CA' FOSCARI UNIVERSITY OF VENICE

DEPARTMENT OF
ENVIRONMENTAL SCIENCES, INFORMATICS AND
STATISTICS

Master's Thesis in
Computer Science

Pruning strategies for Additive Ranking Models

Supervisor: Prof. Salvatore Orlando
Co-Supervisor: PhD Franco Maria Nardini
Co-Supervisor: PhD Salvatore Trani

Candidate:
Francesco Busolin, 851884

Academic Year 2020 - 2021

Abstract

The majority of current commercial web search engines rely on additive machine learned models for ranking web documents. The effort required to evaluate each document on all the sub models that compose the main one is directly tied to the responsivity of the system as a whole by influencing the query response time. In this thesis we explore some strategies which are query dependant and aim to stop evaluating documents unlikely to be relevant. The strategies are evaluated using an ensemble of regression trees as ranking model trained and tested over the well known MSLR dataset. We show that we can achieve speed-ups of over 3x with almost no loss in result quality, evaluated using the NDCG@10 index.

Contents

List of Algorithms

List of Tables

List of Figures

Preface

Chapter 1 serves as a general introduction. It starts by giving a brief overview on the topic of web search to then move the focus towards ranking and, in particular, Learning To Rank.

Chapter 2 is devoted to explain the problem of ranking and to give a description of some key techniques and models used in the field of Learning To Rank, putting a particular stress on ensembles of tree-based models.

Chapter 3 introduces two different approaches to reduce the cost of ranking. Firstly, we discuss how to efficiently use large tree ensembles to rank. Then, in the second part of the chapter, we describe a series of strategies presented by B. Barla Cambazoglu et al. in [18] to prune from the document collection some of the likely uninteresting candidates.

Chapter 4 describes the work done for this thesis. In particular we explain how to modify some of the pruning strategies to make them adapt to the different distributions of document scores, how we added the possibility of performing Document Early Exit in the ranking tool QuickScorer and the experiments done to assess the validity of this new approach.

Chapter 5 is dedicated to the analysis and the discussions of the results obtained.

Finally, in chapter 6 we will review and complete the discussion and give some ideas for possible future developments.

Chapter 1

Introduction

The Internet, and particularly the World Wide Web (WWW), is an enormous collection of heterogeneous data. So much so that nowadays it has become the *de-facto* standard place where anyone will look to satisfy any sort of information need they may have. However, this great repository of data as we know it today is substantially different from its initial concept. Originally, the Web was born as a distributed computer platform to share and link documents in a standardized way among experts. Instead, today the Web is regularly used by billions of users with widely different interests, needs and capabilities. For this reason, a large number of software tools have been developed over the years to facilitate the retrieval of information from the Web.

Typically, people try to satisfy their information needs, namely their intents, through a specific category of such softwares, called Web search engines. These have become the universally adopted interfaces between users and the whole information contained on the Web. As such, devising, implementing and improving search engines is an important aspect of the modern era.

As one would expect, search engines are complex products, composed of many different parts, each responsible for a particular task. Nevertheless, we can simplify a search engine into three main components: Crawler, Indexer and Ranker. The crawler is a simple piece of software which is responsible to traverse the net and download web documents. These are then parsed by the indexer which builds an index using the content (keywords) of each page. Then, when a user submits a query, the query processor will extract some keywords from it and use the index to retrieve a number of matching documents. Finally, the ranker sorts the retrieved documents by relevance before sending them back to the user as a list of URLs.

For the remaining part of this thesis we will focus on ranking, that

is to sort documents by their relevance to the user's query. Unfortunately, the true objective of ranking is not to find which documents are the most similar to the query content, but it is to promote documents that better than others satisfy the user's information need, and that is far from a trivial task. An important observation of this last statement is that we will not necessarily find the true relevance values of documents with respect to a query, but it will be sufficient to present the results in the right order, hence our aim is to find the right ordering of documents rather than their "true" relevance whatever it may be.

The majority of the modern ranking systems use a considerable number of parameters, all of which need to be fine-tuned to achieve acceptable ranking performance. Machine learning has been demonstrating its effectiveness in automatically tuning parameters, combining multiple evidences, and avoiding overfitting [22]. Therefore, it seems quite promising to adopt ML technologies to solve our problem. However, with the except of a few examples, like OPRF in 1989 [2] and SLR in 1992 [3], such methods historically found little ground in ranking or more generally in Information Retrieval tasks. This was due majorly to two factors, on one hand for a long time data was scarce and, outside academic settings, poorly descriptive of users' needs. Secondly, most of the early models are either non-parametric (e.g. Binary, Vector space and Latent semantic models) or use a very limited amount of parameters (e.g. BM25) and thus it is possible to tune them manually. Things progressively changed over time as the explosion of the internet increased exponentially the availability of data, more sophisticated models were introduced and ad-hoc techniques for Information Retrieval were presented. As such in the recent years a new research area emerged called "Learning To Rank" with the purpose of using Machine Learning tools to train ranking models.

Usually, modern ranking architectures are composed by a two-stage approach. The first stage retrieves from the index a reasonably large set of documents, usually by performing some sort of word matching with the query. This phase is aimed at maximizing the recall and is usually carried out by using a simple and fast ranking function. Learning To Rank models are then employed in the second phase to retrieve an even smaller subset of documents with the objective of maximizing the precision [22].

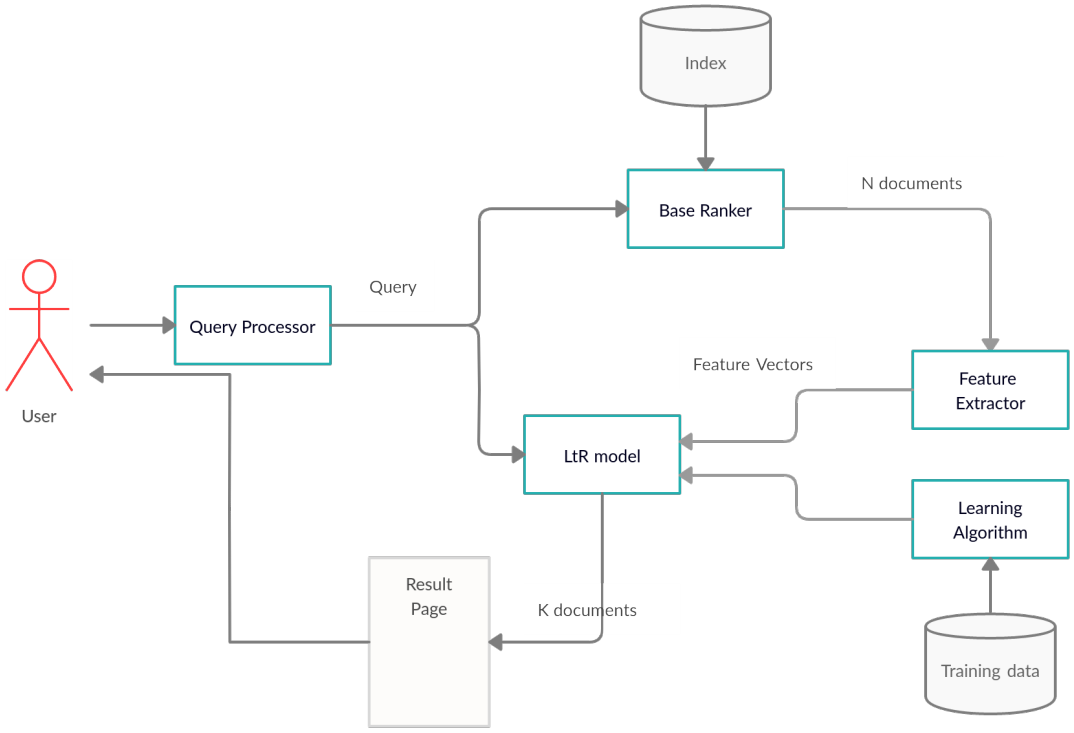


Figure 1.1: Graphical representation of the ranking process.

Chapter 2

Learning To Rank

Typically in ranking and even more so in learning to rank problems, we are not interested in documents nor queries alone but rather in the relationships between them, as such the common way to represent instances are the so called query-document pairs. These pairs are nothing more than feature vectors where the features can describe characteristics of the document, the associated query or something linking the two [21]. This representation allows us to view each pair as a point in $\mathbb{R}^d$, where d is the number of features.

The Learning to Rank algorithms and techniques that we are going to discuss are all types of supervised learning, which means that a perfectly known dataset is required for training. In our case it consists of a set of query-document pairs labelled by relevance (from 0 being irrelevant to 4 for absolutely relevant). An interesting point to note is that while it is true that each vector encompasses aspects of both query and document, in practice we can still group vectors by query since a typical entry of a dataset is something like [24]:

```
0 qid:1 1:3 2:0 3:2 4:2 … 135:0 136:0
```

Where the first value is the target label (relevance), then we have the query id and after that the list of features. As such, we can view these vectors improperly but without ambiguity as documents which are associated to a specified query. After this brief introduction, we can define more formally the concepts introduced so far.

2.1 Ranking Problem

Let $\mathcal{Q}$ be the set of Q queries and $\mathcal{D}$ the set of N documents. For each query q_i there is a set of documents $D_i = \{d_{i,1}, \cdots, d_{i,j}, \cdots, d_{i,n_i}\}$ with $D_i \subseteq \mathcal{D}$. A feature vector $x_{i,j} = \phi(q_i, d_{i,j})$ is crafted for each pair $(q_i, d_{i,j})$ so that $x_{i,j} \in \mathcal{X}$ and $\mathcal{X} \subseteq \mathbb{R}^d$ where d is the number of

features. As observed before, we can group feature vectors by query so that $\mathbf{x}_i = \{x_{i,1}, x_{i,2}, \cdots x_{i,n_i}\}$ is the set of vectors associated to q_i. As such, from now on, unless differently stated, we will use the terms document, instance and query-document pair interchangeably. Let the documents in $\mathbf{x}_i$ be identified by the integers $\{1, 2, \cdots, n_i\}$. A ranking π_i on $\mathbf{x}_i$ is a permutation of $\{1, 2, \cdots, n_i\}$ where $\pi_i(j)$ denotes the rank of document $x_{i,j}$. The goal is to train a function $F(\mathbf{x}_i) = \left[s_1, s_2, \cdots, s_{n_i}\right]$ which outputs a list of scores such that we have $\pi_i(j) < \pi_i(k) \iff s_j > s_k$. Generally, the model learned is local, which means that we learn a scorer $f(x)$ which takes as input a single feature vector. Doing so, the model described earlier can be expressed as $F(\mathbf{x}_i) = \left[f(x_{i,1}), \cdots, f(x_{i,n_i})\right]$.

To train F using a supervised learning method, a set of ground-truth labels is needed. Let $Y = \{\mathbf{y}_1, \cdots, \mathbf{y}_Q\}$ be a set of label vectors with $\mathbf{y}_i = \{y_{i,1}, \cdots, y_{i,n_i}\}$ the labels of q_i where $y_{i,j} \in \mathcal{Y} \subset \mathbb{N}$ represent the relevance of document $d_{i,j}$ with respect to the query. Define $S = \{(d_{i,j}, y_{i,j})\}$ or equivalently $S = \{(x_{i,j}, y_{i,j})\}$ as the training set. More concisely, $S = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{Q}$.

2.2 Ranking Quality

Since we assumed to have a labelled set of documents, we can suppose to have an optimal ranker $F^*(q_i) = \mathbf{y}_i$ which will result in the optimal ranking π_i^*. Thus, we can compare it with our ranking π_i and so measure the "distance" between the two. There exist a large number of possible measures to be used, in this work we primarily used the Normalized Discounted Cumulative Gain (NDCG).

NDCG

The NDCG, as the normalized in its name suggest, is a regularized version of another measure, the DCG or Discounted Cumulative Gain. The DCG is often used because it takes into account both the relevance and the position of documents in the result set, supporting the ones appearing at the top and penalizing those in the tail. Given a query q_i, a ranking π_i and the vector of labels $\mathbf{y_i}$. The DCG of q_i is defined as:

$$DCG(\mathbf{y}_i, \pi_i) = \sum_{j \in \pi_i} G(\mathbf{y}_i, j) D(\pi_i(j))$$

where $G(j)$ is the a measure of the Gain obtained by ranking a document in position j and takes into account its relevance y_{ij}. While $D(\pi)$ is the Discount which penalizes the measure based on the rank j. A possible practical implementation of the DCG is the following

$$DCG(\mathbf{y}_i, \pi_i) = \sum_{j \in \pi_i} \frac{2^{y_{i,j}} - 1}{\log_2(\pi_i(j) + 1)}$$

In order to normalize the DCG, we need a normalization factor, a perfect candidate for that is the so-called Ideal DCG, that is, the DCG that one would obtain if $\pi_i = \pi_i^*$. All things considered, we can define the NDCG as:

$$NDCG(\mathbf{y}_i, \pi_i, \pi_i^*) = \frac{DCG(\mathbf{y}_i, \pi_i)}{DCG(\mathbf{y}_i, \pi_i^*)}$$

Doing so, we will guarantee that $0 \leq NDCG \leq 1$.

NDCG@K Usually we are really interested only in the first k ranks, as such we can truncate the computation after k values:

$$DCG@K(\mathbf{y}_i, \pi_i) = \sum_{j \in \pi_i : \pi_i(j) \leq k} \frac{2^{y_{i,j}} - 1}{\log_2(\pi_i(j) + 1)}$$

And consequently:

$$NDCG@K(\mathbf{y}_i, \pi_i, \pi_i^*) = \frac{DCG@k(\mathbf{y}_i, \pi_i)}{DCG@k(\mathbf{y}_i, \pi_i^*)}$$

2.3 Simple Learning to Rank models

In the literature, there exist three main approaches to solve the ranking problem:

- **Pointwise**: Pointwise solutions work by casting the ranking problem into one of either regression or classification so to directly predict the right relevance of each document separately. Some examples of this category are McRank [14] for classification and Prank [6] for regression.
- **Pairwise**: Pairwise solutions work by predicting the relative order of documents taken in pairs, minimizing the number of pairwise errors. Belong to this category models such as RankNet [10], RankBoost [7], LambdaRank [11], MART [8] and LambdaMART [20].

- **Listwise**: Listwise solutions work by predicting an entire ranking sequence at once by optimizing a query-wise measure (i.e. NDCG) or by minimizing a loss function defined by considering all the scores in a given query. Examples of models using this approach are SoftRank [15], AdaRank [13] and ListNet [12].

Since the model used to score documents in our experiments is a LambdaMART ensemble [20] we will focus on some key models using a pairwise approach presented in the last couple of decades. Starting from RankNet, a model built on top of neural networks, up to ensembles of gradient boosted regression trees.

2.3.1 RankNet

RankNet is a ranking algorithm which uses an underlying model f to score documents, such that for any input feature vector $x \in \mathcal{X}$ it produces a number $\hat{y} = f(x) \in \mathbb{R}$. Such model can be anything for which the output is a differentiable function of the model parameters (originally RankNet used a neural network) [10]. The training process works as follow. Given that the problem we are tackling is ranking, it means that we have the training data partitioned in queries, as such for a given query, each possible pair of documents x_i and x_j is presented to the model which computes the scores $s_i = f(x_i)$ and $s_j = f(x_j)$. Let $x_i \prec x_j$ denote the event that x_i should be ranked higher that x_j or, in other words, that $\pi^*(x_i) < \pi^*(x_j) \implies \pi(x_i) < \pi(x_j)$. The outputs are then mapped into a probability of this event through a sigmoid function [10, 17]

$$P_{ij} \equiv P(x_i \prec x_j) = \frac{1}{1 + e^{-\sigma(s_i - s_j)}} = \frac{e^{\sigma(s_i - s_j)}}{1 + e^{\sigma(s_i - s_j)}} \tag{2.1}$$

where the parameter σ dictates the shape of the sigmoid. The use of a sigmoid is borrowed from the original neural network based version of RankNet [10]. As Loss function we use the cross entropy, penalizing the deviation of the model probabilities from the desired ones. Let $\bar{P}_{ij}$ be the known probability of $x_i \prec x_j$. The loss is then

$$L = -\bar{P}_{ij} \log P_{ij} - (1 - \bar{P}_{ij}) \log \left(1 - P_{ij}\right) \tag{2.2}$$

Let

$$S_{ij} = \begin{cases} 1, & \text{if } f(x_i) > f(x_j) \\ 0, & \text{if } f(x_i) = f(x_j) \\ -1, & \text{if } f(x_i) < f(x_j) \end{cases} \tag{2.3}$$

Since, during training, we know the desired ranking π^* we can compute $\bar{P}_{ij}$ [17]

$$\bar{P}_{ij} = \frac{1}{2}\left(1 + S_{ij}\right) \tag{2.4}$$

Merging these observations gives

$$L = \frac{1}{2}(1 - S_{ij})\sigma(s_i - s_j) + \log(1 + e^{-\sigma(s_i - s_j)}) \tag{2.5}$$

The loss is symmetric, that is, swapping the roles of x_i and x_j (and the sign of S_{ij}) does not change it:
If $S_{ij} = 1$ we have

$$L = \log(1 + e^{-\sigma(s_i - s_j)}) \tag{2.6}$$

while with $S_{ij} = -1$

$$L = \log(1 + e^{-\sigma(s_j - s_i)}) \tag{2.7}$$

An interesting point is that when $s_i = s_j$ the loss is $\log 2$, which implies that documents with different labels, but to which the model predicts the same score are still pushed away from each other. Moreover, as desirable, the loss is asymptotically zero when the ranking is correct [10, 17]. Since RankNet is trained using stochastic gradient descent [10], the next step is to compute the partial derivatives of the loss with respect to each of the parameters of the model. Starting from

$$\frac{\partial L}{\partial s_i} = \sigma\left(\frac{1}{2}(1 - S_{ij}) - \frac{1}{1 + e^{\sigma(s_i - s_j)}}\right) = -\frac{\partial L}{\partial s_j} \tag{2.8}$$

We can update each of the weights $w_k \in \mathbb{R}$ as

$$w_k \to w_k - \eta\frac{\partial L}{\partial w_k} = w_k - \eta\left(\sum_{ij}\frac{\partial L}{\partial s_i}\frac{\partial s_i}{\partial w_k} + \frac{\partial L}{\partial s_j}\frac{\partial s_j}{\partial w_k}\right) \tag{2.9}$$

where η is a positive, usually small, learning rate. The change in loss can be explicated as

$$\begin{aligned}\delta L &= \sum_k \frac{\partial L}{\partial w_k}\delta_k = \sum_k \frac{\partial L}{\partial w_k}\left(\eta\frac{\partial L}{\partial w_k}\right) \\ &= -\eta\sum_k\left(\frac{\partial L}{\partial w_k}\right)^2 < 0\end{aligned} \tag{2.10}$$

The idea of learning using gradient descent is found in many different settings, even where the desired loss does not have well-posed

gradients or even when the model itself has non-differentiable parameters (such as in boosted regression trees) [11, 8]. Let us now make some algebraic manipulations to make an observation regarding the gradient of the loss with respect to the parameters

$$\begin{aligned}\frac{\partial L}{\partial w_k} &= \sum_{ij}\left(\frac{\partial L}{\partial s_i}\frac{\partial s_i}{\partial w_k} + \frac{\partial L}{\partial s_j}\frac{\partial s_j}{\partial w_k}\right) \\ &= \sum_{ij}\sigma\left(\frac{1}{2}(1-S_{ij}) - \frac{1}{1+e^{-\sigma(s_i-s_j)}}\right)\cdot\left(\frac{\partial s_i}{\partial w_k} - \frac{\partial s_j}{\partial w_k}\right) \\ &= \sum_{ij}\lambda_{ij}\left(\frac{\partial s_i}{\partial w_k} - \frac{\partial s_j}{\partial w_k}\right)\end{aligned} \tag{2.11}$$

with

$$\lambda_{ij} = \sigma\left(\frac{1}{2}(1-S_{ij}) - \frac{1}{1+e^{-\sigma(s_i-s_j)}}\right) = \frac{\partial L(s_i - s_j)}{\partial s_i} \tag{2.12}$$

Let I denote the set of pairs of indices $\{(i,j)\}$ for which $\pi(i) \neq \pi(j)$ [17]. For convenience let's include in I only the pairs for which $x_i \prec x_j$, so that $S_{ij} = 1$. Now if we compute the gradient step for a parameter w_k we obtain [11]

$$\delta_k = -\eta\sum_{(i,j)\in I}\left(\lambda_{ij}\frac{\partial s_i}{\partial w_k} - \lambda_{ij}\frac{\partial s_j}{\partial w_k}\right) \equiv -\eta\sum_i \Lambda_i\frac{\partial s_i}{\partial w_k} \tag{2.13}$$

where Λ_i is computed by considering all j such that $(i,j) \in I$ and all the k for which $(k,i) \in I$. In the first case we increment Λ_i by λ_{ij}, in the second we decrement it by λ_{ki}. In general, we have

$$\Lambda_i = \sum_{j:(i,j)\in I}\lambda_{ij} - \sum_{j:(j,i)\in I}\lambda_{ij} \tag{2.14}$$

Intuitively, each Λ can be seen as a force pushing a document to a higher rank if positive and to a lower one if negative. This fact alone allows for a significant speedup in training time of RankNet, since it cuts the amount of weight updates required by accumulating the steps in the Λ's and only when all pairs of documents have been evaluated the weights are actually changed [11, 17].

2.3.2 LambdaRank

RankNet has been proven to work generally well, with the desirable characteristic of following a classic pairwise approach by optimizing

for (a smooth and convex approximation of) the number of pairwise errors [11, 17]. Unfortunately, such approach is not well translated to the optimization of other IR measures (such as the NDCG). Since most of those measures involve a sorting operation, then computing gradients may become problematic. The idea behind LambdaRank is to bypass the problem by giving the desired gradients directly[11].

Before, we have observed that the Λ's can be viewed as forces pushing documents up or down the order. Expanding on this idea, LambdaRank works by defining an implicit loss function L such that the Λ's are exactly the gradients $\frac{\partial L}{\partial s_i}$, without the need of computing the losses [11]. Empirically it was shown that modifying equation (2.12) by multiplying by the change in the target IR measure Z obtained by swapping the ranks of x_i and x_j gives excellent results[11, 17].

$$\lambda_{ij} = \frac{\partial L(s_i - s_j)}{\partial s_i} = \frac{-\sigma|\Delta Z_{ij}|}{1 + e^{\sigma(s_i - s_j)}} \tag{2.15}$$

resulting in

$$\begin{aligned} \frac{\partial L}{\partial w_k} &= \sum_{ij} \lambda_{ij} \left(\frac{\partial s_i}{\partial w_k} - \frac{\partial s_j}{\partial w_k} \right) \\ &= \sum_{ij} \frac{-\sigma|\Delta Z_{ij}|}{1 + e^{\sigma(s_i - s_j)}} \left(\frac{\partial s_i}{\partial w_k} - \frac{\partial s_j}{\partial w_k} \right) \end{aligned} \tag{2.16}$$

which, as expected, does not require the derivative of the loss with respect to the scores. Usually, for many measures, it is more natural to rephrase the minimization of a loss function into a maximization of an utility. As such, equation (2.9) becomes

$$w_k \to w_k + \eta \frac{\partial L}{\partial w_k} \tag{2.17}$$

leading to

$$\delta L = \frac{\partial L}{\partial w_k} \delta_k = \eta \left(\frac{\partial L}{\partial w_k} \right)^2 > 0 \tag{2.18}$$

This fact tells us that although IR measures, viewed as functions of the scores, are either flat or discontinuous everywhere [11], LambdaRank bypasses this issue by computing the gradients after the sorting of the scores. Furthermore, we did not specify which measure to optimize, suggesting that we are able to quickly adapt LambdaRank to work with any of them [11, 17].

2.3.3 Decision Trees

Two of state-of-the-art ranking models are Gradient Boosted Regression Trees (GBRT) [5] and LambdaMART [11], both of which are ensembles of decision trees. As such, it is useful to review and briefly discuss what decision trees are and how they work.

The idea of tree-based models is to partition the input space $\mathcal{X}$ into disjoint sets $R_1, \cdots, R_j, \cdots, R_J$. A possible strategy to achieve this is by performing successive binary splittings based on different features (as seen in figure 2.1). The final prediction of a sample $x \in R_j \subseteq \mathcal{X}$ will be the average of the responses in his partition, that is, $\hat{y}(x) = \frac{1}{|R_j|} \sum_{x_i \in R_j} y(x_i)$.

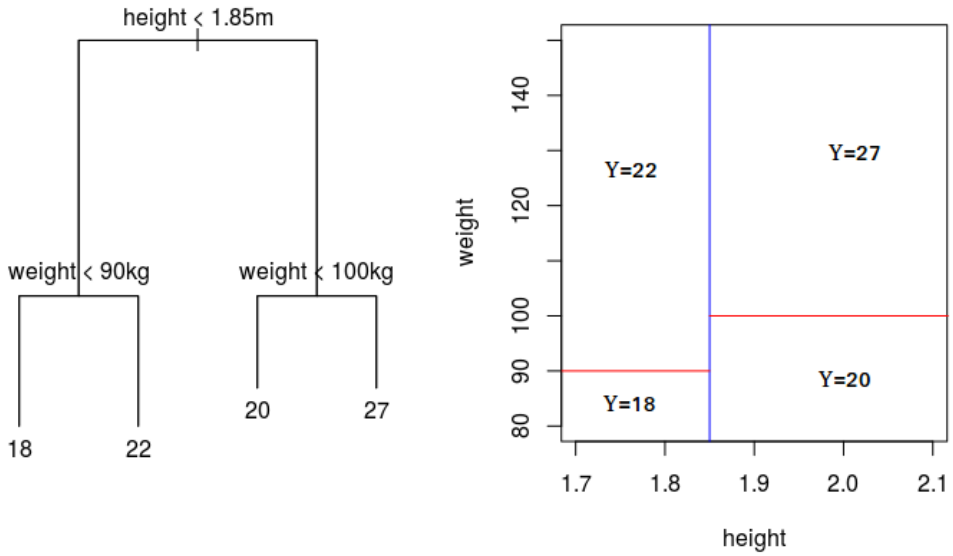


Figure 2.1: Example of regression tree with corresponding partitions. (`https://insightr.wordpress.com/2017/09/23/how-random-forests-improve-simple-regression-trees/`)

Since later we are going to use regression trees, let us focus on those. Suppose we have a scalar outcome $y \in \mathbb{R}$, and a feature vector $x \in \mathcal{X} \subseteq \mathbb{R}^d$ composed by d features, also called prediction variables. A regression tree partitions the $\mathcal{X}$ space into J disjoint regions R_j and provides a value $E(y|x \in R_j)$ for each of them. To build "good" trees, we can use CART, a greedy algorithm which, in general, works as follow[1]:

1. **Grow** an oversized tree using forward selection. At each step select the **best** split. Grow until all the leaves either

(a) have $< n$ data points, maybe even $n = 1$

(b) are "pure" (all points have [almost] the same output)

2. **Prune** the tree back, creating a nested sequence of trees, decreasing the overall complexity.

The problem in construction is now how to determine the the best split at each step. The key idea is to pick the split so that the data in the resulting descendants are "purer" then that in the parent. Thus, the greedy approach tells us to choose the split that leads to the greatest decrease in impurity [1, 5]. For regression trees, a common measure of impurity is the residual sum of squares:

$$D = RSS = \sum_{x_i \in u_j} \big(y(x_i) - \mu_j\big)^2 \tag{2.19}$$

where μ_j is the mean of the y's of all points belonging to the same node u_j as x_i. The growth proceeds until the aforementioned conditions are met or a custom stopping criterion is reached. Examples of simple stopping criteria are maximum number of leaves or a maximal height of the tree.

Algorithm 1: CART growth [1]

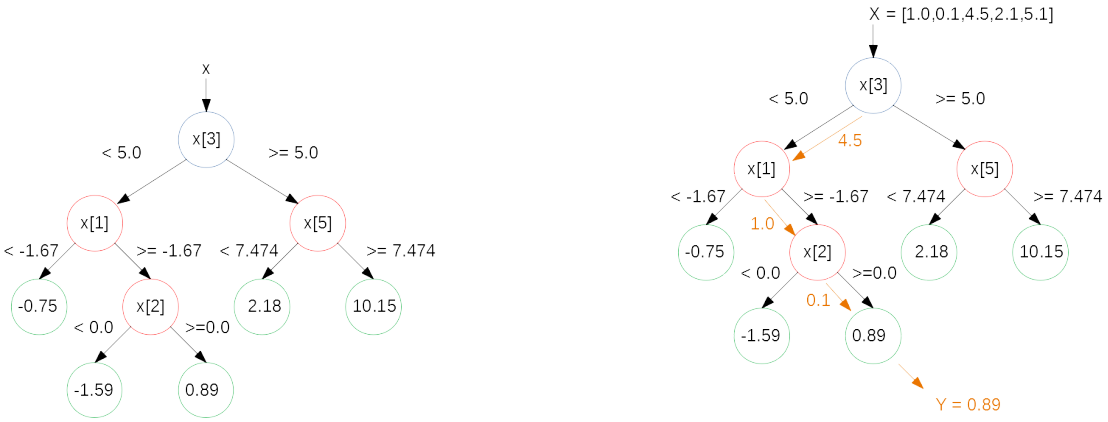
Data: **S**: stopping criterion
$T = \{root\}$;
Set all samples in the root node ;
while ***not*** S **do**
    find best allowed split ;
    divide the leaf according to split ;
    split the samples in the node into the two new leaves ;
end

After building the initial tree, we need to reduce the complexity by pruning some of its subtrees. This is necessary because decision trees that are too large are susceptible to overfitting [16]. Pruning attempts to improve the generalization capability of a decision tree by trimming the branches of the initial tree, it is done by following the steps below:

1. Start from a big tree T
2. Consider all rooted subtrees $\hat{T}_k$
3. Let L_{ik} be the impurity at leaf i in $\hat{T}_k$
4. Define $L_k = \sum_i L_{ik}$ as the impurity of $\hat{T}_k$

5. Let σ_k be the number of leaves in $\hat{T}_k$
6. Let $L_\alpha(T_k) = L_k + \alpha \cdot \sigma_k$

The final tree T_α, result of CART, is the nested set S of rooted subtrees of T minimizing $L_\alpha = \sum_{k \in S} R_\alpha(\hat{T}_k)$ [16].



(a) Example of regression tree

(b) Example of a traversal

Figure 2.2: Example of the result of CART.

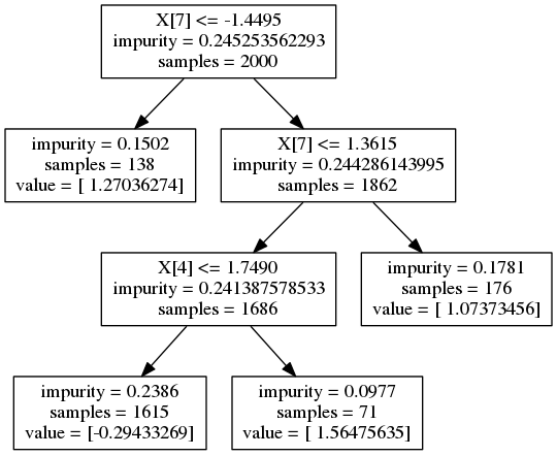


Figure 2.3: Example of regression tree with specified the level of impurity in each node.
(`https://www.add-for.com/ensemble-methods-gradient-boosted-trees/`)

Trees are a popular choice for a variety of different problems thanks to some convenient properties: are fast to use, features can be of heterogeneous type being numerical or categorical, they are resistant to outliers and there is an automatic intrinsic feature selection since not all features will be used as prediction variables in the internal nodes. However, they come also with some disadvantages, for example, as mentioned before, they are prone to overfitting due to the fact that each layer of a tree is essentially another AND predicate in a chain which may result in an overly specific sentence [1,

16]. On the other hand, trees can be inherently inaccurate if not complex enough, those are the main reasons why they are almost always used in ensembles where these weaknesses can be mitigated by distributing the complexity among different trees [1, 8].

2.4 Ensemble Models

Many machine-learned ranking architectures are based on ensembles of models, where many scorers are executed sequentially in a chain and score contributions of individual scorers are accumulated to compute the final document prediction. For this reason, we will first give an overview of the general idea of ensemble models, how it is translated to regression trees and then we will focus on how to train them using Boosting and in particular Gradient Boosting.

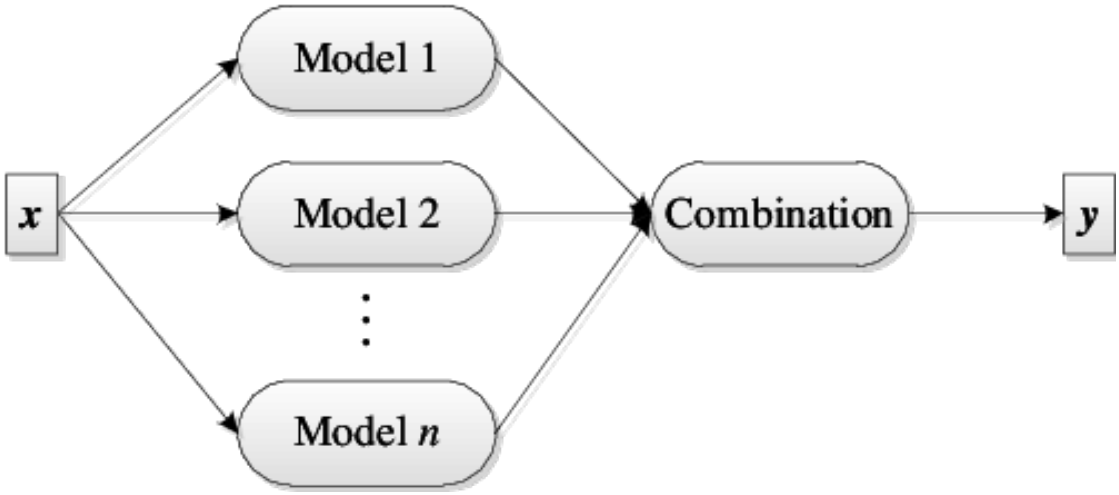


Figure 2.4: A typical ensemble architecture[27]

An Ensemble is a cooperation of a group of independent models whose individual results are combined to create a final unique prediction, hopefully more accurate than each single model prediction. In other words, an ensemble is a set of individually trained models whose outputs are combined when classifying new instances. A popular kind of ensembles are *additive ensembles* where the final result is a linear combination of the individual predictions (in case of classification is a weighted voting) [8, 17]. Let $M = \{m_1, \cdots, m_n\}$ be an additive ensemble of n models and $\Theta = \{\theta_1, \cdots, \theta_n\}$ a set of n multipliers. The score given by the ensemble M to a instance x is

$$M(x) = \theta_1 m_1(x) + ... + \theta_n m_n(x) = \sum_{i=1}^{n} \theta_i m_i(x) \tag{2.20}$$

2.4.1 Boosting

Boosting refers to a general technique to combine a set of weak learners[1] (i.e., shallow regression trees) into an ensemble to build a more reliable and performant model. Boosting is based on the assumption that training (or craft) many weak learners is easier that training a single powerful predictor [9]. The general procedure works by repeating for M boosting rounds a training algorithm, building each time a new model trained upon a weighted dataset where each sample weight is proportional to the mispredictions of that sample in the previous round, in this way we emphasize those which are hard to predict correctly. Finally, we will have an ensemble where each component tries to rectify the mistakes of the others. The final result will be again a weighted sum of the weak predictions.

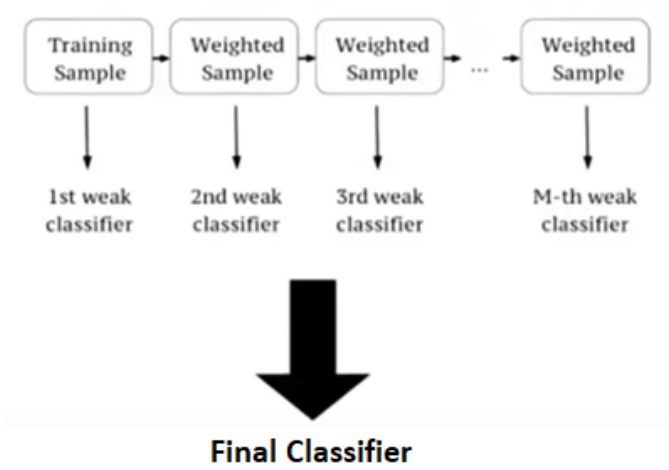


Figure 2.5: Boosting Overview.
(`https://www.geeksforgeeks.org/ml-xgboost-extreme-gradient-boosting/`)

To give a practical example of how a boosting model is trained let's review one of the first of such models, AdaBoost [9]. In its initial formulation, AdaBoost was designed for a generic binary classification problem. Given a training set $S = \{(x_1, y_1), \cdots, (x_m, y_m)\}$ with $x_i \in X$ and $y_i \in Y = \{-1, +1\}$. Set a number of rounds M. For each example in S set a weight $D_t(i)$ being the one given to example i in round m, for the first round set all equal weights. With these premises, we can summarize AdaBoost in algorithm 2.

[1] simple models whose performances (accuracy) are at least better than chance

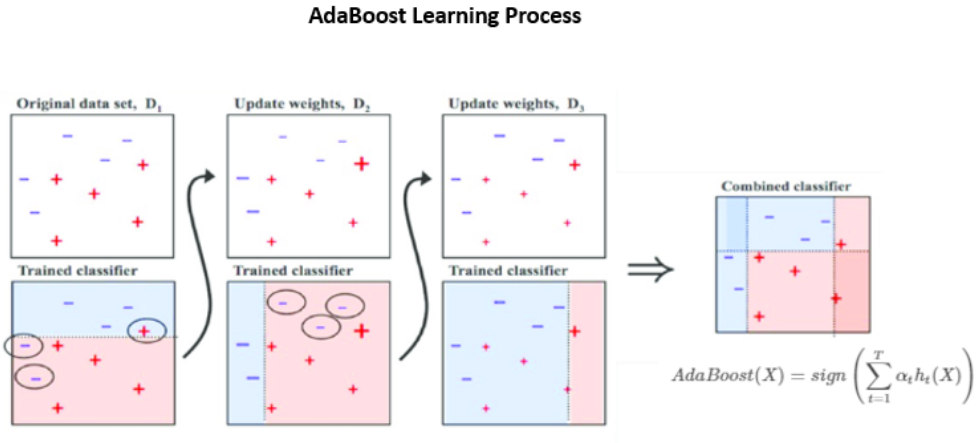


Figure 2.6: Visualization of the AdaBoost training process. (`https://www.pluralsight.com/guides/ensemble-methods:-bagging-versus-boosting`)

Algorithm 2: AdaBoost

foreach $i \in \{1, \dots, m\}$ **do**
  $D_1(i) = \frac{1}{m}$
end
foreach $m \in \{1, \cdots, M\}$ **do**
  train a weak classifier h_m on S with weights D_m ;
  choose (or learn) a weight α_m ;
  update the examples weights:

$$D_{m+1}(i) = \frac{D_m(i) e^{-\alpha_m y_i h_m(x_i)}}{Z_m}$$

  with Z_m normalization factor so that $\sum_i D_{m+1}(i) = 1$;
end
build the final classifier H(x) = sign$\left(\sum_m \alpha_m h_m(x)\right)$;

The choice of the weights α's, also called expansion multipliers [8], is done to give a measure of importance to each learner. In the AdaBoost paper, a suggested value for binary classifiers is

$$\alpha_m = \frac{1}{2} \log\left(\frac{1 - \epsilon_m}{\epsilon_m}\right)$$

where ϵ_m is the error of the classifier[4]:

$$\epsilon_m = Pr_{i \sim D_i}(h_m(x_i) \neq y_i)$$

However, as we will see, those weights could be also part of the training process [8, 20].

Gradient Boosting

Gradient boosting is a variant of the general boosting method used to perform gradient descent (or ascend) or otherwise hard to optimize loss (or utility) functions. To better formulate this idea, consider now the problem of function estimation. In such problems, starting from a training set of known (x, y) pairs, one would like to obtain a function $F^*(x)$ that maps x to y such that the chosen loss function $L(\mathbf{y}, F(\mathbf{x}))$ is minimized [5]:

$$F^*(\mathbf{x}) = \arg\min_{F(\mathbf{x})} E_{\mathbf{y},\mathbf{x}} L(y, F(\mathbf{x})) \tag{2.21}$$

Boosting, as seen in (2.20), finds an additive approximation F in the form:

$$F(x) = \sum_m \alpha_m h_m(x, \mathbf{a}_m) \tag{2.22}$$

where the function $h_m(x, \mathbf{a}_m)$ is the weak learner obtained in the m-th round and $\mathbf{a}_m$ are the parameters of that model. Both the expansion multipliers α's and the parameters $\mathbf{a}$'s are learned from the training data. At each step m of the boosting process, we augment the overall model as such:

$$F_m(x) = F_{m-1}(x) + \alpha_m h_m(x, \mathbf{a}_m) \tag{2.23}$$

Gradient boosting finds an approximated solution of the minimization problem in two steps. First, we fit $h(x, \mathbf{a})$ using a least-squares rule:

$$\mathbf{a}_m = \arg\min_{\mathbf{a},\rho} \sum_i (\bar{y}_{im} - \rho \cdot h(x_i, \mathbf{a}))^2 \tag{2.24}$$

where $\bar{y}$ are called pseudo-residuals [8]:

$$\bar{y}_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \tag{2.25}$$

Then the multiplier α is computed:

$$\alpha_m = \arg\min_{\alpha} \sum_i L(y_i, F_{m-1}(x_i) + \alpha h_m(x, \mathbf{a}_m)) \tag{2.26}$$

In this way, the original minimization problem (2.21) is replaced by a likely simpler least-square formulation (2.24) followed by a single parameter optimization (2.26).

2.4.2 MART

MART, which stands for Multiple Additive Regression Trees [8], is a gradient boosting algorithm which, as the name suggests, uses regression trees. Similarly to what defined in equation (2.22), a MART model can be summarized as:

$$F_M(x) = \sum_{i=1}^{M} \alpha_i f_i(x) \tag{2.27}$$

where each $f_i(x)$ is a function modeled by the ith regression tree and α_i is the weight associated with it.
Let

$$I(p) = \begin{cases} 1, & \text{if } p \text{ is } \textbf{true} \\ 0, & \text{otherwise} \end{cases} \tag{2.28}$$

So that each tree $T_i = (X, \{R_{jm}\}_1^J)$ gives a score

$$f_i(x) = \sum_{j=1}^{J} \bar{y}_{jm} I(x \in R_{jm}) \tag{2.29}$$

where $\bar{y}_{jm} = mean_{x_i \in R_{jm}}(\bar{y}_{im})$ is the mean of the pseudo-residuals (2.25) in each region R_{jm} at iteration m. The parameters $\mathbf{a}_m$, first seen in (2.22), are the regions of the current tree. With trees, the minimization problem (2.26) can be solved separately within each region R_{jm} defined by the corresponding leaf node j of the mth tree [8]. Since trees predict a constant value for each leaf, the solution to problem (2.21) is simplified to

$$\gamma_{jm} = \arg\min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x) + \gamma) \tag{2.30}$$

In MART, the hyper-parameters that need to be chosen before training are: the number of trees, the number of leaves and a positive fixed learning rate η. MART trains the trees sequentially using gradient descent [8]. Suppose that $m-1$ trees are already being trained, the mth tree will model the n derivatives of the loss with respect to the current model score evaluated at each training sample: $\frac{\partial L}{\partial F_{m-1}}(x_i),\ i = 1 \cdots, N$. Giving:

$$\delta L \approx \frac{\partial L(F_{m-1})}{\partial F_{m-1}} \delta F \tag{2.31}$$

if we take $\delta F = -\eta \frac{\partial L(F_{m-1})}{\partial F_{m-1}}$ then $\delta L < 0$. The new model is then:

$$F_m(x_i) = F_{m-1}(x_i) + \eta \sum_{k=1}^{L} \gamma_{km} I(x_i \in R_{km})$$

To better appreciate how to combine the ideas of MART and LambdaRank to obtain LambdaMART, which is to be discussed later in section 2.4.3, let us examine how MART actually works in a simple binary classification problem [17]. Let the labels be $Y = \{-1, +1\}$. Denote the conditional probabilities $P_+ \equiv P(y = 1|x)$ and $P_- \equiv P(y = -1|x)$. For convenience, define also two indicator functions:

$$I_+(x_i) = \begin{cases} 1, & \text{if } y_i = +1 \\ 0, & \text{if } y_i = -1 \end{cases} \tag{2.32}$$

and

$$I_-(x_i) = \begin{cases} 1, & \text{if } y_i = -1 \\ 0, & \text{if } y_i = +1 \end{cases} \tag{2.33}$$

As Loss function, we use the cross-entropy:

$$L(y, F) = -I_+ \log P_+ - I_- \log P_- \tag{2.34}$$

Thus, if $F(x)$ is the model output, we can compute both probabilities as

$$P_+ = \frac{1}{1 + e^{-2\sigma F(x)}}, \quad P_- = 1 - P_+ = \frac{1}{1 + e^{2\sigma F(x)}} \tag{2.35}$$

which results in

$$L(y, F) = \log\left(1 + e^{-2y\sigma F}\right) \tag{2.36}$$

The gradients of the loss with respect to the scores are the pseudo-residuals defined in gradient boosting:

$$\bar{y}_i = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} = \frac{2y_i\sigma}{1 + e^{2y_i\sigma F_{m-1}(x_i)}} \tag{2.37}$$

Given that we are using regression trees, our goal is to find the (approximate) optimal step for each leaf, that is

$$\gamma_{jm} = \arg\min_{\gamma} \sum_{x_i \in R_{jm}} \log(1 + e^{-2y_i\sigma(F_{m-1}(x_i)+\gamma)}) \equiv \arg\min_{\gamma} g(\gamma) \tag{2.38}$$

Using Newton's approximation, for a function $g(\gamma)$ a step towards an extremum of g is

$$\gamma_{n+1} = \gamma_n - \frac{g'(\gamma_n)}{g''(\gamma_n)} \tag{2.39}$$

Starting from $\gamma = 0$ we want to compute

$$\arg\min_{\gamma} g(\gamma) = \arg\min_{\gamma} \sum_{x_i \in R_{jm}} \log S_i(\gamma) \tag{2.40}$$

with $S_i(\gamma) = 1 + e^{-2v_i} = 1 + e^{-2y_i\sigma(F_{m-1}(x_i)+\gamma)}$.
Expanding from this we have

$$\begin{aligned} g' &= \sum_{x_i \in R_{jm}} \frac{1}{S_i}\left(-2y_i\sigma e^{-2v_i}\right) \\ g'' &= \sum_{x_i \in R_{jm}} \frac{-1}{S_i^2}(-2y_i\sigma e^{-2v_i})^2 - \frac{2y_i\sigma}{S_i}(-2y_i\sigma)e^{-2v_i} \\ &= \sum_{x_i \in R_{jm}} \frac{4}{S_i^2} y_i^2\sigma^2 e^{-2v_i} \end{aligned} \tag{2.41}$$

However,

$$\bar{y}_i = \frac{2y_i\sigma}{1 + e^{2y_iF}} \implies g' = \sum_{x_i \in R_{jm}} -\frac{2y_i\sigma}{e^{2v_i}S_i} = \sum_{x_i \in R_{jm}} -\bar{y}_i \tag{2.42}$$

And

$$g'' = \sum_{x_i \in R_{jm}} \frac{4y_i^2\sigma^2}{(1 + e^{2v_i})^2} e^{2v_i} \tag{2.43}$$

Given that $y_i^2 = +1$ we know that

$$|\bar{y}_i| = \frac{2\sigma}{1 + e^{2v_i}} \tag{2.44}$$

so

$$|\bar{y}_i|(2\sigma - |\bar{y}_i|) = \frac{4\sigma^2 e^{2v_i}}{(1 + e^{2v_i})^2} \tag{2.45}$$

Ultimately resulting in

$$\gamma_{jm} = -\frac{g'}{g''} = \frac{\sum_{x_i \in R_{jm}} \bar{y}_i}{\sum_{x_i \in R_{jm}} |\bar{y}_i|(2\sigma - |\bar{y}_i|)} \tag{2.46}$$

One last consideration is that we no longer need the α's since that information is now encapsulated inside the γ's which, as described

above, can be computed directly in some cases or be approximated via Newton's approximation in the others. This simplifies the overall formulation to just

$$F_M(x) = \sum_{i=1}^{M} f_i(x)$$

Algorithm 3: MART

Data: **M**: number of rounds; **N**: number of samples

$F_0 = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$;

foreach $m \in \{1, \cdots M\}$ **do**

$\bar{y}_{im} = -\left[\frac{\partial L(y, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}, \quad i = 1, N$;

$\{R_{jm}\}_1^J = \text{new Tree } (\{\bar{y}_{im}, x_i\}_1^N)$;

$\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma)$;

$F_m(x) = F_{m-1}(x) + \eta \sum_j \gamma_{jm} I(x \in R_{jm})$;

end

2.4.3 LambdaMART

In this section, we will describe how to modify MART in order to be able to give a value for each of the γ's depending only on the scores and the relative ordering of all document pairs, thus making the computation of gradients completely implicit. We will do that by adapting the ideas described in section 2.3.2 regarding LambdaRank into MART to obtain LambdaMART.

To define each step γ, in MART we need to compute the pseudo residuals $\bar{y}$, which in our case will be the Λ's. Recalling from equation (2.12) we know that

$$\lambda_{ij} = \frac{-\sigma |\Delta Z_{ij}|}{1 + e^{\sigma(s_i - s_j)}} \tag{2.47}$$

also

$$\Lambda_i = \sum_{j:(i,j)\in I} \lambda_{ij} - \sum_{j:(j,i)\in I} \lambda_{ij} \tag{2.48}$$

To simplify the notation, define

$$\Lambda_i \equiv \sum_{(i,j)\rightleftharpoons I} \lambda_{ij} = \sum_{j:(i,j)\in I} \lambda_{ij} - \sum_{j:(j,i)\in I} \lambda_{ij} \tag{2.49}$$

Thus, we can define an implicit utility function L for which Λ_i is its derivative

$$L = \sum_{(i,j)\rightleftharpoons I} |\Delta Z_{ij}| \log\left(1 + e^{-\sigma(s_i - s_j)}\right) \tag{2.50}$$

so that

$$\frac{\partial L}{\partial s_i} = \sum_{(i,j)\rightleftharpoons I} \frac{-\sigma|\Delta Z_{ij}|}{1+e^{\sigma(s_i-s_j)}} \equiv \sum_{(i,j)\rightleftharpoons I} -\sigma|\Delta Z_{ij}|\rho_{ij} \tag{2.51}$$

with[2]

$$\rho_{ij} = \frac{1}{1+e^{\sigma(s_i-s_j)}} = \frac{-\lambda_{ij}}{\sigma|\Delta Z_{ij}|} \tag{2.52}$$

So that

$$\frac{\partial^2 L}{\partial s_i^2} = \sum_{(i,j)\rightleftharpoons I} \sigma^2 \rho_{ij}(1-\rho_{ij})|\Delta Z_{ij}| \tag{2.53}$$

Putting everything together we can compute the step for each leaf. Given a leaf k the gradient step at round m for that leaf is [20]

$$\gamma_{km} = \frac{\sum_{x_i \in R_{km}} \frac{\partial L}{\partial s_i}}{\sum_{x_i \in R_{km}} \frac{\partial^2 L}{\partial s_i^2}} \tag{2.54}$$

Finally we have

$$\gamma_{km} = \frac{-\sum_{x_i \in R_{km}} \sum_{\{i,j\}\rightleftharpoons I} |\Delta Z_{ij}|\rho_{ij}}{\sum_{x_i \in R_{km}} \sum_{\{i,j\}\rightleftharpoons I} |\Delta Z_{ij}|\sigma\rho_{ij}(1-\rho_{ij})} \tag{2.55}$$

All the above allows to compute the value of any given leaf in the ensemble based only on scores or pairwise changes in the target IR measure up to that point, without the need to perform any derivative.

One of the perks of LambdaMART is that, since each tree models the Λ_i for the entire dataset and not just for a single query [20, 17], it is able to decrease the utility for some queries as long as the overall utility of the ensemble increases. This is possible since LambdaMART updates the weights only after all data has been examined, this means that although it changes only a few parameters at a time (the current leaves' values), those are influenced by all training samples, since all of them will land on some leaf of the current tree [20, 17].

[2] That is indeed correct since

$$\frac{\partial L}{\partial s_i} = \sum_{(i,j)\rightleftharpoons I} -\sigma|\Delta Z_{ij}|\rho_{ij} = \sum_{(i,j)\rightleftharpoons I} \frac{-\sigma|\Delta Z_{ij}|}{1+e^{\sigma(s_i-s_j)}} = \sum_{(i,j)\rightleftharpoons I} \lambda_{ij} = \Lambda_i$$

Algorithm 4: LambdaMART

Data: **M**: number of rounds; **N**: number of samples; **L**: number of leaves per tree

foreach $i \in \{1 \dots, N\}$ **do**
  $F_0(x_i) = \text{BaseModel}(x_i)$;
end
foreach $k \in \{1, \dots, M\}$ **do**
  foreach $i \in \{1, \dots, N\}$ **do**
    $y_i = \Lambda_i$;
    $w_i = \frac{\partial y_i}{\partial F_{k-1}(x_i)}$;
  end
  $\{R_{lk}\}_{l=1}^{L}$ // create L leaf tree on $\{x_i, y_i\}_{i=1}^{N}$;
  $\gamma_{lk} = \frac{\sum_{x_i \in R_{lk}} y_i}{\sum_{x_i \in R_{lk}} w_i}$;
  $F_k(x_i) = F_{k-1}(x_i) + \eta \sum_l \gamma_{lk} I(x_i \in R_{lk})$
end

Chapter 3

Faster Ranking

The ranking process is a demanding, yet invaluable, task for Web search engines, which on one hand need to maximize the quality of the response to the users' queries but on the other they also need to minimize the time required to process a query [18, 26]. As we discussed, many of the modern ranking models are based upon ensembles of regression trees [8, 20]. Such ensembles are typically composed of hundreds or even thousands of trees, all of which to be traversed at scoring time by any given document. As intuition suggests, it has been shown that such rankers are the most impacting component in terms of computational time, thus heavily influencing the latency and throughput of query processing. Therefore, devising techniques and strategies to speed up document ranking without losing in quality is definitely a hot research topic in web search [19, 23, 25, 26].

In this chapter we are going to discuss two different approaches, firstly we present various techniques of traversing trees and how we can optimize this task at low level by using a clever representation of each tree which provides a better usage and management of resources leading to a lower running time [26]. In the second part we are going to see how to sacrifice some potential quality in order to reduce significantly the overall amount of trees to traverse and thus how to find a good trade-off between speed and quality [18].

Let us now recall all the needed concepts from previous chapters: By the additive nature of ensembles such as MART or LambdaMART models, we know that the score of a document $x = \left[v_1, v_2, ..., v_d\right]$ composed by d features is

$$s(x) = F_M(x) = \sum_{i=1}^{M} w_i f_i(x) \tag{3.1}$$

where f_i is a scorer of an ensemble $\mathcal{M} = \{f_1, \cdots, f_M\}$ of M scorers evaluated in sequence and w_i is the weight associated with the ith

model. The result of ranking, π, is then a sorted sequence of the ids referring to the k highest scored documents. From now now we will suppose that $\mathcal{M}$ is an ensemble of decision trees, which means that each of them is composed by a set of internal nodes $I = \{u_0, u_1, ...\}$ and a set of leaves $L = \{l_0, l_1, ...\}$ enumerated in breadth-first order. Each $u \in I$ is associated with a Boolean test over a feature with id ϕ (meaning $v_\phi \in V$), and a constant threshold $\theta \in \mathbb{R}$. We assume that x is densely packed in a floating-point array. This means that performing the test at each node consists in a simple access to an array and a numerical comparison, being $x[\phi] < \theta$. Each leaf l_j stores a prediction value γ_j representing the potential contribution of the tree to the score of x. We denote as *false nodes* all of whose test is evaluated to false and *true nodes* the others [26]. In general, the traversal of a tree begins at its root and recursively proceeds towards the leaves by answering to the internal nodes' tests. If a node is a false one then the right branch is picked, otherwise the node is true and the left branch is the one taken. When we arrive at a leaf then its γ is returned. Such leaf is called an exit node and it is denoted as $e(x)$ [26]. The tree traversal is then repeated for all other trees in the ensemble. Using this slightly new notation, we can rewrite the equation (3.1) as

$$s(x) = F_M(x) = \sum_{i=1}^{M} w_i \cdot e_i(x).\gamma \tag{3.2}$$

where $e_i(x).\gamma$ is the value stored in the exit node of the ith tree.

3.1 Efficient Tree Traversals

In order to be able to process a large quantity of queries and, at the same time, guarantee a low response time to the users, the time budget available to the final ranking of candidate documents is reduced as much as possible [19, 26], therefore an ever increasing number of optimizations are being presented. In the following section we are going to discuss a particular kind of optimization regarding the traversal of decision trees, starting from a naive approach up to a modern algorithm based on bitvectors [26].

3.1.1 Naïve approach

A first and very basic traversal algorithm is a classic recursive procedure. Exactly as described earlier, the traversal starts from the root and moves down following a path to the leaves accordingly to the results of the Boolean conditions on the traversed nodes. This simple approach, however, is unnecessarily slow for a number of reasons. First, the next node to be processed is known only after the test is evaluated. This entails that the next instruction to be executed is not known, this induces frequent *control hazards*, i.e., instruction dependencies introduced by conditional branches [23, 26]. As a consequence, the efficiency of a code is strongly related to the branch misprediction rate of the processor. Finally, the traversal has low temporal and spatial locality, heavily hindering the cache hit ratio. This becomes more prominent when dealing with a large number of documents traversing a large ensemble of trees, since neither the documents nor the trees may fit in the cache[25, 26]. Although this last issue can be mitigated by preemptively storing the whole tree in a contiguous chunk of memory (STRUCT+ [23]), the former requires some more advanced techniques to reduce the number of mispredictions.

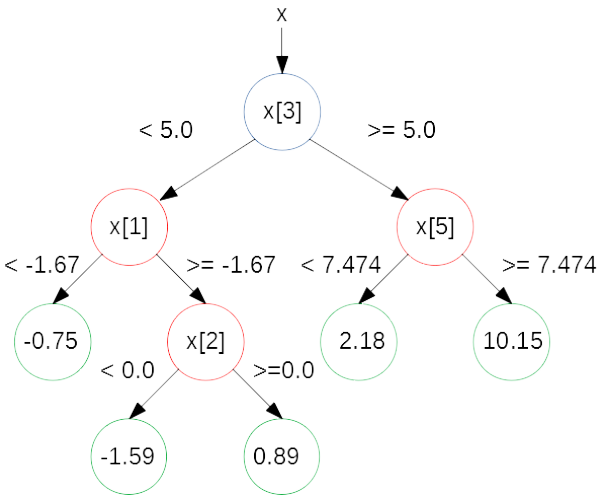
Algorithm 5: Naive Tree Scoring

```
def score(x: featureVector, u: treeNode):
    if u.isLeaf() then
        return u.γ;
    end
    else
        if x[u.φ] < u.θ then
            return score(x, u.left);
        end
        else
            return score(x, u.right);
        end
    end
```

3.1.2 IF-THEN-ELSE

Another basic, yet surprisingly performing, approach is to translate each tree into a nested sequence of conditional sentences. The resulting code is then compiled (i.e., in C) to generate the final scorer. This strategy will produce a static description of the tree and thus makes

the generated machine instructions more compact, allowing a better use of the instruction cache, significantly improving reference locality and, as by product, slightly reducing the branch mispredictions thanks to optimizations done by the compiler[23, 26].



(a) Example of regression tree

```
IF X[3] < 5.0 THEN
    IF X[1] < 1.67 THEN
        RETURN -0.75
    ELSE
        IF X[2] < 0.0 THEN
            RETURN -1.59
        ELSE
            RETURN 0.89
ELSE
    IF X[5] < 7.474 THEN
        RETURN 2.18
    ELSE
        RETURN 10.15
```

(b) Corresponding translation

Figure 3.1: Comparison of a graphical representation of a tree with its IF THEN ELSE translation

3.1.3 Prediction

A possible strategy to reduce branch mispredictions is to remove branches altogether. This idea, known as Prediction, is widely used in the field of compilers and it works by transforming control dependencies into data dependencies (i.e., an instruction needs the result of the previous one), removing jumps from the resulting assembly code [23]. In the context of tree traversal, Prediction is implemented by adding to each node, alongside the test feature id and threshold, an array **idx** containing a pair of indexes referring to the left and right child of the node respectively. Then the result of the comparison $x[\phi] > \theta$ is used directly as index of this array, retrieving the next node to process. Finally, the whole traversal loop is unrolled into h operations, where h is the height of the tree. In the end we achieve a simple sequence of operations [23]:

$$h \text{ steps} \begin{cases} j & \leftarrow u_0.\text{idx}[x[\phi_0] > \theta_0] \\ j & \leftarrow u_j.\text{idx}[x[\phi_j] > \theta_j] \\ & \vdots \\ j & \leftarrow u_j.\text{idx}[x[\phi_j] > \theta_j] \end{cases} \tag{3.3}$$

Leaf nodes are trivially encoded using placeholder tests which in turn will induce self-loops. At the end, the exit node of x is identified by j and its γ will be recovered using a lookup table. While Prediction clearly solves the problem of control dependencies, it inherently worsens the problem of data dependency. Furthermore, it adds a new considerable overhead for many samples since even if x reaches an early leaf, it will perform all subsequent steps anyway [23].

Vectorized Prediction

To reduce the impact of data dependencies, we can try to borrow a technique widely used in databases called vectorization. In practice, it consists in evaluating multiple vectors at once, in an interleaved fashion. Thus, while the processor is waiting for a memory access, we are able to start working on another sample. For example, say that we process 4 instances in parallel, the resulting instructions will look like [23]:

$$
\begin{aligned}
&\vdots \\
j_0 &\leftarrow u_{j_0}.\text{idx}[x[\phi_{j_0}] > \theta_{j_0}] \\
j_1 &\leftarrow u_{j_1}.\text{idx}[x[\phi_{j_1}] > \theta_{j_1}] \\
j_2 &\leftarrow u_{j_2}.\text{idx}[x[\phi_{j_2}] > \theta_{j_2}] \\
j_3 &\leftarrow u_{j_3}.\text{idx}[x[\phi_{j_3}] > \theta_{j_3}] \\
j_0 &\leftarrow u_{j_0}.\text{idx}[x[\phi_{j_0}] > \theta_{j_0}] \\
j_1 &\leftarrow u_{j_1}.\text{idx}[x[\phi_{j_1}] > \theta_{j_1}] \\
j_2 &\leftarrow u_{j_2}.\text{idx}[x[\phi_{j_2}] > \theta_{j_2}] \\
j_3 &\leftarrow u_{j_3}.\text{idx}[x[\phi_{j_3}] > \theta_{j_3}] \\
&\vdots
\end{aligned}
\tag{3.4}
$$

The idea is to traverse a layer in the tree with many instances at once. So that while we are waiting for the memory answering to $x[\phi_{j_0}]$ operations needed to access $x[\phi_{j_1}]$ can be started, and so on. Hopefully, by the time the final memory access has been dispatched, the contents of the first memory access are available, and we can continue without processor stalls [23]. In this manner, we expect vectorization to mask memory latencies, thus reducing the impact of dependencies between instructions [23].

3.1.4 QuickScorer

An entirely different approach is to restate the problem of tree traversal into a series of bitwise operations between bitvectors. This new strategy will produce a bitvector encoding the exit leaf for a given document [26]. A first nice property of this idea is that, thanks to the nature of bitwise operations, it's insensible to the order in which the nodes are processed.

To better understand how can we implement this idea let us start from a more general approach. Given a feature vector x, a tree $T_h(I_h, L_h)$ with nodes enumerated in a breath-wise manner and a set of candidate exit leaves $\mathcal{C}_h \subseteq L_h$, initially containing all leaves. The goal is to refine $\mathcal{C}_h$ until it contains only one element which will be the exit leaf of x [26]. The algorithm works by evaluating the internal nodes in an arbitrary order. For any node $u \in I_h$ if it is a false node then all leaves in its left subtree cannot be the exit leaf and so are removed from $\mathcal{C}_h$, likewise if u is a true node then the leaves in the right subtree are removed. Once all the nodes are considered, $\mathcal{C}_h$ will surely contain only $e(x)$. This initial formulation does not induce a clear advantage in traversing the trees since we are performing all nodes' tests. However, suppose now to have an oracle called **FindFalse** that given a tree will return all its false nodes without evaluating any of the true nodes. With that we can remove from $\mathcal{C}_h$ all the leaves in the left subtrees of the nodes returned by the oracle, unfortunately now $\mathcal{C}_h$ will contain more than one node. To understand this, consider the edge case in which T_h has no false nodes, then we will not remove any node from $\mathcal{C}_h$ and thus it will still be equal to L_h. This issue is resolved by the interesting fact that the exit leaf will always be the leftmost leaf still in $\mathcal{C}_h$, being the one with the lowest identifier [26] (see the proof of theorem 1). To perform operations in $\mathcal{C}_h$ efficiently represent it using a bitvector $\mathbf{v}_h$, where each bit corresponds to a leaf in L_h. Each internal node u is also associated with a bitvector acting as a mask signaling the leaves to be removed from $\mathcal{C}_h$ whenever u is a false node. Doing so, performing a bitwise AND between v_h and the mask of a false node u is equivalent to removing all leaves in the left subtree of u from $\mathcal{C}_h$ [26]. This approach is summarized in algorithm 6.

Algorithm 6: General scoring algorithm using bitvectors [26]

def *score(x: featureVector, T_h: Tree)*:
  $\mathbf{v}_h \leftarrow 11...11$;
  $U \leftarrow$ **FindFalse**(x, T_h);
  foreach $u \in U$ **do**
    $\mathbf{v}_h \leftarrow \mathbf{v}_h \wedge u$.bitvector;
  end
  $j \leftarrow$ index of the leftmost 1 in $\mathbf{v}_h$;
  return $l_j.\gamma$;

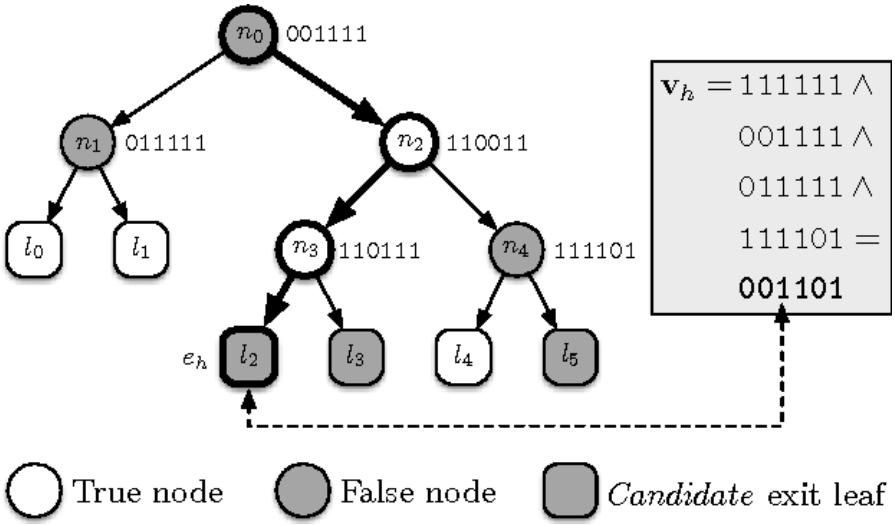


Figure 3.2: Tree traversal using bitvectors. (Image taken from the QuickScorer paper [26])

Theorem 1. *Algorithm 6 is correct.*

Proof. The proof of correctness in divided in two parts, first we prove that at the end of the algorithm the bit corresponding to $e_h(x)$ in $\mathbf{v}_h$ is set to 1. Then we prove that the leftmost bit equal to 1 in $\mathbf{v}_h$ at the end of the algorithm is the one associated to $e_h(x)$. For the first part, observe that only the bitvectors of the nodes in the path from the root to $e_h(x)$ may change $e_h(x)$ to 0. However, since $e_h(x)$ is the exit leaf, it belongs to the left subtree of any true node and to the right subtree of any false node in the path. Thus, since the only bits that are ever set to 0 reside on the left subtree of false nodes, then surely $e_h(x)$ will remain unchanged and so be 1 at the end of the algorithm. For the second part of the proof, let $l_{\leftarrow}$ be the leftmost bit equal to 1 in $\mathbf{v}_h$. Assume, by contradiction, that $e_h(x) \neq l_{\leftarrow}$. Let u be their lowest common ancestor. Since $l_{\leftarrow}$ is smaller than $e_h(x)$, then the leaf $l_{\leftarrow}$ belongs to the left subtree of u and $e_h(x)$ to its right

subtree. This leads to a contradiction. Observe that u has to be a true node, otherwise $l_{\leftarrow}$ would have been set to 0 by its bitvector, at the same time u should be a false node since the exit leaf $e_h(x)$ is in its right subtree. Thus, the only option is that $l_{\leftarrow} = e_h(x)$ [26]. □

Unfortunately, this simple version relies on an unrealistic oracle. As such, the next step will be to efficiently extrapolate the false nodes of T_h. The key idea of QuickScorer is to traverse the ensemble not tree by tree but rather feature by feature [26]. The algorithm works by looping over the features, discovering for each $v_k \in V$ all false nodes in any tree of the ensemble involving it. This is an interesting choice for two main reasons: for once we are able to consider all and only the false nodes, without considering the true ones; on the other we can proceed in a cache-friendly way reducing the cache misses, the amount of comparisons and, consequently, branch mispredictions[26]. QuickScorer maintains all the bitvectors $\mathbf{v}_h$'s, one for each tree, updating the corresponding one as soon as a node is recognized as false. Once all features have been considered, each of the $\mathbf{v}_h$ will contain the information needed to retrieve the exit node of that tree. After that, we can safely compute the score of x by summing up the γ's of the exit nodes [26].

As the last step let us see how we can efficiently retrieve the false nodes using a given feature v_ϕ. We can describe any node using v_ϕ by a triplet composed by: the threshold of the test; the id of the tree in which the node resides; the node bitvector [26]. We sort these triplets by ascending order of their thresholds. This sorting is a essential step to achieve an efficient implementation [26]. Observing that all tests are in the form $x[\phi] \leq \theta_s^h$ then the feature value $x[\phi]$ splits the sorted list of all thresholds involving v_ϕ into two sublists. The first contains all thresholds for which $x[\phi] \leq \theta_s^h$ evaluates to false, symmetrically, the other contains the ones evaluated to true. Thus, scanning the list we would encounter a long sequence of the thresholds associated to false nodes, when we encounter a value for which $x[\phi] \leq \theta_s^h$ evaluates to true, we can stop since we already encountered all, and only, the false nodes regarding v_ϕ [26]. In short, the only comparisons we would ever do are all evaluated to false, which will heavily reduce the number of branch mispredictions [26].

Algorithm 7: QuickScorer [26]

def *QUICKSCORER(x: featureVector, M: Ensemble)*:
Data:
x: input feature vector
M: ensemble of binary decision trees with:
- $w_0, \dots, w_{|M|-1}$ weights, one per tree
- **thresholds**: sorted list of tresholds, one per feature
- **treeids**: tree's ids, one per threshold
- **bitvectors**: node bitvectors, one per threshold
- **offsets**: offsets of the triplets
- **v**: result bitvectors, one per tree
- **leaves**: output values, one per leaf
foreach $h \in 0, 1 \dots, |M| - 1$ **do**
  $\mathbf{v}_h \leftarrow 11...11$;
end
foreach $k \in 0, 1 \dots, |V| - 1$ **do**
  $i \leftarrow$ offsets$[k]$;
  $end \leftarrow$ offsets$[k+1]$;
  while $x[k] >$ *thresholds[i]* **do**
    $h \leftarrow$ treeids$[i]$;
    $v[h] \leftarrow v[h] \wedge$ bitvectors$[i]$;
    $i \leftarrow i + 1$;
    if $i \geq end$ **then**
      break;
    end
  end
end
$s \leftarrow 0$;
foreach $h \in 0, 1, \dots, |M| - 1$ **do**
  $j \leftarrow$ index of the leftmost 1 in $v[h]$;
  $l \leftarrow h \cdot |L_h| + j$;
  $s \leftarrow s + w_h \cdot$ leaves$[l]$;
end

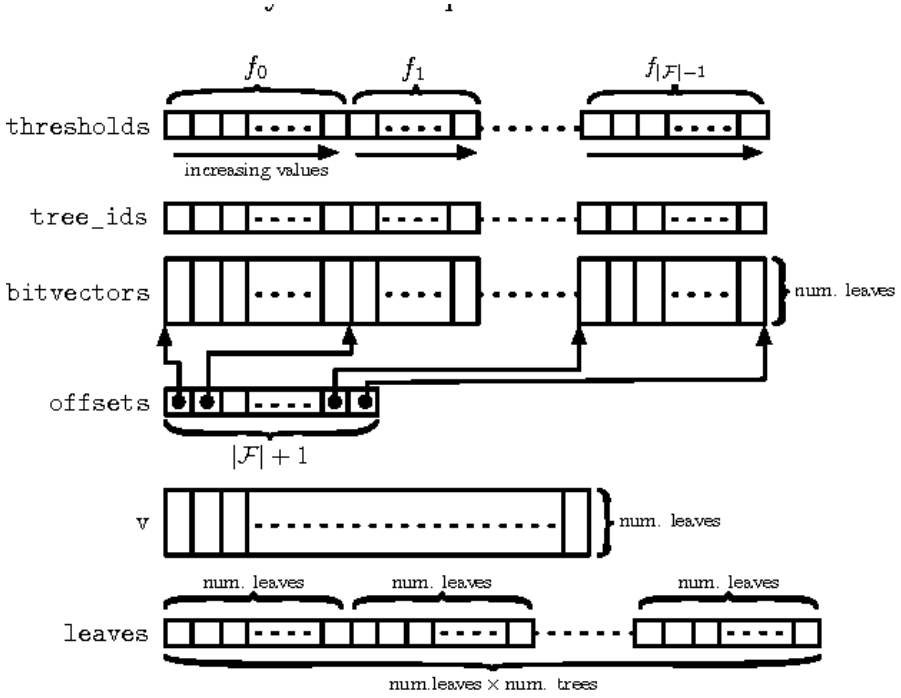


Figure 3.3: All the data structures used by quickscorer. Here the set of features is represented as $\mathcal{F} = \{f_0, f_1, \dots, f_\phi, \dots, f_{|\mathcal{F}|-1}\}$ instead of $V = \{v_0, v_1, \dots, v_\phi, \dots, v_{|V|-1}\}$(Image taken from the QuickScorer paper [26])

Block wise QuickScorer

We can slightly modify the algorithm to increase data locality by scoring the documents in sequential portions of the ensemble, essentially scoring each document in sub-ensembles. This approach consists in partitioning the trees in blocks. Doing so allows us to process each block independently, allowing more of the data structures likely to be used to reside in the cache [26]. The downside of this approach is that we have to keep track of the partial scores for each document. We can further build upon this idea by considering also blocks of documents to be scored on a given tree block, for this we would need to replicate the result bitvectors v. The main motivation behind this idea is that QuickScorer has been observed to work better with small ensembles [26]. However, using a block-wise strategy increases the space required and may result in a worse cache hit ratio. Combining the two observations by a good trade-off between data duplication and block size(s) will likely result in an overall increase in performance [26].

3.2 Document Pruning Strategies

Thanks to the fact that in ranking we do not necessarily need accurately computed document scores as long as the resulting ranking is correct, and that users are usually interested only in a small subset of the results (i.e. top k documents) we can employ another, different but complementary, approach by performing some optimizations at a higher level, altering the scoring mechanism itself to speedup the overall process.

If we assume that the scorers have all approximately the same constant scoring cost $c > 0$ then each document has the same cost $C(x) \approx cM$. Thus, the cost of ranking a query composed of n documents is $C(q) = cMn$. These observations lead to estimate the overall ranking cost of a set $\mathcal{D}$ of N documents divided in a set $\mathcal{Q}$ of Q queries as

$$C(\mathcal{Q}) = \sum_{i=1}^{Q} C(q_i) = \sum_{i=1}^{Q} \sum_{j=1}^{|q_i|} C(x_{ij}) \approx cMN \tag{3.5}$$

where $|q_i|$ is the number of documents associated to query q_i.
This clearly shows that the cost can become quite large when both M and N grow in size (e.g., $M > 10^5$ and $N > 1000 \implies C \geq c10^8$).

A possible strategy to achieve better performance is to interrupt the scoring of likely irrelevant samples. In general, we can early exit a document x at any position $h < M$, obtaining a partial score $\hat{s}(x) = \sum_{i=1}^{h} f_i(x)$ using only the first h scorers. The decision to early exit a document depends on the likelihood that it will be one of the top k documents once all scorers have been executed, and accordingly either continue the scoring of x_i or exit it after the execution of scorer f_h [18]. The decision will be performed using a so-called early exit function. An early exit function $\mathcal{E}_h(x, \mathrm{T}, H)$ at position h is a function indicating if x should be early exited at h which takes as input the scored instance, x, a vector of parameters $\mathrm{T} = \left[\tau_1, ...\right]$ and some history information H about the previous decisions. Thus, we can define a general early exit function as

$$\mathcal{E}_h(x, \mathrm{T}, H) = \begin{cases} 1, & \text{if } x \textbf{ is not} \text{ stopped} \\ 0, & \text{if } x \textbf{ is} \text{ stopped} \end{cases} \tag{3.6}$$

Although it is possible to perform early exit at every possible position $1 \leq h \leq M$, generally we mark some scorers as "sentinels" after which an early exit function is executed, resulting in a sequence of early exit positions $e = \left[e_1, \cdots, e_P\right]$ with $P \ll M$. Thus we can define an early exit strategy $S = (e, \{\mathcal{E}_h\}_{h \in e})$ as a sequence of sentinels, each with an associated early exit function. The cost of scoring using early exit is

$$C(\mathcal{D}, S) = C(\mathcal{Q}, S) = \sum_{i=1}^{Q} C(q_i, S) = \sum_{i=1}^{Q} \sum_{j=1}^{|q_i|} p(x_{ij})$$

where $p(x_{ij})$ is the position at which the scoring of x_{ij} was stopped by S [18]. This cost is computed without considering the cost of performing the early exit itself.

Clearly, considering only a partial score may degrade the result quality as now some documents will not be in the rank they were supposed to be with a complete score. Let $\hat{\pi}^S$ be the ranking obtained using early exit strategy S. Moreover, let $\chi_k\left(\pi_q, \hat{\pi}_q^S\right)$ be a function that measures the relevance loss in the top k (i.e., NDCG@k) of $\hat{\pi}_q^S$ relative to π_q. The early exit problem can now be stated as finding the early exit strategy S which minimizes the expected relevance loss $E\left[\chi_q\left(\hat{\pi}_q^S, \pi_q\right)\right]$ and minimizes the scoring cost $C(\mathcal{D}, S)$. The remaining part of this section is dedicated to the discussion of three possible early exit functions, presented in a paper in 2010 [18], which

served as the main building blocks for this work. In their original formulation presented by B. Barla Cambazoglu et al. in [18], the early exit functions required an offline computed array of fixed thresholds (either score or ranked based).

3.2.1 Score Based Pruning

A first and intuitive approach is to stop documents based on their partial scores, such method is called Early exit with Score Thresholds (EST). EST works by filtering documents with low partial scores. That is, at each position p we compare the current partial score of x with the lowest allowed score, being a fixed threshold st[p]. If the score is less than the threshold, we stop scoring x. The main

Algorithm 8: EST

```
Data: st[M]: array of M score thresholds
scores = [0, 0, ··· , 0] ;
foreach x ∈ {0, ... , N} do
    foreach p ∈ {1, ··· , M} do
        scores[x] = scores[x] + F_p(x) ;
        if scores[x] < st[p] then
            X ← X − {x}    //early exit;
            break ;
        end
    end
end
```

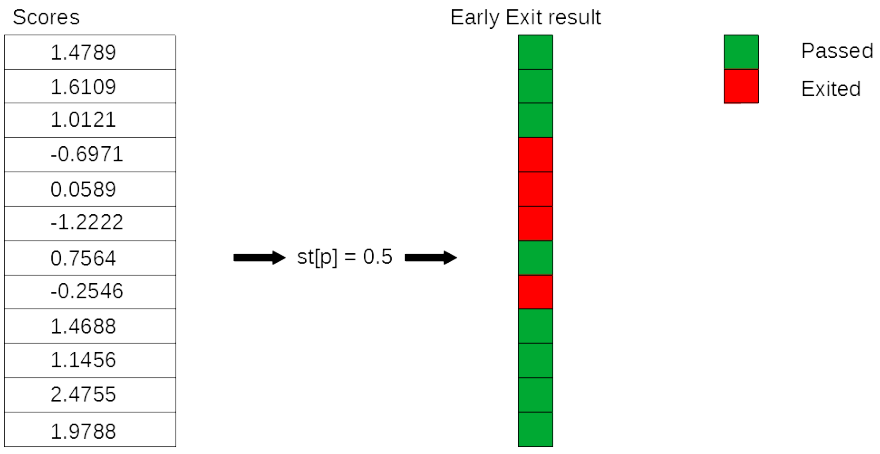


Figure 3.4: Graphical representation of a possible EST execution

advantage of EST is its simplicity, making it incredibly trivial to implement in any scoring architecture. Unfortunately, static score thresholds lead to poor results as distribution of scores varies heavily between queries.

3.2.2 Rank Based Pruning

Before introducing an early exit function based on rank, we need to define the partial ranking of a query. A partial ranking $\tilde{\pi}_{ih}$ of a query q_i at depth h is the ranking induced by the partial scores accumulated after the first h scorers [18].

With that definition, we can introduce the Early exit with Rank Threshold (ERT). ERT works by comparing the partial ranking of each document at position p with a threshold rt[p]. The documents with a better rank than the threshold may continue to be scored. ERT can be modified to retrieve the document with partial rank rt[p] and use its score as filter. ERT, as we will see, works very well

Algorithm 9: ERT

```
Data: rt[M]: array of M rank thresholds
scores = [0, 0, ··· , 0] ;
foreach x ∈ {0, ... , N} do
    foreach p ∈ {1, ··· , M} do
        if x ∈ X then
            scores[x] = scores[x] + F_p(x) ;
        end
        sort(scores);
        x' ← scores[rt[p]] ;
        if scores[x] < scores[x'] then
            X ← X − {x}    //early exit;
            break;
        end
    end
end
```

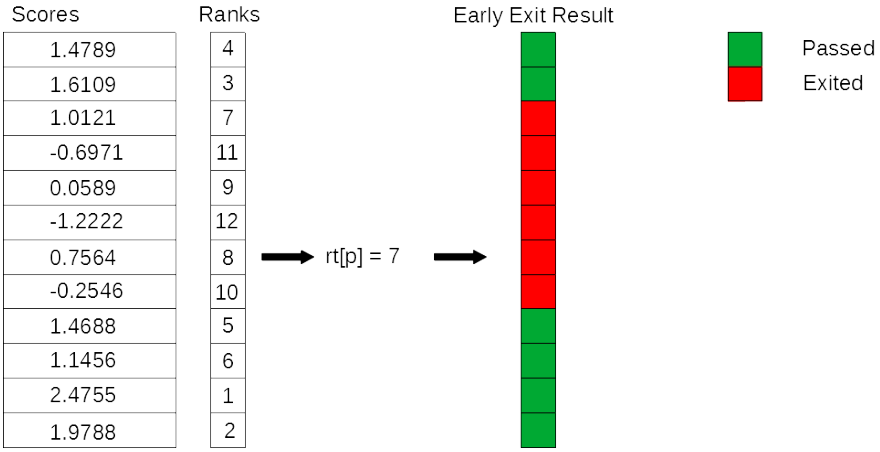


Figure 3.5: Graphical representation of a possible ERT execution

in practice since at the cost of a more demanding exiting process, it is more resilient to the varying score distributions [18].

3.2.3 Proximity Based Pruning

The last early exit function we used combines the first two. Early exit with Proximity Thresholds (EPT) works by preserving the first $\hat{k} \geq k$ documents in the partial ranking and those whose score is sufficiently close to the score of the $\hat{k}$-th document, where the closeness is parameterized by a threshold $pt[p]$. EPT is essentially an ERT with dynamic thresholds [18].

Algorithm 10: EPT

```
Data: pt[M]: array of M proximity thresholds
scores = [0, 0, ··· , 0] ;
foreach x ∈ {0, ... , N} do
    foreach p ∈ {1, ··· , M} do
        if x ∈ X then
            scores[x] = scores[x] + F_p(x) ;
        end
        sort(scores);
        x' ← scores[k] ;
        if scores[x] < scores[x'] - pt[p] then
            X ← X − {x}    //early exit;
            break ;
        end
    end
end
```

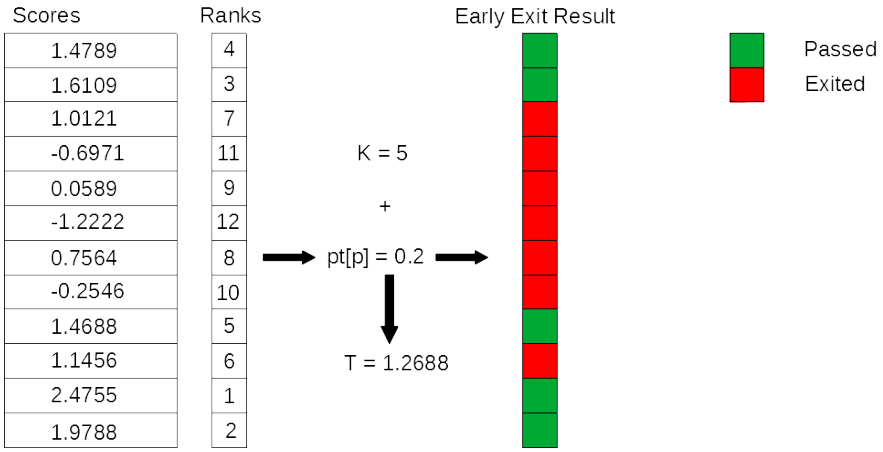


Figure 3.6: Graphical representation of a possible EPT execution

Chapter 4

Analysis Description

As seen in section 3.2 the thresholds of the early exit functions need to be statically decided. The novel idea we propose is to compute them just before applying the pruning so that, by deciding beforehand only one or two hyper-parameters, the information of the partial scores and ranks can also be exploited. We tested this idea on three out of four options provided in [18] being those described earlier.

In our analysis we posed the following research question: what is the impact of placing up to two early exit sentinels and the corresponding functions in our model? Recalling that placing an early exit sentinel in an additive ensemble means to interrupt the scoring of any document after a predetermined scorer and choosing, using a early exit function, to either continue or stop processing that particular document, as depicted in figure 4.1.

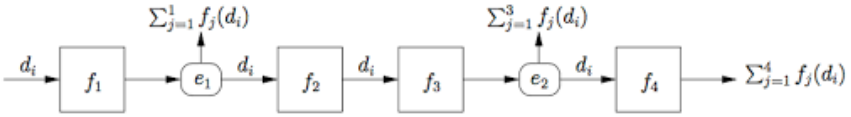


Figure 4.1: Additive ensemble with early exit sentinels.
(Image taken from [18])

To give an answer to our question we performed a three step analysis:

1. Firstly we searched for a single sentinel combination, we tried to place it at trees 25, 50, 75 and 100 trying all the available functions with some parameters. After doing so we picked three candidates that seemed promising by looking at both their induced speedup and drop in NDCG;

2. After that, for each of those candidates we tried every possible

choice of a second sentinel in the trees 150, 200, 250 and 300. With those results we chose again three pairs forming three possible early exit strategies, each composed by two sentinels and two early exit functions.

3. Finally, we tested these strategies applying them in the test set both in RankEval (Python) and QuickScorer (C++).

We also tested a more deeply placed single sentinel strategy at trees 150, 200, 250 and 300. This was done to understand if a single well-placed sentinel could work better than a multisentinel approach.

4.1 Thresholds' definition

Just to recall what said in the beginning of chapter 2, our data is composed of N documents divided in Q queries, for each query-document pair (q_i, d_j) a feature vector $x_{ij} \in \mathcal{X}$ is crafted. Feature vectors are then scored by the ensemble composed by M regression trees computing $F(x_{ij})$. Suppose now to be at an early exit sentinel e_h after accumulating h scores for all samples of query q_i, this means that for each x_{ij} we have a partial score $\hat{F}_h(x_{ij}) = \sum_{l=1}^{h} f_l(x_{ij})$. Giving for each each query q_i a partial score vector $\hat{s}_i = \left[\hat{F}_h(x_{i1}), \ldots, \hat{F}_h(x_{i|q_i|})\right]$. For convenience, define also $r(s, z)$ the function that returns the zth element of the sorted version of a score vector s, essentially giving the score of the document with rank z. Using these information, we are able to dynamically define the pruning thresholds for each of the three possible pruning methods of the early exit function associated to e_h. Since we use information coming from all the scores of a given query, each pruning threshold p_i is applied to all and only the documents of query q_i:

1. For EST we decided to use the mean and standard deviation of the partial scores:

$$p_i = \alpha\mu(\hat{s}_i) + \beta\sigma(\hat{s}_i) \qquad \alpha \in \mathbb{R},\ \beta \in \mathbb{R} \tag{4.1}$$

2. For ERT we based the thresholds on the size of the query:

$$p_i = k + \delta|q_i| \qquad 0 < \delta \leq 1 \tag{4.2}$$

3. For EPT we implemented the notion of score proximity to the kth partial ranked document using the variability of the scores through the standard deviation:

$$p_i = \hat{F}_d\left(r(\hat{s}_i, k)\right) + \beta\sigma(\hat{s}_i) \qquad \beta \in \mathbb{R} \tag{4.3}$$

where $\mu(s)$ is the average of the scores in s and $\sigma(s)$ is their standard deviation. The hyper-parameters α, β, δ are three multipliers chosen beforehand and so they need to be tuned.

4.2 QuickScorer adaptations

In its original formulation, all that quickscorer does is score a sequence of documents, or blocks of documents, without any concept of query grouping [26]. However, since we are interested in assessing how our new pruning strategies affect the NDCG, which is a query-oriented measure, we need to make some little variations to quickscorer to make it more query-aware. The solution adopted was to modify the concept of document blocking to make it work so that the resulting portions of documents are aligned with the queries. For example, instead of making quickscorer run on a batch of 50 documents at a time, we made it run on a block of 10 queries. In practice, we implemented this idea by specifying the number of blocks in which divide them, for example, the test set of the MSLRF1 dataset contains 6306 queries, so if we specified 10 as the number of query blocks we would end up with nine blocks of 630 queries and one of 636.

Later, we expanded this idea by allowing the on-demand read of blocks of documents directly from file, the idea is that we do not need the actual instances to build the data structures of quickscorer, making it possible to save a good amount of memory by reading only the documents belonging to the queries in the current block being scored. We called this concept online read, and we will see if and how making multiple accesses to the file affect the performance.

Lastly, we introduced the possibility to perform early exit in quickscorer by allowing any tree block to behave as a sentinel, that is, at the end of some predetermined blocks, we can perform document pruning, as such we do not have a complete freedom in positioning our sentinels but we have to place them at multiples of the tree block size, the resulting structure will be similar to the one of figure 4.2.

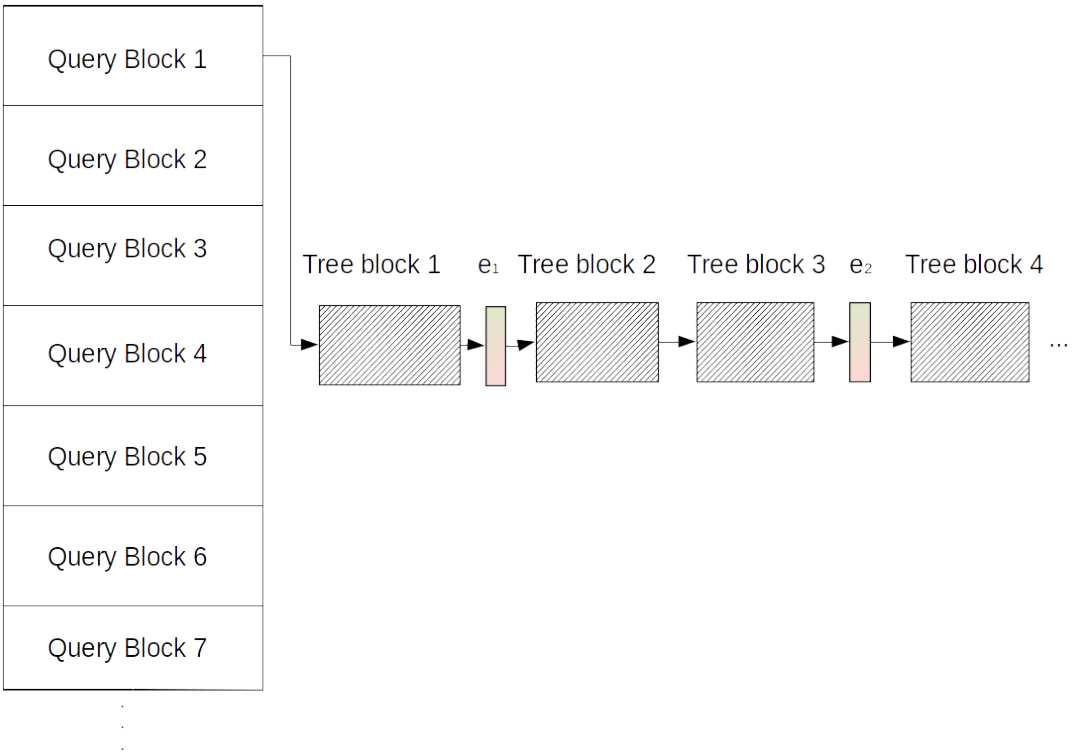


Figure 4.2: QuickScorer with Early Exit

4.3 Experimental Setting

For our experiments we used the first fold of the publicly available MSLR dataset [24], named MSLRF1 and described in table 4.1. The ranking ensemble used is a LambdaMART model optimizing the NDCG@10, it is composed by 1047 trees with 64 leaves each. The ensemble was trained on the training split of the dataset using LightGBM [28] as training framework. The searches described earlier and analyzed in chapter 5 are performed in the validation split, the selected candidates are then re-evaluated in the test split, see tables 5.7 and 5.8. The experiments are performed on a single core in a machine equipped with: an Intel Xeon E5-2650v3 clocked at 2.30 GHz, with 126 GB of RAM, running Ubuntu Linux with 4.4.0-189-generic as kernel. The CPU has 3 levels of cache. The L1 cache is divided between data and instruction memory each of 32KB, while L2 and L3 are shared and their sizes are, respectively, of 256 KB and 25600 KB.

Property	Split		
	Train	Validation	Test
# of features	136	136	136
# of queries	18919	6306	6306
# of documents	2270296	747218	753611
Average # of documents per query	120.0	118.5	119.5

Table 4.1: MSLRF1 dataset composition

Please note that to avoid modifying the internal structures of the queries (sizes and offsets), to perform the early exit we simply marked as deleted pruned documents and set their score to a very low value (-1000) so that they will be ranked always at the bottom of the list, effectively achieving a behaviour similar to the pseudo code presenter earlier.

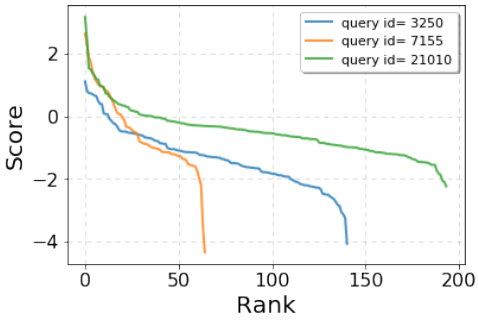
Chapter 5

Results

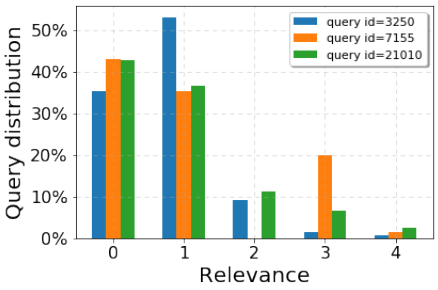
In this chapter we are going to explore our results starting from a general overview of the data and the behaviour of our model to then explore the actual impact of early exiting documents with dynamic thresholds. For easiness in both presentation and analysis, we decided to combine the parameters discussed in section 4.1 into one, called τ, which will behave accordingly to the exit function considered (for EST we assumed $\alpha = 1$).

5.1 General observations

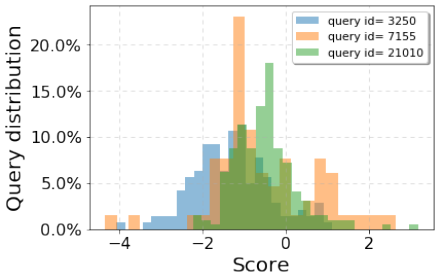
Let us start our discussion by observing the true relevance distribution of randomly selected queries. As shown in Figure 5.1, in general, relevances are distributed highly asymmetrically, since most of the documents associated with a query are irrelevant to it (labels 0 and 1) while the truly relevant ones are scarce (labels 3 and 4). This fact is partially reflected also in the distribution of the scores given by the model, since while it is true that only a few of them have a high score, there is not a clear gap between the scores of relevant and irrelevant documents but rather it is a smooth transition, which in turn seems to suggest that it will be hard to find a good static threshold based entirely on the scores. Another consideration is that query sizes can be quite different, ranging from a small collection of a dozen samples to queries containing hundreds of documents, again giving a possible hint on the difficulty of selecting static thresholds based on rank.



(a) Distribution of the score by rank



(b) Distribution of relevance labels



(c) Distribution of document scores

Figure 5.1: Example of the asymmetry of score and label distributions in three randomly selected queries

One last interesting fact concerning the scores is how the average NDCG@10 varies as the documents traverse the ensemble. In Figure 5.2 we can observe that after the first approximately 250 trees, the growth of NDCG slows down significantly, improving only marginally for the remaining part of the scoring process.

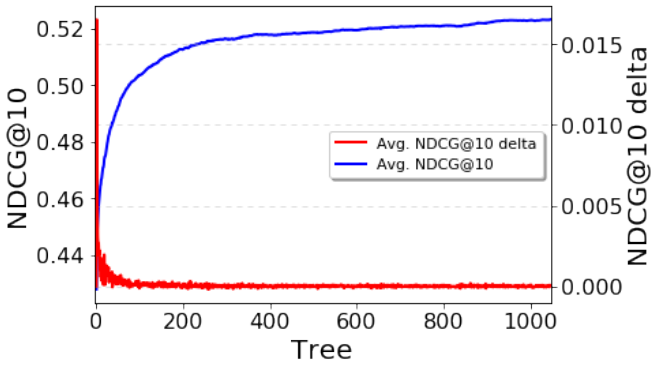
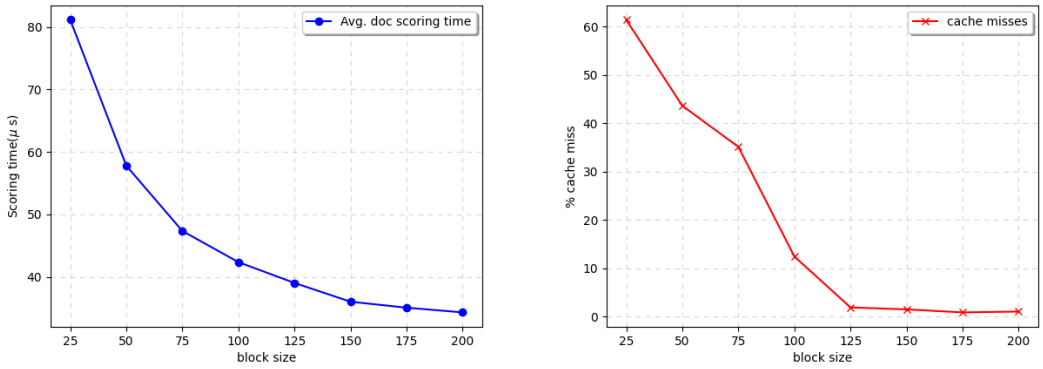


Figure 5.2: Variation of the NDCG by tree

As the second part of this introductory section, let us explore how our two main parameters for quickscorer, tree and query block size, affect the scoring time.

Figure 5.3 clearly shows how dividing the ensemble in very small blocks affects enormously the scoring time and how it is strongly correlated with the high cache miss ratio registered when using small

sizes for tree blocks. On the other hand, the number of query blocks (figure 5.4) seems to have only a marginal effect on the scoring time (roughly 2μs per document), although using an online read approach slows the scoring process by roughly 20%, with a difference of approximately 10μs per document between online and offline reading.



(a) Scoring time and tree block size

(b) Cache misses and tree block size

Figure 5.3: Scoring time and cache misses relative to the tree block size

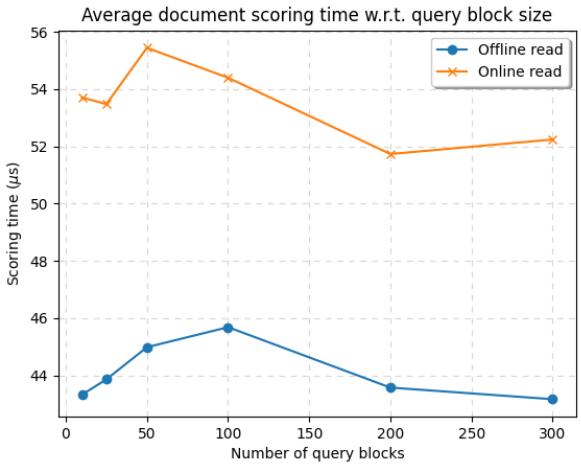
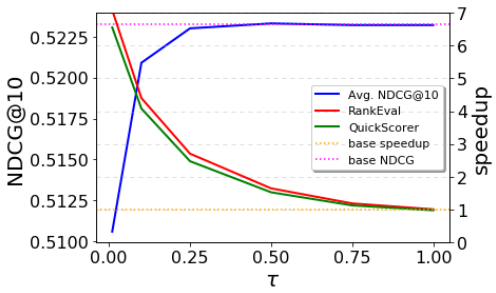
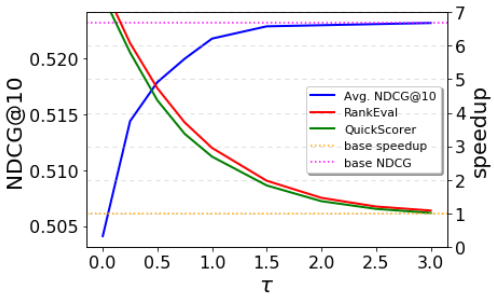


Figure 5.4: Scoring time with online and offline read (tree b. size=100)

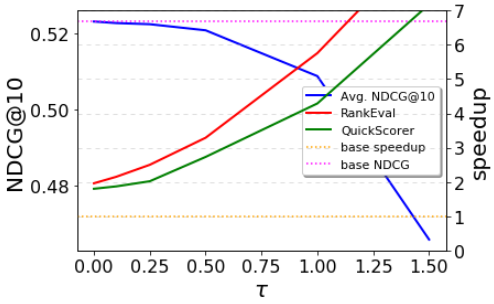
5.2 Single sentinel search



(a) Speedup and NDCG in ERT



(b) Speedup and NDCG in EPT



(c) Speedup and NDCG in EST

Figure 5.5: Relation between speedup and drop in NDCG on the three strategies with $h = 50$

		Sentinel position (h)							
		25		50		75		100	
$\mathcal{E}$	τ	Δ NDCG (-%)	speedup	Δ NDCG (-%)	speedup	Δ NDCG (-%)	speedup	Δ NDCG (-%)	speedup
EPT	0.25	3.09	7.01	1.72	6.12	1.32	5.42	0.98	4.86
	0.50	1.75	5.25	1.02	4.75	0.72	4.33	0.48	3.97
	0.75	1.06	4.01	0.60	3.73	0.36	3.46	0.25	3.24
	1.00	0.73	3.14	0.28	2.96	0.18	2.79	0.10	2.65
ERT	0.10	1.05	4.79	0.46	4.41	0.31	4.07	0.18	3.77
	0.25	0.23	2.81	**0.03**	**2.70**	0.03	2.59	**0.04**	**2.49**
	0.50	0.00	1.66	**0.00**	**1.64**	0.00	1.62	0.00	1.59
	0.75	0.00	1.19	0.00	1.18	0.00	1.18	0.00	1.17
EST	1.00	3.87	6.73	2.84	5.79	2.31	5.12	2.01	4.06
	0.50	0.23	2.62	0.23	2.40	0.25	2.22	0.24	2.06
	0.00	0.18	2.03	0.04	1.97	0.04	1.90	0.09	1.86
	-0.50	0.00	1.39	0.00	1.38	0.01	1.37	0.01	1.35
	-1.00	0.01	1.15	0.01	1.15	0.02	1.15	0.01	1.15

Table 5.1: Result of simulation with one pruning sentinel with local strategy

Table 5.1 shows the results obtained by placing a single sentinel using all three strategies. The results, depicted also in figure 5.5, clearly show that there is an obvious trade-off between speedup and NDCG preservation, indeed, while facing drops in NDCG up to almost 4% we are able to obtain speedups reaching 7x. On the other hand, we can get speedups around 2.7x without any significant loss in quality (drop $\leq 0.05\%$).

Given that we have observed a high variation in average NDCG in the first third of the model (figure 5.2), for the first sentinels, we

chose to be a bit conservative by picking combinations of functions and parameters giving a low loss in quality, but still a substantial speedup. Considering also the behaviour of QuickScorer when dealing with small tree blocks, we also limited our choices to sentinels positioned in multiples of at least 50. With these considerations, we have highlighted the three candidates that we are going to use as starting points for placing the second sentinel.

As anticipated at the beginning of chapter 4 we also tested the possibility of placing a single sentinel well beyond the 100th tree, obtaining the results summarized in table 5.2. The main insight that we can extrapolate from those values is that the idea of placing a single sentinel at, for example, tree 200 will still provide an interesting speedup (2.92x) with a very small drop in NDCG (0.04%), coupled with a larger size for the blocks of trees it may result as a valid alternative to a multi sentinel strategy.

		Sentinel position							
		150		200		250		300	
$\mathcal{E}$	τ	Δ NDCG (-%)	speedup	Δ NDCG (-%)	speedup	Δ NDCG (-%)	speedup	Δ NDCG (-%)	speedup
EPT	0.25	0.55	4.03	0.34	3.45	0.22	3.02	0.16	2.68
	0.50	0.20	3.43	0.13	3.02	0.13	2.70	0.09	2.44
	0.75	0.12	2.89	0.08	2.61	0.07	2.38	0.06	2.19
	1.00	0.06	2.43	0.05	2.25	0.04	2.09	0.04	1.96
ERT	0.10	0.08	3.29	**0.04**	**2.92**	0.03	2.62	0.03	2.38
	0.25	0.03	2.31	0.03	2.15	0.03	2.02	0.03	1.90
	0.50	0.00	1.54	0.00	1.50	0.01	1.46	0.00	1.42
	0.75	0.00	1.16	0.00	1.15	0.00	1.14	0.00	1.13
EST	1.00	1.42	3.85	1.31	3.32	1.32	2.92	1.21	2.61
	0.00	0.06	1.77	0.06	1.70	0.03	1.63	0.03	1.57
	-0.50	0.01	1.33	0.01	1.31	0.01	1.28	0.00	1.26
	-1.00	0.01	1.14	0.01	1.13	0.01	1.12	0.00	1.11

Table 5.2: Result of simulation with one deep pruning sentinel with local strategy

Before continuing with the analysis towards two sentinels, let us examine how our strategies perform compared to the original static ones, tables 5.5, 5.3 and 5.4 reflect just that. From those results we can observe that the only function with a clear and objective improvement is EST, since considering an equal, for example, speedup, it has always a lower impact on the NDCG than the static counterpart. EST is the one more affected by this change in perspective, most likely because, as observed in Figure 5.1, scores can be significantly different from a query to another and thus finding an appropriate static threshold is inherently non-trivial. While the other two have comparable performances since there may exist some good guesses for a rank-based threshold such as some multiple of k, being the number of return results from the search engine.

		Sentinel position							
		25		50		75		100	
$\mathcal{E}$	τ	Δ NDCG (-%)	speedup	Δ NDCG (-%)	speedup	Δ NDCG (-%)	speedup	Δ NDCG (-%)	speedup
EST	1.00	3.87	6.73	2.84	5.79	2.31	5.12	2.01	4.06
	0.00	0.18	2.03	0.04	1.97	0.04	1.90	0.09	1.86
	-0.50	0.00	1.39	0.00	1.38	0.01	1.37	0.01	1.35
	-1.00	0.01	1.15	0.01	1.15	0.02	1.15	0.01	1.15
GEST	0.25	15.75	9.4	12.24	6.12	10.83	4.89	10.35	4.25
	0.00	6.24	3.92	6.29	3.56	6.34	3.26	6.37	3.06
	-0.50	0.18	1.26	0.98	1.49	1.62	1.61	2.08	1.66
	-1.00	0.00	1.03	0.09	1.12	0.21	1.20	0.4	1.25

Table 5.3: EST Comparison between static (global) and dynamic (local) thresholds

		Sentinel position							
		25		50		75		100	
$\mathcal{E}$	τ	Δ NDCG (-%)	speedup	Δ NDCG (-%)	speedup	Δ NDCG (-%)	speedup	Δ NDCG (-%)	speedup
EPT	0.25	3.09	7.01	1.72	6.12	1.32	5.42	0.98	4.86
	0.50	1.75	5.25	1.02	4.75	0.72	4.33	0.48	3.97
	0.75	1.06	4.01	0.60	3.73	0.36	3.46	0.25	3.24
	1.00	0.73	3.14	0.28	2.96	0.18	2.79	0.10	2.65
GEPT	0.30	0.67	2.76	0.63	3.59	0.52	3.76	0.36	3.70
	0.50	0.18	1.73	0.22	2.4	0.18	2.68	0.15	2.78
	0.70	0.04	1.28	0.08	1.75	0.07	2.01	0.03	2.14
	0.90	0.02	1.11	0.04	1.39	0.04	1.60	0.02	1.72

Table 5.4: EPT Comparison between static (global) and dynamic (local) thresholds

		Sentinel position							
		25		50		75		100	
$\mathcal{E}$	τ	Δ NDCG (-%)	speedup	Δ NDCG (-%)	speedup	Δ NDCG (-%)	speedup	Δ NDCG (-%)	speedup
ERT	0.10	1.05	4.79	0.46	4.41	0.31	4.07	0.18	3.77
	0.25	0.23	2.81	0.03	2.70	0.03	2.59	0.04	2.49
	0.50	0.00	1.66	0.00	1.64	0.00	1.62	0.00	1.59
	0.75	0.00	1.19	0.00	1.18	0.00	1.18	0.00	1.17
GERT	20	1.19	5.15	0.67	4.7	0.54	4.31	0.38	3.97
	30	0.61	3.67	0.25	3.46	0.16	3.27	0.13	2.16
	50	0.14	2.37	0.09	2.3	0.05	2.23	0.03	2.16
	75	0.09	1.69	0.02	1.67	0.01	1.64	0.00	1.62

Table 5.5: ERT Comparison between static (global) and dynamic (local) thresholds

5.3 Double sentinel search

Observing table 5.6 and figure 5.6 we can ascertain that, within some degree, starting from any of the three first choices we are able to reach a wide variety of results, from very safe strategies that without losses in NDCG provide a speedup ranging from 1.6x to roughly 2x up to some that facing a drop of around 1% will result in speedup of over 5x, with a maximum of 6.32x. Additionally, we can state that all strategies have a smooth transition from high gain - high risk options to more conservative ones. Therefore, it is not unrealistic to think about a procedure that given a target maximum loss of quality will return a set of possible early exit strategies to employ.

As the last point in this analysis, we tested the highlighted strategies on table 5.6 in the test set both in our simulation environment and in QuickScorer so to have a more plausible measure of speedup. We also picked a fourth one (marked in *italics*) since its parameters will allow for a tree block size of 100 instead of 50 like the others when run in QuickScorer.

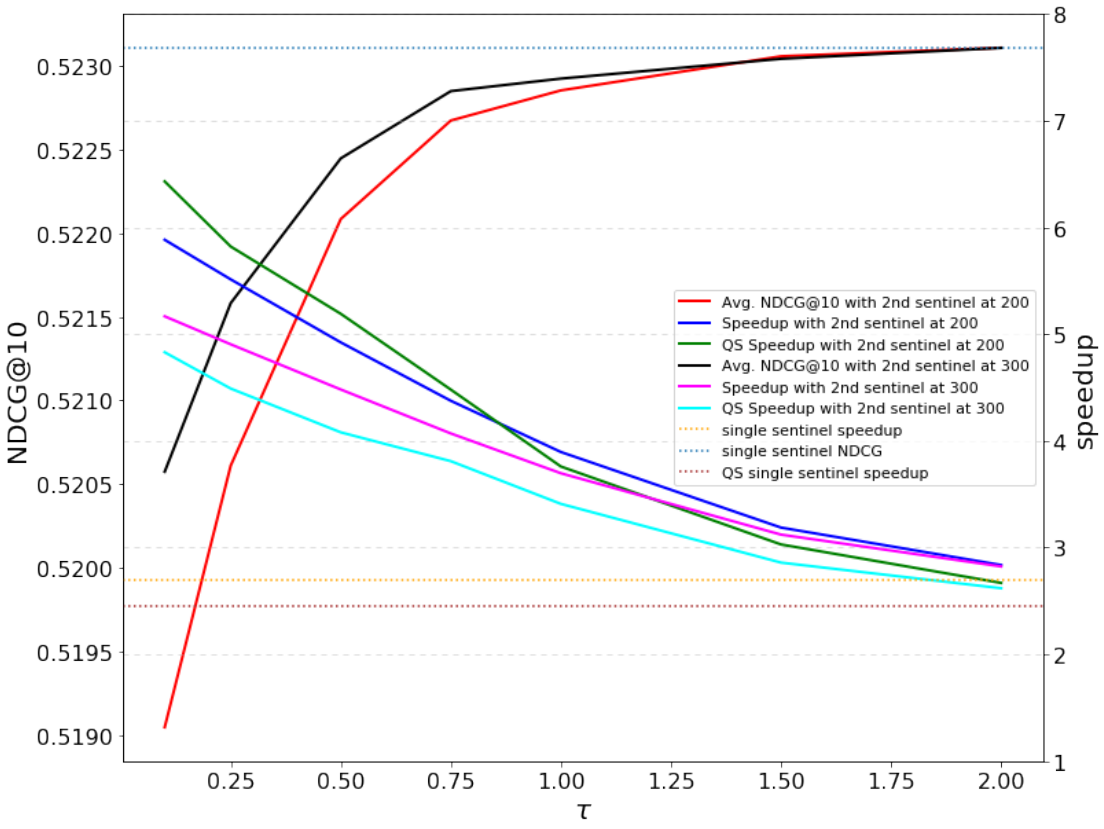


Figure 5.6: Speedup and NDCG drop with a dual sentinel strategy, first strategy is (ERT0.25)@50 and the second are (EPTτ)@200 and (EPTτ)@300

		First Sentinel strategy						
			($\mathcal{E}_1$=ERT, $\tau_1 = 0.25$)@50		($\mathcal{E}_1$=ERT, $\tau_1 = 0.50$)@50		($\mathcal{E}_1$=ERT, $\tau_1 = 0.25$)@100	
$\mathcal{E}_2$	τ_2	h_2	Δ NDCG (-%)	speedup	Δ NDCG (-%)	speedup	Δ NDCG (-%)	speedup
EPT	0.1	150	1.15	6.32	1.18	5.28	1.13	5.46
		200	0.80	5.88	0.81	4.97	0.79	4.84
		300	**0.51**	**5.17**	0.54	4.45	0.47	3.93
	0.25	150	0.74	5.88	0.78	4.97	0.67	5.08
		200	0.50	5.51	0.52	4.71	0.49	4.55
		300	0.32	4.90	0.32	4.26	0.26	3.76
	0.50	150	0.36	5.19	0.39	4.48	0.30	4.49
		200	0.22	4.92	0.22	4.28	0.17	4.08
		300	0.15	4.48	**0.15**	**3.95**	0.12	3.47
	1.00	150	0.12	4.01	0.12	3.58	0.09	3.43
		200	**0.07**	**3.90**	0.08	3.48	*0.05*	*3.21*
		300	0.06	3.70	0.09	3.32	0.05	2.88
	2.00	150	0.03	2.84	0.05	2.59	0.00	2.03
		200	0.03	2.84	0.05	2.59	0.00	2.00
		300	0.03	2.82	0.05	2.58	0.00	1.95
ERT	0.01	150	0.92	6.14	0.95	5.14	0.93	5.35
		200	0.66	5.73	0.66	4.86	0.65	4.76
		300	0.41	5.06	0.41	4.37	0.38	3.89
	0.10	150	0.12	4.16	0.08	3.68	0.07	3.78
		200	0.07	4.04	0.05	3.58	0.04	3.53
		300	0.04	3.82	0.04	3.41	0.02	3.11
	0.25	150	0.03	2.71	0.04	2.49	0.02	2.54
		200	0.03	2.71	0.04	2.49	0.02	2.47
		300	0.03	2.71	0.04	2.49	0.02	2.33
	0.50	150	0.02	2.70	0.04	2.49	0.01	1.64
		200	0.03	2.70	0.04	2.49	0.01	1.64
		300	0.03	2.70	0.04	2.49	0.01	1.64
	0.75	150	0.03	2.70	0.04	2.49	0.01	1.64
		200	0.03	2.70	0.04	2.49	0.01	1.64
		300	0.03	2.70	0.04	2.49	0.01	1.64
EST	0.25	150	1.68	5.85	1.78	5.0	0.25	3.78
		200	1.39	5.46	1.49	4.72	0.21	3.51
		300	1.11	4.86	1.26	4.25	0.22	3.08
	-0.1	150	0.38	4.84	0.44	4.28	0.02	3.03
		200	0.32	4.60	0.34	4.09	0.03	2.88
		300	0.29	4.22	0.29	3.78	0.01	2.63
	-0.5	150	0.11	3.79	0.13	3.46	0.02	2.34
		200	0.05	3.69	0.10	3.36	0.03	2.27
		300	0.04	3.52	0.06	3.21	0.01	2.17
	-0.75	150	0.03	3.27	0.06	2.98	0.01	2.02
		200	0.02	3.23	0.03	2.95	0.00	1.99
		300	0.03	3.16	0.04	2.88	0.00	1.94
	-1.00	150	0.03	2.94	0.05	2.66	0.00	1.80
		200	0.03	2.94	0.05	2.66	0.00	1.80
		300	0.02	2.91	0.04	2.65	0.00	1.78

Table 5.6: Result of simulation with two pruning sentinels with local strategy

5.4 Final Results

$\mathcal{E}_1$	τ_1	h_1	$\mathcal{E}_2$	τ_2	h_2	Document Pruned (%)	Δ NDCG@10 (-%)	speedup
ERT	0.25	50	EPT	0.1	300	90.83	0.57	5.18
ERT	0.50	50	EPT	0.5	300	85.98	0.15	3.48
ERT	0.25	50	EPT	1.0	200	80.30	0.21	3.92
ERT	0.25	100	EPT	1.0	200	80.47	0.13	3.51
ERT	0.10	200	-	-	-	81.27	0.13	2.93

Table 5.7: Results obtained in simulation on the test set

$\mathcal{E}_1$	τ_1	h_1	$\mathcal{E}_2$	τ_2	h_2	Doc. Pruned(%)	Δ NDCG@10 (-%)	Avg. Doc time (μs)	speedup
BWQS vanilla[1]	-	-	-	-	-	0	0	43.73	1
ERT	0.25	50	EPT	0.1	300	90.90	0.57	12.01	4.83
ERT	0.50	50	EPT	0.5	300	86.00	0.15	13.33	3.28
ERT	0.25	50	EPT	1.0	200	80.40	0.21	11.83	3.69
ERT	0.25	100	EPT	1.0	200	80.50	0.13	12.89	3.29 [2]
ERT	0.10	200	-	-	-	81.27	0.13	12.36	2.79 [3]

Table 5.8: Results obtained using QuickScorer on the test set

In tables 5.7 and 5.8 are summarized the results of performing document early exit in the test set. From these results we can say that the observations done on the validation set in the previous section are well maintained also in the test.

Moreover, we observe lower speedups when using QuickScorer and that is absolutely expected since these speedups, contrary to the simulation's, do take into account also the cost of performing the pruning, that is sorting the scorers and comparing them with the thresholds. Another quick note to make is that it is possible that QuickScorer prunes slightly more documents than the simulation. That is due to the fact that NumPy by default uses a 32 bit representation of floating point numbers while QuickScorer uses 64 bits. Given that the decision of stopping a document is done by both as a *greater or equal* comparison, then some documents with a score equal to the pruning threshold, and thus kept in Python, are stopped in QuickScorer.

Lastly, we focus on the two strategies that we picked aside from table 5.8 we can see that both of them have performances more or less equal to the three that we chose from the simulation. Particularly, the one with only one sentinel, which uses 200 as tree block size, even though it scored the whole collection for a much longer period (19% of the ensemble, reflected by its 2.79x speedup) has an average

[1]Block wise quickscorer with 50 as tree block size
[2]speedup relative to a BWQS with tree block size of 100
[3]speedup relative to a BWQS with tree block size of 200

scoring time comparable with the others. Interestingly, if its speedup is computed with respect to the base scorer tree block size of 50, the result would be of 3.53x, which is better than two out of four of the dual sentinel strategies tested and with the lowest drop of NDCG among all of them. Thus, making it a surely valid option when deciding which one to use, confirming once more that the choice of the parameters of the scorer is a fundamental one.

Chapter 6

Conclusions and Future work

In this thesis, we explored the possibility of extending the strategies presented in [18] used to interrupt the scoring process of some documents in predetermined points of an additive ensemble. We did that by allowing the aforementioned strategies to adapt their stopping criteria to the different score distributions of queries. We explored how such ideas behave in an idealized scenario and in a production-like environment. In this work, we also reviewed some of the state-of-the-art ranking models based on ensembles of gradient boosted regression trees and how we can efficiently use them. We then discussed how we implemented the pruning process in a state-of-the-art scoring system and the limitations that we encountered. Ultimately, we summarized our findings in chapter 5, arriving at the conclusion that performing document early exit is a viable option to improve the responsivity of a search engine and that it has, when performed cautiously, only a minimal impact on the quality of the results.

The overall conclusion that we can derive from all this discussion is that it is possible to effectively prune a lot of documents without hindering considerably the quality of the final result, so much so that in our experiments we were able to traverse approximately two thirds of the trees of our ensemble with only 10% of the initial documents and still have a reduction of NDCG@10 of just 0.57%, obtaining a realistic speedup close to 5x. In a more conservative scenario where we would want to limit the decrease of final result quality, we could still achieve speedups of well over 3x with only a marginal loss of NDCG. Furthermore, it seems plausible to be being able to find some strategies that will at least halve the total scoring time without any measurable difference of result quality.

One last important consideration is that all of this discussion could be easily replicated with any additive ranking model since the

early exit functions, either their static or dynamic variants, do not depend on the internal structure of the models composing the ensemble but only on their outputs, making it a very flexible approach.

For future developments, we would like to separate the scorer parameters from the early exit sentinel positioning, to benefit from both the running efficiency of large tree blocks and the pruning capacity of early exit points. Other than that, a possible direct future development of this work is to have a learned model which, given a position and a score vector, computes the right threshold for a determined early exit function. Another interesting research question is to couple these document-wise pruning strategies with a recently presented query-wise approach [30] and observe how the two behave together.

Bibliography

[1] Leo Breiman et al. *Classification and regression trees*. The Wadsworth statistics/probability series. Monterey, CA: Wadsworth, Brooks/Cole Advanced Books, and Software, 1984.

[2] Norbert Fuhr. "Optimum polynomial retrieval functions based on the probability ranking principle". In: *ACM Transactions on Information Systems (TOIS)* 7.3 (1989), pp. 183–204.

[3] William S Cooper, Fredric C Gey, and Daniel P Dabney. "Probabilistic retrieval based on staged logistic regression". In: *Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*. 1992, pp. 198–210.

[4] Yoav Freund. "An adaptive version of the boost by majority algorithm". In: *Machine learning* 43.3 (2001), pp. 293–318.

[5] Jerome H Friedman. "Greedy function approximation: a gradient boosting machine". In: *Annals of statistics* (2001), pp. 1189–1232.

[6] Koby Crammer and Yoram Singer. "Pranking with ranking". In: *Advances in neural information processing systems*. 2002, pp. 641–647.

[7] Yoav Freund et al. "An efficient boosting algorithm for combining preferences". In: *Journal of machine learning research* 4.Nov (2003), pp. 933–969.

[8] Jerome H Friedman and Jacqueline J Meulman. "Multiple additive regression trees with application in epidemiology". In: *Statistics in medicine* 22.9 (2003), pp. 1365–1381.

[9] Robert E Schapire. "The boosting approach to machine learning: An overview". In: *Nonlinear estimation and classification*. Springer, 2003, pp. 149–171.

[10] Chris Burges et al. "Learning to rank using gradient descent". In: *Proceedings of the 22nd international conference on Machine learning*. 2005, pp. 89–96.

[11] Christopher J Burges, Robert Ragno, and Quoc V Le. "Learning to rank with nonsmooth cost functions". In: *Advances in neural information processing systems*. 2007, pp. 193–200.

[12] Zhe Cao et al. "Learning to rank: from pairwise approach to listwise approach". In: *Proceedings of the 24th international conference on Machine learning*. 2007, pp. 129–136.

[13] 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*. 2007, pp. 391–398.

[14] Ping Li, Qiang Wu, and Christopher J Burges. "Mcrank: Learning to rank using multiple classification and gradient boosting". In: *Advances in neural information processing systems*. 2008, pp. 897–904.

[15] Mike Taylor et al. "SoftRank: Optimising Non-Smooth Rank Metrics". In: *WSDM 2008*. Feb. 2008. URL: https://www.microsoft.com/en-us/research/publication/softrank-optimising-non-smooth-rank-metrics/.

[16] Dan Steinberg and Phillip Colla. "CART: classification and regression trees". In: *The top ten algorithms in data mining* 9 (2009), p. 179.

[17] Chris J.C. Burges. *From RankNet to LambdaRank to LambdaMART: An Overview*. Tech. rep. MSR-TR-2010-82. June 2010. URL: https://www.microsoft.com/en-us/research/publication/from-ranknet-to-lambdarank-to-lambdamart-an-overview/.

[18] B Barla Cambazoglu et al. "Early exit optimizations for additive machine learned ranking systems". In: *Proceedings of the third ACM international conference on Web search and data mining*. 2010, pp. 411–420.

[19] Lidan Wang, Jimmy Lin, and Donald Metzler. "Learning to efficiently rank". In: *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*. 2010, pp. 138–145.

[20] Qiang Wu et al. "Adapting boosting for information retrieval measures". In: *Information Retrieval* 13.3 (2010), pp. 254–270.

[21] Hang Li. "A short introduction to learning to rank". In: *IEICE TRANSACTIONS on Information and Systems* 94.10 (2011), pp. 1854–1862.

[22] Tie-Yan Liu. *Learning to rank for information retrieval*. Springer Science & Business Media, 2011.

[23] Nima Asadi, Jimmy Lin, and Arjen P De Vries. "Runtime optimizations for tree-based machine learning models". In: *IEEE transactions on Knowledge and Data Engineering* 26.9 (2013), pp. 2281–2292.

[24] Tao Qin and Tie-Yan Liu. "Introducing LETOR 4.0 Datasets". In: *CoRR* abs/1306.2597 (2013). URL: http://arxiv.org/abs/1306.2597.

[25] Lidan Wang et al. "Learning to efficiently rank on big data". In: *Proceedings of the 23rd International Conference on World Wide Web*. 2014, pp. 209–210.

[26] Claudio Lucchese et al. "QuickScorer: A Fast Algorithm to Rank Documents with Additive Ensembles of Regression Trees". In: *SIGIR 2015: Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*. Santiago, Chile, 2015.

[27] Aleksandra Petrakova, Michael Affenzeller, and Galina Merkuryeva. "Heterogeneous versus Homogeneous Machine Learning Ensembles". In: *Information Technology and Management Science* 18 (Dec. 2015). DOI: 10.1515/itms-2015-0021.

[28] Guolin Ke et al. "LightGBM: A Highly Efficient Gradient Boosting Decision Tree". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. Curran Associates, Inc., 2017, pp. 3146–3154. URL: http://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf.

[29] Claudio Lucchese et al. “RankEval: An Evaluation and Analysis Framework for Learning-to-Rank Solutions”. In: *SIGIR 2017: Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. Tokyo, Japan, 2017.

[30] Claudio Lucchese et al. “Query-level Early Exit for Additive Learning-to-Rank Ensembles”. In: *arXiv preprint arXiv:2004.14641* (2020).