boosting approach in RankBoost (Freund et al., 2003). A bit simplified FRank is trying to combine a number of weak rankers to get a good ranking function. A weak ranker is a function that classifies a document by just looking at one feature. By looking at many different weak rankers and then choose the one in each iteration giving rise to the minimal fidelity loss, the weak rankers is combined to create a ranking function.

4.1 Fidelity Loss

Fidelity Loss is traditionally used in physics used to measure the difference between two states of a quantum (Birrell and Davies, 1984) but it can also be used to measure the *fidelity* between two different probability distributions. In FRank the loss function is adapted to pairwise classification and the loss of a pair is measured as:

$$F_{ij} = 1 - (\sqrt{P^*_{ij} \cdot P_{ij}} + \sqrt{(1 - P^*_{ij}) \cdot (1 - P_{ij})})$$

P^*_{ij} is the target probability, which is assigned $\{0, 0.5, 1\}$ depending on the labels of the pair. If i has a higher label than j the target is 1, if i and j have the same label, the target is set to 0.5. Finally if document j is more relevant and have a higher label than document i the target probability P^*_{ij} is set to 0. The logistic function is applied to F_{ij} and the new loss function becomes:

$$F_{ij} = 1 - \left(\left[P^*_{ij} \cdot (\frac{e^{o_{ij}}}{1 + e^{o_{ij}}})\right]^{\frac{1}{2}} + \left[(1 - P^*_{ij}) \cdot (\frac{1}{1 + e^{o_{ij}}})\right]^{\frac{1}{2}}\right)$$

where o_{ij} is the rank order of document i and j. Figure 4.1 shows that the loss for $P^*_{ij} = 0.5$ has its minimum at the origin and give rise to zero loss. This is a useful property, RankNet (Burges et al., 2005) for example, which is based on the minimization of a similar loss function does not have minimum loss equal to zero.

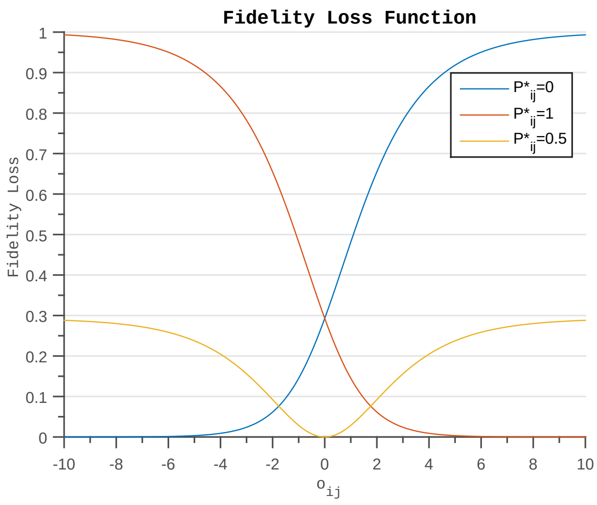


Figure 4.1: The loss function in FRank as a function of o_{ij} for the different target probabilities.

4.2 Algorithm description

The goal is to construct the ranker function $H(x)$ as a combination of weak rankers $h(x)$ and corresponding weights α at each iteration t:

$$H(x) = \sum_t \alpha_t h_t(x)$$

where h_t and α_t denotes the weak ranker and the corresponding weight found at each iteration. In FRank the weak ranker $h_t(x)$ is binary, each feature of each document is compared to a threshold value, and are assigned either 0 or 1. The threshold value can be chosen in different ways, for example many different thresholds for each feature can be used and create several weak rankers and then pick the values with best performance. As the documents often have a large number of features to pick from (typically above 40 features) only one threshold is chosen for each feature in this implementation to keep down the running time. The idea of binary thresholds is adapted from RankBoost (Freund et al., 2003) which also uses the same type of binary classifiers as FRank. In this implementation each threshold is randomly chosen for each feature from a feature value.

The ranking function is constructed by in each iteration choosing the weight a_t which minimize the Fidelity Loss. The Fidelity Loss J over the ranking function H_k is defined as:

$$J(H_k) = \sum_q \frac{1}{|\#q|} \sum_{ij} F_{ij}^{(k)}$$

The first sum is a weight for each document pair where q is the total number of documents for a query. This will make all queries equally important and not bias a query because it got a large number of document pairs. From now on this weight is denoted $D(i,j)$ for a document pair i, j.

$$D(i,j) = \frac{1}{|\#q|}$$

With the formula for the fidelity loss in addition to $D(i,j)$ we get the following equation for the loss:

$$J(H_k) =$$

$$= \sum_{ij} D(i,j) \cdot \Big(1 - \big[P_{ij}^* \cdot (\frac{e^{H_{k-1}^{i,j} + \alpha_k h_k^{i,j}}}{1 + e^{H_{k-1}^{i,j} + \alpha_k h_k^{i,j}}})\big]^{\frac{1}{2}} + \big[(1 - P_{ij}^*) \cdot (\frac{1}{1 + e^{H_{k-1}^{i,j} + \alpha_k h_k^{i,j}}})\big]^{\frac{1}{2}}\Big)$$

where

$$h_k^{i,j} \triangleq h_k(x_i) - h_k(x_j)$$

Because the weak rankers only can have the values 0 or 1, the pairwise ranker $h_k^{i,j}$ can take the values $\{-1, 0, 1\}$ this means that the derivation of the fidelity loss can be simplified as described in Tsai et al. (2007), to obtain the following expression for the weight α_k:

$$\alpha_k = \frac{1}{2} \ln \frac{\sum_{h_k^{i,j}=1} W^{i,j}}{\sum_{h_k^{i,j}=-1} W^{i,j}}$$

$W^{i,j}$ are the pairwise weights initially set to $D(i,j)$ and updated as following in each iteration k:

$$W_k^{i,j} = D(i,j) \cdot \left(\frac{(P_{ij}^* e^{H_{k-1}^{i,j}})^{\frac{1}{2}} - e^{H_{k-1}^{i,j}}(1 - P_{ij}^*)^{\frac{1}{2}}}{(1 + e^{H_{k-1}^{i,j}})^{\frac{3}{2}}} \right)$$

Algorithm 1 shows the pseudocode of the FRank implementation.

Algorithm 1 FRank

```
Input: Document pairs and weak ranker candidates $h_m(x)$ where $m$ is the number of features.

  Calculate the initial pair-weights $D(i,j)$ for each document pair
  for t=1 to k do          ▷ k is the number of weak learners to be combined
      for c=1 to m do                          ▷ m is the number of features
          Calculate $\alpha_{t,c}$
          Calculate fidelity loss $J(H_k)$
      $h_t(x) \leftarrow h_{t,c}(x)$ with minimum loss
      $\alpha_t \leftarrow$ corresponding $\alpha_{t,c}$
      update pair-weights according to $W_k^{i,j}$
Output: Ranker $H(x_i) = \sum_{t=1}^{k} \alpha_t h_t(x)$
```

Chapter 5

Experiments

This section displays the results of the tests ran on the algorithms in RankLib and FRank. The test setup, the score metric and the benchmark datasets are described. This section also describes the started work of implementing learning to rank in Apache Solr.

5.1 Experimental Setup

The main goal with the comparisons was to see if one or several algorithms could perform well when the training data was limited. This is interesting as the algorithms in RankLib usually have very large training sets (the case in the benchmark datasets) which not is easy to create in an enterprise search solution. As the number of extractable features from Solr is limited the comparisons also include tests with the features that can be extracted from Solr. These tests are included to see if this ranking works as good as in a web based setting with more features per document. The top 3 and top 5 results was identified as good numbers of hits to consider relevant.

To be able to draw sensible conclusions from the tests, two baseline algorithms were included in the test runs. Linear Regression was the first method included as it is a simple and more straightforward approach compared to the other algorithms in RankLib. It is valuable to see if and how much better the more advanced approaches perform on the benchmark tests. The second baseline algorithm is just random ordering of the documents. The motivation to include this is that the scores of the other algorithms will be more intuitive when they can be related to the score of the random ordering. As the test sets are scaled down the hypothesis is that the score of the RankLib algorithms eventually will decrease and converge with the score of the random algorithm.

5.2 Definitions & Score Metrics

This sub section describes some common information retrieval definitions and the score metric used in the experiments to measure the performance of the learning to rank algorithms.

5.2.1 Common Definitions in Information Retrieval

In the introduction to the theory chapter two key terms in information retrieval were introduced: *precision* and *recall* (Carterette, 2009). Just to recall, precision is a probability measure of how likely it is that a random chosen document out of all of the documents in the search result is relevant. Recall is a probability measure of how likely it is that a random chosen relevant document is in the search result. Precision and recall are more precisely defined with the following definitions:

$$Precision = \frac{RelevantDocuments \cap RetrievedDocuments}{RetrievedDocuments}$$

$$Recall = \frac{RelevantDocuments \cap RetrievedDocuments}{RelevantDocuments}$$

Often only the top n results are of concern, this is notated as $Precision@n$ or $P@n$. $@n$ notation is intuitive; for example in a web search engine one probably never is concerned with all the retrieved search hits. Seldom more than the first page of search results is of interest.

Three other terms more related to the technical aspects of the search are *term frequency* (tf), *inverse document frequency* (idf) and *term frequency - inverse document frequency* (tf-idf) (Manning et al., 2008). The tf is the number of times a specific word occurs in a document. This is usually calculated for all words in the *corpus*, with *stop words* removed. A corpus is a set of words, for example all word in a specific language. Stop words are small words ('at', 'in', 'this' etc.) that are removed because they do not reflect the content and to make the statistical measures useful. In the *tf model* the documents marked as relevant will simply be the documents with the most frequent occurrences of the word/s in the search query. In the *bag of words* *model* documents are represented by the terms they contain and the corresponding term frequencies.

The idf is as the name implies the inverse of the document frequency. The document frequency for a term t is the number of documents in the collection of documents $\mathcal{D}$ which contains the term. The purpose of the inverse document frequency is to weigh uncommon terms more heavily than common terms. As an example, think of a collection of economics articles, a search query "car finance" will weight "car" much higher than "finance" if the term "finance" is in nearly every economics article and the term "car" is uncommon among the articles in the collection. The inverse document frequency is calculated for each term in the query as:

$$idf = \frac{|\mathcal{D}|}{\log(df)}$$

The scale of the logarithm above can be tuned for the purpose depending on how much one want to boost rare terms.

The tf-idf is meant as an improved version of the term frequency, as sometimes the search results will not be very precise with all terms equally weighted. By combining the inverse document frequency with the term frequency a good composition of two models is obtained. The tf-idf is calculated for each term in the query as:

$$tfidf = tf \cdot idf$$

A high tf-idf score indicates that the term is frequent in a few documents and if the term gets less frequent or more distributed over documents the score will drop.

5.2.2 Discounted Cumulative Gain

One of the most common methods to measure the relevance of a search result with is the *Discounted Cumulative Gain* (DCG) (Li, 2014). The notation $DCG@n$ is used where n is the top number of results taken into consideration. The score is based on the intuition that in a good ranking system the documents with the highest labels will be ranked the highest. With DCG lower search results will contribute less and less to the total score. DCG is calculated with the following equation:

$$DCG@n = \sum_{i=1}^{n} \frac{2^{rel_i} - 1}{\log_2(i+1)}$$

where rel_i is the relevance label of document number i in the ordered result list. DCG is often normalized and becomes the score metric called Normalized Discounted Cumulative Gain (NDCG) (Manning et al., 2008). NDCG is measured as the DCG divided with the Ideal Discounted Cumulative Gain (IDGC). The IDGC is the maximum score a ranked search results possibly can get, i.e. the optimal ordering of the results.

$$NDCG@n = \frac{DCG@n}{IDCG@n}$$

As an illustrating example we calculate the DCG, IDCG and NDCG (@4) for the query with id 1 from Table 2.2. We assume that an imaginary ranking function has ranked the relevance of the documents for query 1 with the relevance in the following decreasing order: $\{4 \succ 1 \succ 3 \succ 2\}$ with row number as document number. The symbol $\succ$ represent ordering, $d_1 \succ d_2$ means that d_1 is ranked higher than d_2. This gives the DCG score:

$$DCG@4 = \frac{2^4 - 1}{\log_2(1+1)} + \frac{2^1 - 1}{\log_2(2+1)} + \frac{2^3 - 1}{\log_2(3+1)} + \frac{2^0 - 1}{\log_2(4+1)} = 19.13$$

the IDCG score (for $\{4 \succ 3 \succ 1 \succ 2\}$):

$$IDCG@4 = \frac{2^4 - 1}{\log_2(1+1)} + \frac{2^3 - 1}{\log_2(2+1)} + \frac{2^1 - 1}{\log_2(3+1)} + \frac{2^0 - 1}{\log_2(4+1)} = 19.92$$

and finally the NDCG score:

$$NDCG@4 = \frac{19.13}{19.92} = 0.97$$

5.3 Benchmark Datasets

The idea behind using benchmark datasets is to let the developers focus on the development of algorithms instead of the gathering of and set up of good data. To find and pre-process data can be very time consuming if no data is available and benchmark datasets can save a lot of time. Benchmark datasets also brings justice to comparisons of algorithms as the developers can not construct a dataset designed specifically for their algorithm. A possible drawback with benchmark datasets is that the developers design an algorithm exclusively for performance on a certain dataset with a poor general performance.

5.3.1 Preprocessing of data

As described in "LETOR: A Benchmark Collection for Research on Learning to Rank for Information Retrieval" by Qin et al. (2010) there are four main preprocessing steps to get useful training data; selecting corpora, sampling of documents, extracting features and extracting metadata. The Gov[1] and Gov2[2] datasets were the chosen corpora in the benchmark datasets. The Gov and Gov2 datasets are publicly available and they are common in different measurements of information retrieval implementations. The datasets are usually very large and as 25 million queries were not needed to create a training set a subset was chosen (the sampling of documents step). The third preprocessing step (the feature extraction part), must be seen as the most important step as the features to represent each query/document pair are chosen. The number of features varies a lot between different benchmark datasets and the decision must be made which of the features that are of interest to create a useful benchmark set. The features extracted into the LETOR datasets was selected with the main goals to include the "classical information retrieval features" (such as tf, idf and tfidf) and choose features often represented in *SIGIR*[3] (Special Interest Group on Informaition Retrieval) papers. The last step, extraction of metadata, is done to be able to tune the dataset. A few examples of metadata (stored in XML) are the number of documents, the length of term vectors and the values of the parameters for the features from the BM25 and the LMIR model. Furthermore some of the features in the benchmark datasets are only document dependent such as *document length*, some are only query dependent such as *idf* but the most of the features are both query and document dependent.

5.3.2 MQ2008

The *Million Query 2008* dataset (MQ2008) is a benchmark dataset from the LETOR 4.0 package (Qin and Liu, 2013). MQ2008 is created by Microsoft Research Asia and the data is coming from the Million Query track from

[1]`http://ir.dcs.gla.ac.uk/test_collections/govinfo.html`
[2]`http://ir.dcs.gla.ac.uk/test_collections/gov2-summary.htm`
[3]`http://sigir.org/`

TREC 2008[4]. The dataset consists of 784 queries from the Gov2 web page collection, Gov2 is a huge collection of approximately 25 million web pages distributed by University of Glasgow. Each query/document pair is represented by a 46 dimensional feature vector, see appendix A for a description of these features. In total there are 15211 query/document pairs in the MQ2008 dataset.

5.3.3 HP2003

The second benchmark dataset used in this thesis is the *Homepage finding 2003* dataset (HP2003) from the Gov collection (the precursor to Gov2). HP2003 is bigger than MQ2008 in terms of document/query pairs, it contains 147606 query/document pairs but only 150 queries. The documents in the Gov collection contains more features than the documents in Gov2, 64 dimensional feature vectors instead of 46 dimensional. See Appendix B for a description of the features in the HP2003 dataset. It is beneficial to see and analyse how the results differs in this type of dataset compared to MQ2008.

5.4 Comparison between the algorithms in RankLib

5.4.1 MQ2008 test results

The first tests were carried out on the MQ2008 dataset. The testing consisted of 5-fold cross-validation and the tables with corresponding plots shows the average results of the testing (5-fold cross-validation) for all of the algorithms in RankLib. The results from three different categories of tests are presented: NDCG@3 score, NDCG@5 score and NDCG@3 score with a limited number of features.

Table 5.1 and figure 5.1 shows the results of the NDCG@3 measure. The results shows that the neural network approaches in ListNet and RankNet had the worst performance (except for Random). This can be due to that a couple of parameters needs to be fine tuned in these algorithms, but for the tests all of the methods ran out of the box with default parameter values. By inspecting the plots in figure 5.1 the results of the top algorithms appears to have very similar performance, with no clear winner.

[4]`http://trec.nist.gov/tracks.html`

For all the algorithms the score decreases when the training data is shrunken below a certain point. This point is approximately somewhere around 1000 − 2000 query/document pairs or 100 queries. At 1077 query/document pairs the performance is starting to decrease and in the smallest test environment with only 592 query/document pairs the performance is even worse. As a final observations the performance of the Linear Regression is surprisingly good, it is both a memory and time efficient method compared to many of the more sophisticated methods.

Query/Doc. Pairs	592	1077	1955	7769	15211
No. of Queries	39	69	126	410	784
MART	0.27	0.38	**0.44**	0.4	**0.42**
RankNet	0.27	0.33	0.34	0.38	0.39
RankBoost	0.35	0.36	0.42	0.4	0.41
AdaRank	0.32	0.37	0.41	0.4	0.4
Coordinate Ascent	0.33	0.37	0.42	**0.41**	**0.42**
LambdaMART	**0.36**	0.37	0.43	**0.41**	0.41
ListNet	0.28	0.32	0.36	0.38	0.38
Random Forests	0.31	0.38	0.43	**0.41**	0.41
FRank	0.31	**0.4**	0.4	**0.41**	0.39
Linear Regression	0.25	0.35	0.4	0.4	0.4
RANDOM	0.24	0.18	0.21	0.2	0.2

Table 5.1: NDCG@3 score for the MQ2008 dataset.

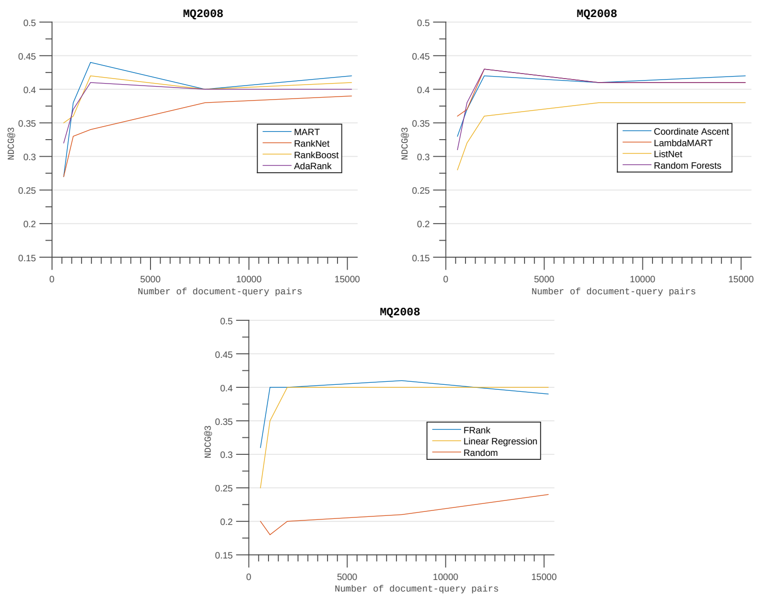


Figure 5.1: NDCG@3 score as a function of the number of query/document pairs in the MQ2008 dataset.

Table 5.2 and figure 5.2 shows the results of the NDCG@5 score on the MQ2008 dataset. The same conclusions as from the NDCG@3 tests can be drawn as the difference between the top 3 and the top 5 results does not differ significantly. The same algorithms are in the top as in the NDCG@3 tests and the decrease in performance occurs at the same size of the tests as in the NDCG@3 case. The performance of FRank is average in both the NDCG@3 and the NDCG@5 case.

Query/Doc. Pairs	592	1077	1955	7769	15211
No. of Queries	39	69	126	410	784
MART	0.33	**0.44**	**0.49**	0.45	**0.46**
RankNet	0.3	0.31	0.43	0.43	0.43
RankBoost	0.42	0.4	0.47	0.45	0.45
AdaRank	0.41	0.43	0.45	0.45	0.45
Coordinate Ascent	**0.43**	0.42	0.46	0.46	**0.46**
LambdaMART	0.38	0.43	0.47	**0.47**	0.45
ListNet	0.3	0.3	0.38	0.41	0.44
Random Forests	0.39	0.42	**0.49**	0.46	0.45
FRank	0.36	0.42	0.45	0.46	0.44
Linear Regression	0.36	0.39	0.45	0.45	0.44
RANDOM	0.32	0.26	0.29	0.26	0.26

Table 5.2: NDCG@5 score for the MQ2008 dataset.

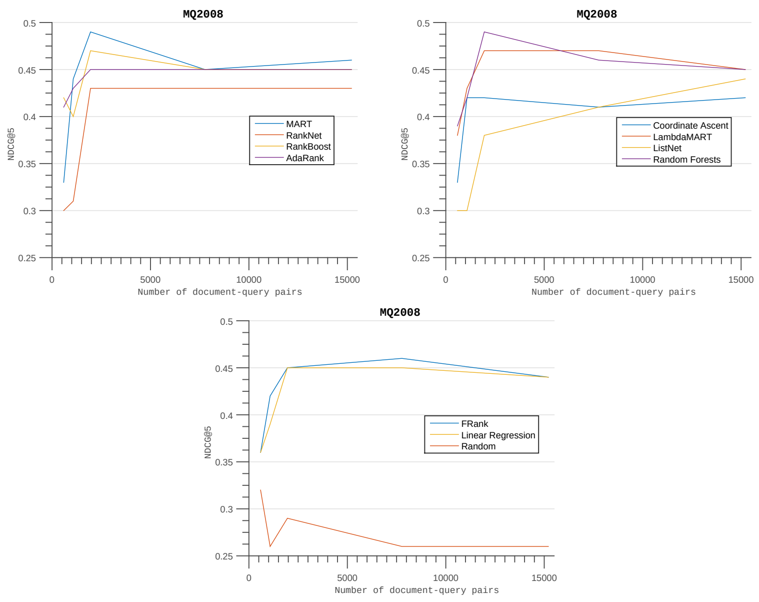


Figure 5.2: NDCG@5 score as a function of the number of query/document pairs in the MQ2008 dataset.

Table 5.3 and figure 5.3 shows the results of the tests ran on the MQ2008 dataset with "Solr features only". This means the features that can be extracted from Solr without too much effort. In the MQ2008 dataset this means feature 1-20 (see Appendix A) tf, idf, tfidf and document length for all indexed fields.As the NDCG@3 and NDCG@5 results above were as similar as they were the Solr features testing was only executed for NDCG@3. The results below clearly shows that the performance decreased when the number of features was limited to 20. The Random Forests algorithm is the top performer when the dataset has the original and half of the original size.

Query/Doc. Pairs	1077	1955	3877	7769	15211
No. of Queries	69	126	253	410	784
MART	0.3	0.27	**0.31**	0.31	0.36
RankNet	0.24	0.22	0.24	0.22	0.21
RankBoost	0.23	0.26	0.25	0.25	0.26
AdaRank	0.24	0.22	0.29	0.26	0.28
Coordinate Ascent	0.3	**0.3**	0.29	0.29	0.31
LambdaMART	**0.32**	0.29	0.28	0.32	0.35
ListNet	0.19	0.19	0.2	0.22	0.24
Random Forests	0.3	0.27	0.3	**0.34**	**0.38**
FRank	0.28	0.28	0.27	0.28	0.26
Linear Regression	0.22	0.2	0.18	0.2	0.22
RANDOM	0.21	0.22	0.22	0.21	0.2

Table 5.3: NDCG@3 score for the MQ2008 dataset with Solr features only.

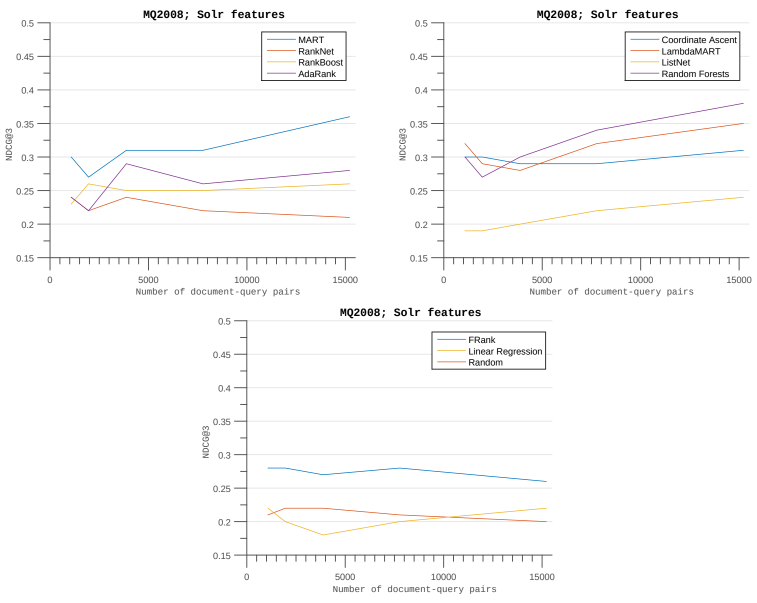


Figure 5.3: NDCG@3 score as a function of the number of query/document pairs in the MQ2008 dataset with Solr features only.

5.4.2 HP2003 test results

This sub section displays the results of the tests ran on the HP2003 benchmark dataset. The HP2003 dataset is bigger than the MQ2008 dataset overall as HP2003 contains more query/document pairs but the number of queries is lower. All algorithms that was included in the MQ2008 tests was not included in the HP2003 tests. This was mainly due to memory issues (not enough memory in the Java heap) regarding the large number of query/document pairs. An important note is that none of the algorithms that had top performance on the MQ2008 tests was left out. The results of the HP2003 set can therefore be seen as a good complement to the MQ2008 tests. The left out algorithms were RankBoost, ListNet and FRank. The results shown below are also as in the MQ2008 case the average results of 5-fold cross-validation.

Table 5.4 and figure 5.4 shows the results of the first tests ran on the HP2003 benchmark set. First and foremost the score is on average higher in the HP2003 tests than in the MQ2008 tests. The NDCG@3 results shows that the best algorithms are Coordinate Ascent and Random Forests. Notice that when the performance of the algorithms in general decrease (when the size of the training data is decreased), the score of the the Linear Regression increase and the different scores are almost converging at 11000 query/document pairs. Random Forests is still the best algorithm overall (closely followed by Coordinate Ascent) even if its score has started to decrease at the 11000 query/document pairs mark.

Query/Doc. Pairs	11000	20000	38000	74000	147606
No. of Queries	10	20	38	74	150
MART	0.48	0.77	0.74	0.75	0.76
RankNet	0.64	0.62	0.83	0.75	0.72
AdaRank	0.67	0.55	0.69	0.77	0.72
Coordinate Ascent	0.69	**0.87**	**0.88**	0.82	0.75
LambdaMART	0.6	0.63	0.73	0.71	0.75
Random Forests	**0.77**	0.85	0.86	**0.85**	**0.79**
Linear Regression	0.68	0.5	0.53	0.51	0.5
RANDOM	0.1	0.08	0.59	0	0.19

Table 5.4: NDCG@3 score for the HP2003 dataset.

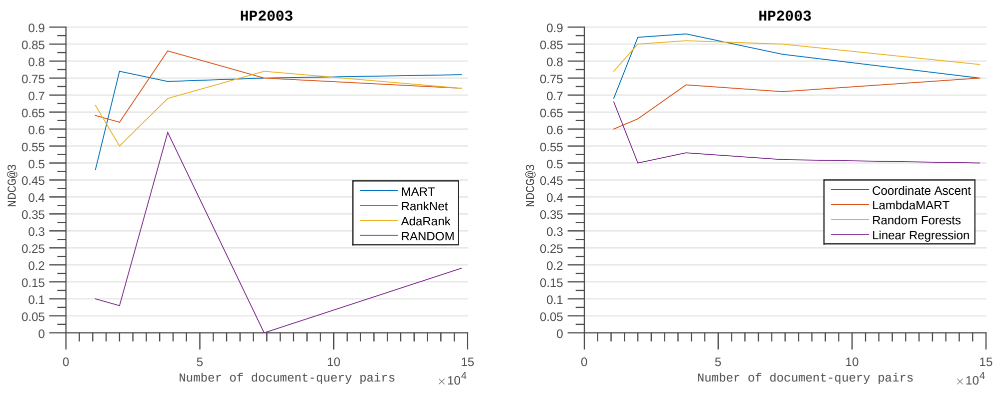


Figure 5.4: NDCG@3 score as a function of the number of query/document pairs in the HP2003 dataset.

The results from the NDCG@5 benchmark tests for the HP2003 dataset are shown in table 5.5 and figure 5.5. Just as in the MQ2008 dataset the top 5 scores resembles to a very large degree the top 3 scores. Notice here that the Random Forests algorithm stands out as the top performer in all of the test configurations closely followed by Coordinate Ascent.

Query/Doc. Pairs	11000	20000	38000	74000	147606
No. of Queries	10	20	38	74	150
MART	0.48	0.78	0.76	0.76	0.77
RankNet	0.48	0.67	0.83	0.8	0.74
AdaRank	0.71	0.74	0.64	0.77	0.75
Coordinate Ascent	0.73	0.84	0.85	0.81	0.76
LambdaMART	0.42	0.65	0.78	0.76	0.74
Random Forests	**0.77**	**0.86**	**0.87**	**0.84**	**0.8**
Linear Regression	0.72	0.57	0.57	0.56	0.52
RANDOM	0.1	0.35	0.61	0.06	0.17

Table 5.5: NDCG@5 score for the HP2003 dataset.

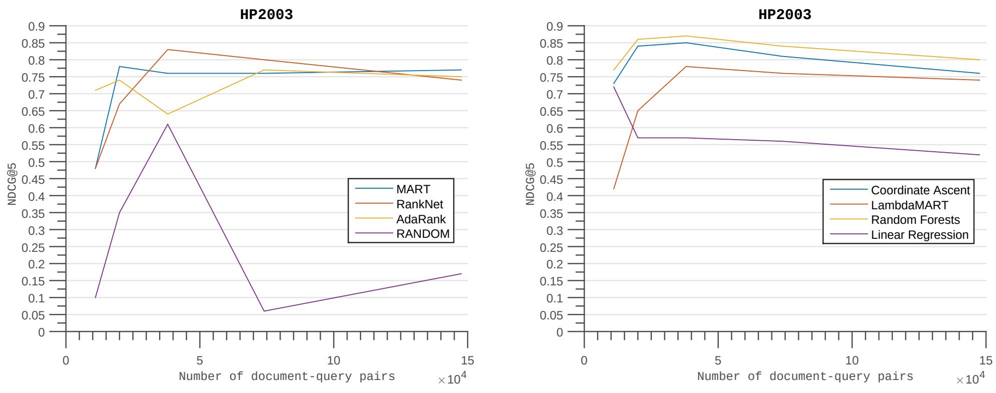


Figure 5.5: NDCG@5 score as a function of the number of query/document pairs in the HP2003 dataset.

The last test carried out was a "Solr features only" test on the HP2003 dataset. The results are displayed in table 5.6 and the corresponding plots are displayed in figure 5.6. By removing more than two thirds of the features the score clearly has decreased compared to the NDCG@3 score above. The Random Forests implementation still has the best performance and it seems like the decrease in performance (in terms of NDCG@3) is equally spread among the different methods. Note that the neural network approach RankNet actually has the best performance without any parameter tuning in the "original size" test (147606 query/document pairs). As the data is scaled the performance of RankNet gets worse and is not better than the other methods.

Query/Doc. Pairs	11000	20000	38000	74000	147606
No. of Queries	10	20	38	74	150
MART	0.51	0.45	0.46	**0.48**	0.44
RankNet	0.24	0.24	0.42	0.41	**0.6**
AdaRank	0.46	0.49	0.43	0.36	0.47
Coordinate Ascent	**0.57**	0.48	0.45	0.4	0.48
LambdaMART	0.48	0.46	0.38	0.47	0.45
Random Forests	**0.57**	**0.5**	**0.53**	**0.48**	0.51
Linear Regression	0.42	0.39	0.43	0.41	0.4
RANDOM	0	0.03	0.08	0.15	0.11

Table 5.6: NDCG@3 score for the HP2003 dataset with Solr features only.

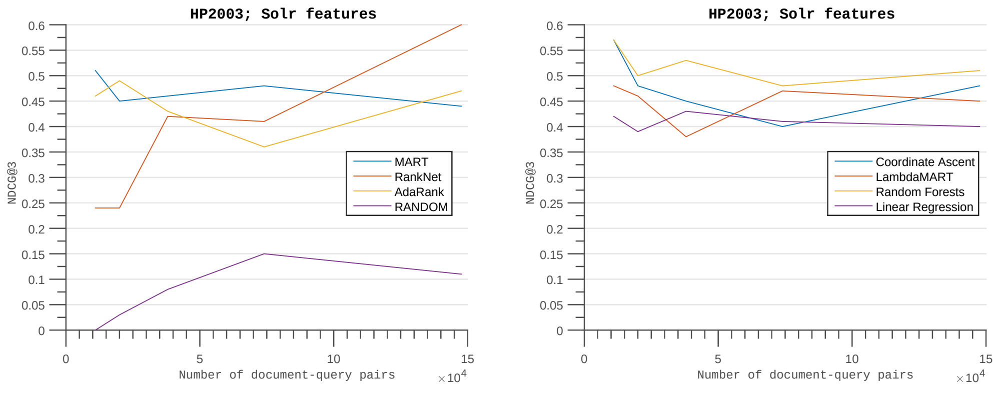


Figure 5.6: NDCG@3 score as a function of the number of query/document pairs in the HP2003 dataset with Solr features only.

5.4.3 A short summary of the results

This summary contains the key content of the results which is stated after careful inspection of the results above. The results in summary is the following:

- The results of the different methods are very similar and there is not a single method that stands out as the best.
- The performance decrease when the number of features is decreased.
- The performance decrease when there is not enough training data.

5.5 Learning to Rank in Apache Solr

This section describes the initiated work to implement learning to rank in Apache Solr. In combination with the future work section it covers what has been accomplished and what should be accomplished next. To be able to follow the description of the initiated work a brief introduction to Solr is given.

5.5.1 Apache Solr

Apache Solr[5] is an open source enterprise search server built on Apache Lucene[6] which is the core that provides search features. Both Solr and Lucene are built entirely in Java and belongs to the Apache Software Foundation, an organization that provides free software based on open source licenses. The software is distributed under the Apache Software Licence which lets anyone modify the code, under the only premise that the modifications are noted. Solr is widely used in the information retrieval industry and the product is considered leading in the field along with Elastic[7].

Solr lets users index documents or equivalent data to a search server (a web application) and users can then issue search queries. Solr matches with documents and returns search results. This is the base functionality of Solr but along comes a number of more advanced features. Some of the more advanced features in Solr are query completion, faceting and full text search.

The default scoring model in Solr is a tfidf model[8] but other scoring models such as BM25 and LMIR are also implemented. This means that the user can choose a suitable scoring model based on the needs. There is a graphical user interface in Solr which can be used for administration. To make the communication more simplified a package called Solrj[9] can be used that allows communication with Solr through REST[10] via Java. Another alternative is direct communication with Solr through REST via the HTTP protocol.

[5] `http://lucene.apache.org/solr/`

[6] `http://lucene.apache.org/core/`

[7] `https://www.elastic.co/`

[8] `http://lucene.apache.org/core/5_0_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html`

[9] `https://cwiki.apache.org/confluence/display/solr/Using+SolrJ`

[10] `http://rest.elkstein.org/`

5.5.2 Learning to Rank in Apache Solr

The implementation communicate with the Lucene core on a quite low-level which means that the code could be implemented theoretically in any Lucene based software (for example Elastic) and not just only in Solr.

The result retrieval starts when a query is posted to the Solr search server. This retrieval consists of two main parts, *matching* and *scoring* (see figure 5.7). Matching, the first part, is where Solr matches documents similar to the query and returns a list of matched documents. In a learning to rank implementation this part does not need to be modified. The second part, the scoring, needs to be modified to be able to implement learning to rank. The modified scoring should be responsible for the ordering of the matched documents with a suitable learning to rank scoring function.

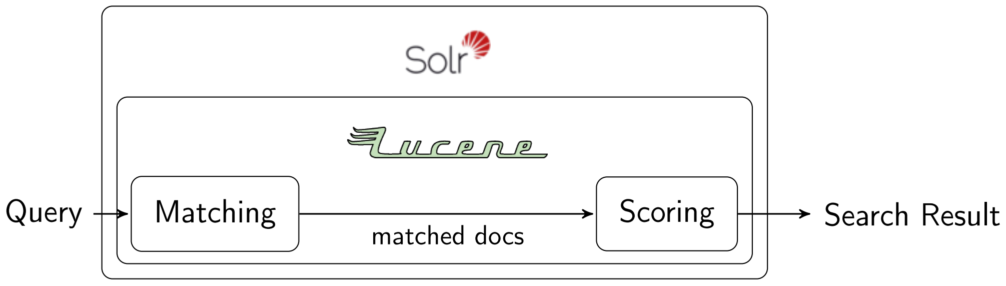


Figure 5.7: Solr matching and scoring.

To understand why the default scoring was needed to be modified underlying understanding of the scoring mechanisms is needed. Each document in Solr consists of a number of indexed fields. The indexed documents are not required to have the same type of fields (TextField, IntField etc.) nor the same number of fields. The way that Solr usually calculates the score is that for all of the fields in a document that match the query, it calculates the score. The score is then either returned as a sum of the individual field scores or as the highest field score. This is depending on the *tie* parameter which can be passed as a parameter in the search query. There are a lot of different parameters in Solr that can be configured according to what fields that should be prioritized in the environment. For the learning to rank implementation these parameters are ignored as the goal is to automatically find good rankers.

Two of the most important steps in the implementation are the feature extraction and the creation of the feature vectors. As the matching documents can have different indexed fields and a different number of features indexed this could result in construction of feature vectors with different features. To solve this problem each document must have the same fields or the *Solr configuration* must force the same fields to be indexed for all documents. Documents often have the same structure and in this implementation the assumption is made that the same fields are indexed. This means that the feature vectors can be constructed with the same lengths. As uniform feature vectors are constructed, the focus is switched to the content of the features and to extract as many features as possible for each field. The feature extraction is important, especially as the experiments showed that the performance of the algorithms improved as the number of features was higher.

In the current implementation the following features are extracted for each field tf, idf and tf-idf. If for example each document has 10 fields, this would mean 30 features to rank on. Features like BM25 and LMIR values for all fields are necessary supplements to get more features. The feature extraction is achieved by creation of a custom query parser plugin. See `LearningPlugin.java` in appendix D for code of how this plugin for Lucene is created. After the plugin is created the custom scoring should be implemented in `LearningQuery.java` (appendix D).

The method `customExplain` demonstrates how terms can be extracted for each field in a document. The `customScore` method should be responsible to do the ranking and return the score for each document. The methods `customScore` and `customExplain` are called for every matched document. The extracted features in the current implementation are extracted from the tfidf similarity model in Lucene, which also is the default similarity model in Solr. Further Lucene contains other similarity models such as BM25 and LMIR. By extracting features for these two similarity models together with the default similarity enough features for each field can be extracted to be able to do learning to rank with good performance. For further information about learning to rank in Solr and which direction that is recommended to continue in see the future work section.

Chapter 6

Discussion

This section reviews the results from the tables and the plots in the experiments section in a wider perspective. To begin with, no evaluated method stands out as the best method. Random Forests had the best performance overall but most of the other algorithms were still very close to Random Forests even if they were a bit behind. The assumption is that the most or perhaps all of the algorithms could gain from parameter tuning, for example the neural networks approaches RankNet and ListNet. This assumption is supported by Li (2014) in *Learning to Rank for Information Retrieval and Natural Language Processing* who presents results from tuned algorithms with better performance than the results in this thesis. There is no proof for which of the methods that could gain the most from tuning and the experimentsf proves that some of the methods are not suitable for out of the box learning to rank. The reason why parameter tuning was ignored was to see the out of the box performance as this often is a sought property in enterprise search. The goal was to find a solution that was time efficient out of the box and to fine tune parameters for all of the methods was not an option.

The NDCG scores decreased when the number of features were limited, this means that the number of features is an essential factor to be able to create a satisfying ranking function. The reason for why the score is decreasing is because when there are not enough features it is impossible to create a good ranker with the limited amount of information available. The difference between the MQ2008 and HP2003 results (HP2003 had higher scores) was expected as the HP2003 set is considered easier to rank according to *Learning to Rank for Information Retrieval and Natural Language Processing* (Li, 2014). This should be partly due to the 64-dimensional features vectors in HP2003 compared to the 46-dimensional in MQ2008. Another likely reason for the better results in HP2003 could be that there are a lot more documents to choose from for each query in HP2003. It should be easier to find three or five relevant documents (labeled highly relevant i.e. labeled 4 on a scale 0-4) when choosing from 1000 documents per query (HP2003) compared to choosing from 20 documents per query (MQ2008). There is still no certainty that that is the case but it is likely as the results are better in the HP2003 performance measures.

Further inspection of the experiments reveals that the training dataset must be big enough to be able to create a good ranking function. How large the training set exactly should be varies from case to case and obviously depends on the documents. A good measure discovered in this thesis is that when the training data is shrunken and the measure-specific loss on the validation data starts to increase there is not enough data.

A final remark about the FRank implementation; the authors (Tsai et al., 2007) claims that the algorithm works better than its competitors on smaller datasets but that clearly is not the case in the tests executed. This could be due to lack of parameter fitting in the experiments (too few iterations etc.) or due to the research bias that seems to occur in many papers in the research field of information retrieval. Researchers develops methods which have the best performance compared to other methods in their test setting. The experiments in this thesis showed that there is not a single method that always is best in a general test setting. This fact is famously stated and known as the "no free lunch" theorem.

Chapter 7

Conclusions & Future Work

7.1 Conclusions

This thesis has evaluated a supervised machine learning approach for ranking of documents called learning to rank. Learning to rank traditionally is used in web based search and this thesis is evaluating learning to rank in enterprise search. The outcome of the thesis is that learning to rank is more suitable for web based search than for enterprise search. There is more data available in web based search and it is easier to train a model efficiently with enough training data. The number of extractable features is often limited in an enterprise search solution. This makes it hard for learning to rank to compete with standard search models in enterprise search as learning to rank is dependent on the number of features. An implementation of learning to rank in Apache Solr does still have potential to be superior to standard models but there must be enough extractable features, a large number indexed fields and enough annotated data. This thesis has begun the implementation of a potential solution, this implementation can be continued as future work.

7.2 Future Work

This section gives advice on how this work can be continued (by another master thesis or similar). As a recommendation four steps are given of how the work can proceed.

1. Create feature vectors in Solr. This is already started in this thesis, and the key is to extract as many features as possible. In Solr many similarity models can be used simultaneously by creating a `QueryParserPlugin` and implement a `CustomScoreQuery` to extract features in different

ways. It is not necessary to implement a new query parser and it is not enough to only implement a custom similarity.

2. Save feature vectors for different queries, inspect and give retrieved search hits relevance labels.

3. Train a suitable learning to rank algorithm with the annotated data created.

4. Use this model as scorer in your `CustomScoreQuery`, to rank and give the documents matched by Solr a custom score.

Finally, online learning have possibilities in learning to rank in combination with Solr. To automatically give relevance labels and create training data based on what the users are clicking (step 2 above). This can be achieved by saving and extracting information from click logs and combine the information with the corresponding documents. The technique is called *click through analysis* (Joachims, 2002) and is described in the next section.

7.2.1 Click through analysis

By using click through analysis a trained learning to rank model can be further trained after it is launched. This type of continuous learning is often refereed to as online learning. The opposite is called offline or batch learning which is the typical case for learning to rank. By monitoring the user behaviour from logs, a model can learn from what the users are actually clicking on in the retrieved search result. In this way the users behaviour automatically creates training data. The notion is that a relevant document makes the user click on the link to the document and stop the searching as the user found relevant information. The time between the retrieved results and the click can be monitored to see how long time the user needed to decide which link to click on. A fast click can mean that the user found the relevant information directly without hesitation. If the user continues to ask new similar queries then the first links probably was not as relevant as desired.

Appendix A

Feature description - MQ2008

The first dataset used to measure the performance of the different algorithms in RankLib was the MQ2008 supervised ranking dataset from LETOR 4.0[1]. The set contains document-query pairs with 46 different features.

Abbreviations:
TF - Term Frequency
IDF - Inverse Document Frequency
DL - Document Length
LMIR.ABS, LMIR.DIR, LMIR.JM - Different LMIR scoring functions.

Table A.1: Features included in the MQ2008 dataset.

Feature Number	Description
1	TF, body
2	TF, anchor
3	TF, title
4	TF, URL
5	TF, document
6	IDF, body
7	IDF, anchor
8	IDF, title
9	IDF, URL
10	IDF, document
11	TF-IDF, body
12	TF-IDF, anchor

Continued on next page

[1]`http://research.microsoft.com/en-us/um/beijing/projects/letor/`

Table A.1 – continued from previous page

Feature Number	Description
13	TF-IDF, title
14	TF-IDF, URL
15	TF-IDF, document
16	DL, body
17	DL, anchor
18	DL, title
19	DL, URL
20	DL, document
21	BM25, body
22	BM25, anchor
23	BM25, title
24	BM25, URL
25	BM25, document
26	LMIR.ABS, body
27	LMIR.ABS, anchor
28	LMIR.ABS, title
29	LMIR.ABS, URL
30	LMIR.ABS, document
31	LMIR.DIR, body
32	LMIR.DIR, anchor
33	LMIR.DIR, title
34	LMIR.DIR, URL
35	LMIR.DIR, document
36	LMIR.JM, body
37	LMIR.JM, anchor
38	LMIR.JM, title
39	LMIR.JM, URL
40	LMIR.JM, document
41	PageRank
42	Number of inlinks
43	Number of outlinks
44	Number of slashs in URL
45	Length of URL
46	Number of child pages

Appendix B

Feature description - HP2003

The second dataset used to measure the performance of the different algorithms in RankLib was the HP2003 supervised ranking dataset from the Gov collection[1]. The set contains document-query pairs with 64 different features.

Abbreviations:
TF - Term Frequency
IDF - Inverse Document Frequency
DL - Document Length
LMIR.ABS, LMIR.DIR, LMIR.JM - Different LMIR scoring functions.

Table B.1: Features included in the HP2003 dataset.

Feature Number	
1	TF, body
2	TF, anchor
3	TF, title
4	TF, URL
5	TF, document
6	IDF, body
7	IDF, anchor
8	IDF, title
9	IDF, URL
10	IDF, document
11	TF-IDF, body
12	TF-IDF, anchor

Continued on next page

[1]`http://ir.dcs.gla.ac.uk/test_collections/govinfo.html`

Table B.1 – continued from previous page

Feature Number	Description
13	TF-IDF, title
14	TF-IDF, URL
15	TF-IDF, document
16	DL, body
17	DL, anchor
18	DL, title
19	DL, URL
20	DL, document
21	BM25, body
22	BM25, anchor
23	BM25, title
24	BM25, URL
25	BM25, document
26	LMIR.ABS, body
27	LMIR.ABS, anchor
28	LMIR.ABS, title
29	LMIR.ABS, URL
30	LMIR.ABS, document
31	LMIR.DIR, body
32	LMIR.DIR, anchor
33	LMIR.DIR, title
34	LMIR.DIR, URL
35	LMIR.DIR, document
36	LMIR.JM, body
37	LMIR.JM, anchor
38	LMIR.JM, title
39	LMIR.JM, URL
40	LMIR.JM, document
41	Sitemap based term propagation
42	Sitemap based score propagation
43	Hyperlink based score propagation: weighted in-link
44	Hyperlink based score propagation: weighted out-link
45	Hyperlink based score propagation: uniform out-link
46	Hyperlink based propagation: weighted in-link
47	Hyperlink based feature propagation: weighted out-link
48	Hyperlink based feature propagation: uniform out-link
49	HITS authority

Continued on next page

Table B.1 – continued from previous page

Feature Number	Description
50	HITS hub
51	PageRank
52	HostRank
53	Topical PageRank
54	Topical HITS authority
55	Topical HITS hub
56	Inlink number
57	Outlink number
58	Number of slash in URL
59	Length of URL
60	Number of child page
61	BM25, extracted title
62	LMIR.ABS, extracted title
63	LMIR.DIR, extracted title
64	LMIR.JM, extracted title

Appendix C

FRank code implementation

This appendix contains the source code for the FRank implementation. FRank is written in Java and the main reason for that is because RankLib also is implemented in Java. A number of helper functions from RankLib are used by the algorithm to parse and iterate through the training and validation data. The performance measures (MAP, NDCG, ERR etc.) are calculated by functionality already implemented in RankLib. A large part of the FRank implementation is based on the RankBoost implementation as the two methods have similar approaches and their implementation details resembles each other. `Frank.java` is the main class and `FrankWeakRanker.java` is a helper class that's representing the weak rankers that are created in each iteration of the FRank algorithm.

The code has two methods of special interest, `init()` and `learn()` (learn calls the `learnWeakRanker()` helper method). The `init()` is just as the name implies a method for initialization of all data structures. In the `init()` the threshold values are assigned, chosen from an already existing feature value. If every feature value is 0 or 1, a random value between 0 and 1 is selected as threshold value. The target probabilities (see section 4) are also calculated in `init()` by doing pairwise comparisons between all the query/documents pairs relevance labels.

The `learn()` method is following the implementation details of the FRank algorithm. The method runs for a specific number of iterations (the number of weak rankers to combine) which can be specified by the input parameter `-round n`. The default number of iterations is currently set to 100. After the ranker function is calculated a number of performance measures are printed out depending on what the user has specified. As a final mention a number of methods are overridden from the `Ranker` superclass to fill the criteria for

Frank.java to be a valid Ranker subclass.

C.1 Frank.java

```java
package ciir.umass.edu.learning;

import java.io.BufferedReader;
import java.io.StringReader;
import java.util.ArrayList;
import java.util.List;

import ciir.umass.edu.metric.MetricScorer;
import ciir.umass.edu.utilities.SimpleMath;

public class Frank extends Ranker{

    public static int nIterations =100;//number of rounds

    // sample weights D(i,j) calculated for each query-pair
    protected double[][][] sweights = null;

    // weights for each query-pair W(i,j)
    protected double[][][] weights = null;

    // binary weak ranker values for each query-pair
    protected double[][][] h_ij = null;
    protected double[][][] H_ij = null;
    protected double[][][] targetProbability=null;

    protected double[][] thresholds = null;

    //best weak rankers at each round
    protected List<FrankWeakRanker> wRankers = null;

    //alpha (weak rankers' weight)
    protected List<Double> rWeight = null;

    protected double bestLoss=1E6;
    protected double H_0 =0.0;

    protected double Inf=Double.POSITIVE_INFINITY;

    protected int n;

    protected double bestAlpha;

    //to store the best model on validation data (if specified)
    protected List<FrankWeakRanker> bestModelRankers =
        new ArrayList<FrankWeakRanker>();
    protected List<Double> bestModelWeights = new ArrayList<Double>();


    public Frank(){
    }

    public Frank(List<RankList> samples, int[] features, MetricScorer scorer){
        super(samples, features, scorer);
    }

    @Override
    public void init(){
        PRINTLN("INITIALIZING...");

        wRankers = new ArrayList<FrankWeakRanker>();
        rWeight = new ArrayList<Double>();

        //generate random thresholds for each feature for each iteration
        thresholds=new double[nIterations][];

        for(int i=0;i<nIterations;i++){
            thresholds[i]= new double[features.length+1];
            //If nIterations > sample size we just choose same thresholds again
            RankList rl = samples.get(i%samples.size());
            for(int f=1;f<=features.length;f++){//features start at 1!
                int v=0;
                while((thresholds[i][f]=rl.get(v).getFeatureValue(f))==0
                    || thresholds[i][f]==1){
                    v++;
                    if(v==rl.size()){
                        thresholds[i][f]=Math.random();
                        break;
                    }
                }
            }
        }

        //calculate the weights for all pairs by equation 1.
        sweights = new double[samples.size()][][];
        weights = new double[samples.size()][][];
        H_ij = new double[samples.size()][][];
        targetProbability = new double[samples.size()][][];
        for(int i=0;i<samples.size();i++){

            //make sure the training samples are ordered by ranking
            samples.set(i, samples.get(i).getCorrectRanking());
            RankList rl = samples.get(i);

            sweights[i] = new double[rl.size()][];
            weights[i] = new double[rl.size()][];
            H_ij[i] = new double[rl.size()][];
            targetProbability[i] = new double[rl.size()][];
            for(int j=0;j<rl.size()-1;j++){

                sweights[i][j] = new double[rl.size()];
                weights[i][j] = new double[rl.size()];
                H_ij[i][j] = new double[rl.size()];
                targetProbability[i][j] = new double[rl.size()];
                for(int k=j+1;k<rl.size();k++){
                    sweights[i][j][k] = 1.0 / rl.size();
                    weights[i][j][k] = 1.0 / rl.size();
                    H_ij[i][j][k]=0.0;

                    if(rl.get(j).getLabel()>rl.get(k).getLabel()){
                        targetProbability[i][j][k]=1.0;
                    }
                    else if(rl.get(j).getLabel()==rl.get(k).getLabel()){
                        targetProbability[i][j][k]=0.5;
                    }
                    else{
                        targetProbability[i][j][k]=0.0;
                    }
                }
            }
        }

        PRINTLN("INITIALIZING DONE");
    }

    private FrankWeakRanker learnWeakRanker(){

        double num;
        double denom;
        double alpha;

        int bestIndex =-1;
        bestAlpha=-1;

        double loss;
        double exp;
        double pow1;
        double pow2;
        double rank_j;
        double rank_i;

        bestLoss=Inf;

        for(int f=1;f<=features.length;f++){
            //calculate alpha for each feature by Equation 3.
            num=0.0;
            denom=0.0;
            h_ij = new double[samples.size()][][];

            for(int i=0;i<samples.size();i++){

                RankList rl = samples.get(i);
                h_ij[i] = new double[rl.size()][];
                for(int j=0;j<rl.size()-1;j++){

                    h_ij[i][j] = new double[rl.size()];
                    for(int k=j+1;k<rl.size();k++){
                        rank_i=0;
                        rank_j=0;

                        if(rl.get(j).getFeatureValue(f)>thresholds[n-1][f]){
                            rank_i=1;
                        }
                        if(rl.get(k).getFeatureValue(f)>thresholds[n-1][f]){
                            rank_j=1;
                        }

                        h_ij[i][j][k]=rank_i-rank_j;

                        if(h_ij[i][j][k]==1){
                            num+=weights[i][j][k];
                        }
                        else if(h_ij[i][j][k]==-1){
                            denom+=weights[i][j][k];
                        }
                    }
                }
            }
            //Equation 3
            alpha=0.5*SimpleMath.ln(num/denom);

            if(Double.isInfinite(alpha) || Double.isNaN(alpha)){
                alpha=0;
            }
            //calculate fidelity loss for each feature by Equation 2.
            loss=0.0;

            //only calculate loss if alpha is valid
            if(!Double.isInfinite(alpha) && !Double.isNaN(alpha)){
                for(int i=0;i<samples.size();i++){
                    RankList rl = samples.get(i);
                    for(int j=0;j<rl.size()-1;j++){
                        for(int k=j+1;k<rl.size();k++){
                            exp=Math.exp(H_ij[i][j][k]+(alpha*h_ij[i][j][k]));
                            pow1=Math.pow(targetProbability[i][j][k]*
                                (exp/(1.0+exp)),0.5);
                            pow2=Math.pow((1.0-targetProbability[i][j][k])*
                                (1.0/(1.0+exp)),0.5);
                            loss+=(sweights[i][j][k]*(1-pow1-pow2));
                        }
                    }
                }
            }

            if(loss < bestLoss){
                bestLoss =loss;
                bestAlpha=alpha;
                bestIndex=f;
            }
        }
        if(bestIndex==-1){
            return null;
        }
        return new FrankWeakRanker(bestIndex, thresholds[n-1][bestIndex]);
    }

    @Override
    public void learn(){

        PRINTLN("-----------------------------------------------------------
            ---------------------");
        PRINTLN("Training starts...");
        PRINTLN("-----------------------------------------------------------
            ---------------------");
        PRINTLN(new int[]{7, 8, 8, 9, 9, 9, 9}, new String[]{"#iter", "Loss",
         "alpha","Threshold",
                "Feature", scorer.name()+"-T", scorer.name()+"-V"});
        PRINTLN("-----------------------------------------------------------
            ---------------------");

        for(n=1;n<=nIterations;n++){
            FrankWeakRanker wr = learnWeakRanker();
            if(wr == null){//no more features to select
                System.out.println("No more features to select");
                break;
            }

            wRankers.add(wr);
            rWeight.add(bestAlpha);

            double num;
            double denom;
            double exp;

            //Update weights according to Equation 4.
            for(int i=0;i<samples.size();i++){
                RankList rl = samples.get(i);
                for(int j=0;j<rl.size()-1;j++){
                    for(int k=j+1;k<rl.size();k++){

                        //update ranker
                        H_ij[i][j][k]+=bestAlpha*h_ij[i][j][k];

                        //update weights
                        exp=Math.exp(H_ij[i][j][k]);
                        num=Math.pow(targetProbability[i][j][k]*exp,0.5)-
                            (exp*Math.pow(1-targetProbability[i][j][k], 0.5));
                        denom=Math.pow(1+exp,1.5);
                        weights[i][j][k]=sweights[i][j][k]*(num/denom);
                    }
                }
            }

            PRINT(new int[]{7, 8, 8, 9, 9}, new String[]{n+"", bestLoss+"",
                SimpleMath.round(bestAlpha,3)+"",
                    SimpleMath.round(wr.getThreshold(),3)+"",wr.getFID()+""});
            PRINT(new int[]{9}, new String[]{
                SimpleMath.round(scorer.score(rank(samples)), 4)+""});

            if(validationSamples != null){
                double score = scorer.score(rank(validationSamples));
                if(score > bestScoreOnValidationData){
                    bestScoreOnValidationData = score;
                    bestModelRankers.clear();
                    bestModelRankers.addAll(wRankers);
                    bestModelWeights.clear();
                    bestModelWeights.addAll(rWeight);
                }
                PRINT(new int[]{9}, new String[]{SimpleMath.round(score, 4)+""});
            }
            PRINTLN("");
        }

        if(validationSamples != null && bestModelRankers.size()>0){
            wRankers.clear();
            rWeight.clear();
            wRankers.addAll(bestModelRankers);
            rWeight.addAll(bestModelWeights);
        }

        scoreOnTrainingData = SimpleMath.round(scorer.score(rank(samples)), 4);
        PRINTLN("--------------------------------------------------------------
            ------");
        PRINTLN("Finished sucessfully.");
        PRINTLN(scorer.name() + " on training data: " + scoreOnTrainingData);

        if(validationSamples != null){
            bestScoreOnValidationData = scorer.score(rank(validationSamples));
            PRINTLN(scorer.name() + " on validation data: " +
                SimpleMath.round(bestScoreOnValidationData, 4));
        }
        PRINTLN("---------------------------------");
    }

    @Override
    public Ranker clone(){
        return new Frank();
    }

    @Override
    public String toString(){
        String output = "";
        for(int i=0;i<wRankers.size();i++)
            output += wRankers.get(i).toString() + ":" + rWeight.get(i) +
        ((i==wRankers.size()-1)?"":" ");
        return output;
    }

    @Override
    public String model(){
        String output = "## " + name() + "\n";
        output += "## Iteration = " + nIterations + "\n";
        output += toString();
        return output;
    }

    @Override
    public void loadFromString(String fullText){
        try {
            String content = "";
            BufferedReader in = new BufferedReader(new StringReader(fullText));

            while((content = in.readLine()) != null){
                content = content.trim();
                if(content.length() == 0)
                    continue;
                if(content.indexOf("##")==0)
                    continue;
                break;
            }
            in.close();

            rWeight = new ArrayList<Double>();
            wRankers = new ArrayList<FrankWeakRanker>();

            int idx = content.lastIndexOf("#");
            if(idx != -1){//remove description at the end of the line (if any)
                //remove the comment part at the end of the line
                content = content.substring(0, idx).trim();
            }

            String[] fs = content.split(" ");
            for(int i=0;i<fs.length;i++){
                fs[i] = fs[i].trim();
                if(fs[i].compareTo("")==0){
                    continue;
                }
                String[] strs = fs[i].split(":");
                int fid = Integer.parseInt(strs[0]);
                double threshold = Double.parseDouble(strs[1]);
                double weight = Double.parseDouble(strs[2]);
                rWeight.add(weight);
                wRankers.add(new FrankWeakRanker(fid, threshold));
            }

            features = new int[rWeight.size()];
            for(int i=0;i<rWeight.size();i++){
                features[i] = wRankers.get(i).getFID();
            }
        }
        catch(Exception ex){
            System.out.println("Error in FRank::load(): " + ex.toString());
        }
    }

    @Override
    public String name(){
        return "FRank";
    }

    @Override
    public void printParameters(){
        PRINTLN("Number of iterations: \t"+nIterations);
    }

    public double eval(DataPoint p){
        double score = 0.0;
        for(int j=0;j<wRankers.size();j++){
                score += rWeight.get(j) * wRankers.get(j).score(p);
            }
            return score;
        }
}
```

C.2 FrankWeakRanker.java

```java
package ciir.umass.edu.learning;

public class FrankWeakRanker {
    private int fid = -1;
    private double threshold = 0.0;

    public FrankWeakRanker(int fid, double threshold){
        this.fid = fid;
        this.threshold = threshold;
    }

    public int score(DataPoint p){
        if(p.getFeatureValue(fid) > threshold){
            return 1;
        }
        return 0;
    }

    public int getFID(){
        return fid;
    }

    public double getThreshold(){
        return threshold;
    }

    public String toString(){
        return fid + ":" + threshold;
    }
}
```

Appendix D

Apache Solr code implementation

This appendix contains the code for the started work on the implementation of learning to rank in Apache Solr. `LearningPlugin.java` is the plugin for Lucene and `LearningQuery.java` is the custom scoring which belongs to this plugin.

D.1 LearningPlugin.java

```
public class LearningPlugin extends QParserPlugin{

@Override
  public void init(NamedList args) {
    SolrParams params = SolrParams.toSolrParams(args);
    // handle configuration parameters
    // passed through solrconfig.xml
  }

  @Override
  public QParser createParser(String qstr,
          SolrParams localParams, SolrParams params, SolrQueryRequest req) {

    return new LearningParser(qstr, localParams, params, req);
  }

  private static class LearningParser extends QParser {

    private Query innerQuery;

    public LearningParser(String qstr, SolrParams localParams,
            SolrParams params, SolrQueryRequest req) {
      super(qstr, localParams, params, req);
      try {
        QParser parser = getParser(qstr, "lucene", getReq());
```

```java
            this.innerQuery = parser.parse();
        } catch (SyntaxError ex) {
            throw new RuntimeException("error parsing query", ex);
        }
    }

    @Override
    public Query parse() throws SyntaxError {
        return new LearningQuery(innerQuery);
    }
  }
}
```

D.2 LearningQuery.java

```java
public class LearningQuery extends CustomScoreQuery{

    private Query subQuery;
    private Set<Term> terms =null;
    private TFIDFSimilarity tfsim=null;
    private LeafReader reader =null;

    public LearningQuery(Query subQuery){
        super(subQuery);
        this.subQuery=subQuery;
        tfsim= new DefaultSimilarity();
        terms = new HashSet<Term>();

        subQuery.extractTerms(terms);
    }

    @Override
    protected CustomScoreProvider getCustomScoreProvider(
            LeafReaderContext context) throws IOException {
        return new MyScoreProvider(context);
    }

    class MyScoreProvider extends CustomScoreProvider {

        public MyScoreProvider(LeafReaderContext context) {
            super(context);
            reader=context.reader();
        }

        @Override
        public float customScore(int doc, float subQueryScore,
        float valSrcScore) throws IOException {
            return customScore(doc, subQueryScore, new float[]{valSrcScore});
        }

        @Override
        public float customScore(int doc, float subQueryScore,
        float[] valSrcScores) throws IOException {
            //here the feature vector should be extracted and the documents
            //should be scored based on a suitable learning to rank algorithm.

            //The score returned here is what Solr will rank on.
            return 1;
        }

        @Override
        public Explanation customExplain(int doc, Explanation subQueryExpl,
            Explanation valSrcExpl) throws IOException{
            return customExplain(doc, subQueryExpl, new Explanation[]
                {valSrcExpl});
        }

        //Explain is a method for printing the extracted features
        @Override
        public Explanation customExplain(int doc, Explanation subQueryExpl,
            Explanation[] valSrcExpl) throws IOException{

            Explanation exp2 = new Explanation();
            Fields fields =MultiFields.getFields(reader);
            for(String field :fields){
                TermsEnum termsEnum = MultiFields.getTerms(reader, field)
                    .iterator(null);
                DocsEnum docsEnum;
                BytesRef bytesRef;
                while ((bytesRef = termsEnum.next()) != null){
                    if (termsEnum.seekExact(bytesRef)){
                        String term = bytesRef.utf8ToString();
                        for(Term t:terms){
                            if(term.equals(t.text())){
                                docsEnum = termsEnum.docs(null, null,
                                    DocsEnum.FLAG_FREQS);
                                while (docsEnum.nextDoc() !=
                                    DocIdSetIterator.NO_MORE_DOCS){
                                    float tf=tfsim.tf(docsEnum.freq());
                                    Explanation exp = new Explanation(tf,
                                        "tf in "+field);
                                    exp2.addDetail(exp);
                                    float idf=tfsim.idf(termsEnum.docFreq(),
                                        reader.numDocs());
                                    exp = new Explanation(idf,"idf in "+field);
                                    exp2.addDetail(exp);
                                    float tfidf=tf*idf;
```

```
                                        exp = new Explanation(tfidf,"tf-idf in "+
                                            field);
                                        exp2.addDetail(exp);

                                    }
                                }
                                else{
                                    //if no field match all fields are 0
                                    Explanation exp = new Explanation(0,"tf in "+
                                        field);
                                    exp2.addDetail(exp);
                                    exp = new Explanation(0,"idf in "+field);
                                    exp2.addDetail(exp);
                                    exp = new Explanation(0,"tf-idf in "+field);
                                    exp2.addDetail(exp);
                                }
                            }
                        }
                    }
                }
                return exp2;
            }
        }
    }
```

Bibliography

N.D. Birrell and P.C.W. Davies. *Quantum Fields in Curved Space*. Cambridge Monographs on Mathematical Physics. Cambridge University Press, 1984. ISBN 9780521278584. URL `http://books.google.se/books?id=SEnaUnrqzrUC`.

Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, October 2001. ISSN 0885-6125. doi: 10.1023/A:1010933404324. URL `http://dx.doi.org/10.1023/A:1010933404324`.

Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *Proceedings of the 22Nd International Conference on Machine Learning*, ICML '05, pages 89–96, New York, NY, USA, 2005. ACM. ISBN 1-59593-180-5. doi: 10.1145/1102351.1102363. URL `http://doi.acm.org/10.1145/1102351.1102363`.

Christopher J. C. Burges, Robert Ragno, and Quoc V. Le. Learning to Rank with Nonsmooth Cost Functions. In Bernhard Schölkopf, John C. Platt, Thomas Hoffman, Bernhard Schölkopf, John C. Platt, and Thomas Hoffman, editors, *NIPS*, pages 193–200. MIT Press, 2006. ISBN 0-262-19568-2.

Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. Learning to rank: From pairwise approach to listwise approach. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pages 129–136, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-793-3. doi: 10.1145/1273496.1273513. URL `http://doi.acm.org/10.1145/1273496.1273513`.

Ben Carterette. Precision and recall. In LING LIU and M.TAMER ÖZSU, editors, *Encyclopedia of Database Systems*, pages 2126–2127. Springer US, 2009. ISBN 978-0-387-35544-3. doi: 10.1007/978-0-387-39940-9_5050. URL `http://dx.doi.org/10.1007/978-0-387-39940-9_5050`.

Olivier Chapelle, Yi Chang, and Tie-Yan Liu, editors. *Proceedings of the Yahoo! Learning to Rank Challenge, held at ICML 2010, Haifa, Israel, June 25, 2010*, volume 14 of *JMLR Proceedings*, 2011. JMLR.org. URL `http://jmlr.org/proceedings/papers/v14/`.

David Cossock and Tong Zhang. Subset ranking using regression. In *Proceedings of the 19th Annual Conference on Learning Theory*, COLT'06, pages 605–619, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-35294-5, 978-3-540-35294-5. doi: 10.1007/11776420_44. URL `http://dx.doi.org/10.1007/11776420_44`.

Koby Crammer and Yoram Singer. Pranking with ranking. In *Advances in Neural Information Processing Systems 14*, pages 641–647. MIT Press, 2001.

Van Dang. RankLib. Online, 2011. URL `http://www.cs.umass.edu/~vdang/ranklib.html`. [Online; accessed 14-April-2015].

Yoav Freund, Raj Iyer, Robert E. Schapire, and Yoram Singer. An efficient boosting algorithm for combining preferences. *J. Mach. Learn. Res.*, 4: 933–969, December 2003. ISSN 1532-4435. URL `http://dl.acm.org/citation.cfm?id=945365.964285`.

Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000.

Starr R. Hiltz and Murray Turoff. Structuring computer-mediated communication systems to avoid information overload. *Commun. ACM*, 28 (7):680–689, July 1985. ISSN 0001-0782. doi: 10.1145/3894.3895. URL `http://doi.acm.org.proxy.lib.chalmers.se/10.1145/3894.3895`.

Thorsten Joachims. Optimizing search engines using clickthrough data. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, pages 133–142, New York, NY, USA, 2002. ACM. ISBN 1-58113-567-X. doi: 10.1145/775047.775067. URL `http://doi.acm.org/10.1145/775047.775067`.

Amy N. Langville and Carl D. Meyer. *Google's PageRank and Beyond: The Science of Search Engine Rankings.* Princeton University Press, Princeton, NJ, USA, 2006. ISBN 0691122024.

Charles L. Lawson and Richard J. Hanson. *Solving least squares problems.* Classics in applied mathematics. SIAM, Philadelphia (Pa.), 1995. ISBN

0-89871-356-0. URL http://opac.inria.fr/record=b1080804. SIAM : Society of industrial and applied mathematics.

H. Li. *Learning to Rank for Information Retrieval and Natural Language Processing: Second Edition*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2014. ISBN 9781627055857. URL http://books.google.se/books?id=lModBQAAQBAJ.

Ping Li, Qiang Wu, and Christopher J. Burges. Mcrank: Learning to rank using multiple classification and gradient boosting. In J.C. Platt, D. Koller, Y. Singer, and S.T. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 897–904. Curran Associates, Inc., 2008. URL http://papers.nips.cc/paper/3270-mcrank-learning-to-rank-using-multiple-classification-and-gradient-boosting.pdf.

Tie-Yan Liu. *Learning to Rank for Information Retrieval*. Springer, 2011. ISBN 978-3-642-14266-6. doi: 10.1007/978-3-642-14267-3. URL http://dx.doi.org/10.1007/978-3-642-14267-3.

Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008. ISBN 0521865719, 9780521865715.

Massimo Melucci. Vector-space model. In LING LIU and M.TAMER ÖZSU, editors, *Encyclopedia of Database Systems*, pages 3259–3263. Springer US, 2009. ISBN 978-0-387-35544-3. doi: 10.1007/978-0-387-39940-9_918. URL http://dx.doi.org/10.1007/978-0-387-39940-9_918.

Donald Metzler and W. Bruce Croft. Linear feature-based models for information retrieval. *Inf. Retr.*, 10(3):257–274, June 2007. ISSN 1386-4564. doi: 10.1007/s10791-006-9019-z. URL http://dx.doi.org/10.1007/s10791-006-9019-z.

Tao Qin and Tie-Yan Liu. Introducing letor 4.0 datasets. *CoRR*, abs/1306.2597, 2013. URL http://dblp.uni-trier.de/db/journals/corr/corr1306.html#QinL13.

Tao Qin, Tie-Yan Liu, Jun Xu, and Hang Li. Letor: A benchmark collection for research on learning to rank for information retrieval. *Inf. Retr.*, 13(4): 346–374, August 2010. ISSN 1386-4564. doi: 10.1007/s10791-009-9123-y. URL http://dx.doi.org/10.1007/s10791-009-9123-y.

S. E. Robertson. The Probability Ranking Principle in IR. *Journal of Documentation*, 33(4):294–304, 1977. doi: 10.1108/eb026647. URL `http://dx.doi.org/10.1108/eb026647`.

Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: Bm25 and beyond. *Found. Trends Inf. Retr.*, 3(4):333–389, April 2009. ISSN 1554-0669. doi: 10.1561/1500000019. URL `http://dx.doi.org/10.1561/1500000019`.

Robert E. Schapire. A brief introduction to boosting. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'99, pages 1401–1406, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. URL `http://dl.acm.org/citation.cfm?id=1624312.1624417`.

Ming-Feng Tsai, Tie-Yan Liu, Tao Qin, Hsin-Hsi Chen, and Wei-Ying Ma. Frank: a ranking method with fidelity loss. In *SIGIR 2007: Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Amsterdam, The Netherlands, July 23-27, 2007*, pages 383–390, 2007. doi: 10.1145/1277741.1277808. URL `http://doi.acm.org/10.1145/1277741.1277808`.

Qiang Wu, Christopher J. Burges, Krysta M. Svore, and Jianfeng Gao. Adapting boosting for information retrieval measures. *Inf. Retr.*, 13(3):254–270, June 2010. ISSN 1386-4564. doi: 10.1007/s10791-009-9112-1. URL `http://dx.doi.org/10.1007/s10791-009-9112-1`.

Jun Xu and Hang Li. Adarank: A boosting algorithm for information retrieval. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '07, pages 391–398, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-597-7. doi: 10.1145/1277741.1277809. URL `http://doi.acm.org/10.1145/1277741.1277809`.