Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Science and Engineering

DIPLOMA THESIS ASSIGNMENT

Student: **Bc. Tomáš Trnka**

Study programme: Open Informatics
Specialisation: Artificial Intelligence

Title of Diploma Thesis: **Implementation of a learning to rank system**

Guidelines:

1) Review existing algorithms for the learning to rank problem.
2) In coordination with an industrial partner, implement data acquisition routines on the production server.
3) Evaluate existing algorithms from the obtained data, choose one and justify your choice.
4) Implement a chosen method as a web-service.
5) Measure the contribution in terms of prediction quality and response time.
6) Optional: Enhance the set of query-independent features using linked-data and evaluate its improvement.

Bibliography/Sources:

[1] C. J. C. Burges, K. M. Svore, P. N. Bennett, A. Pastusiak, Q. Wu: Learning to Rank Using an Ensemble of Lambda-Gradient. JMLR: Workshop and Conference Proceedings 14, 2011.
[2] W. W. Cohen , R. E. Schapire , Y. Singer: Learning to order things. Journal of Artificial Intelligence Research, 1999.
[3] J. Xu , H. Li: AdaRank: a boosting algorithm for information retrieval, Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval, 2007.
[4] C. Burges, T. Shaked, E. Renshaw, A. Lazier, Matt Deeds, Nicole Hamilton, Greg Hullender: Learning to rank using gradient descent. Proceedings of the 22nd international conference on Machine learning, 2005.
[5] Y. Duan: An empirical study on learning to rank of tweets. Proceedings of the 23rd International Conference on Computational Linguistics.
[6] C. D. Manning: An Introduction to Information Retrieval. Cambridge University Press, 2008.

Diploma Thesis Supervisor: MSc. Radomír, Černoch

Valid until the end of the summer semester of academic year 2016/2017

L.S.

prof. Ing. Filip Železný, Ph.D.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 13, 2016

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Science and Engineering

Diploma Thesis

Tomáš Trnka

Implementation of a learning to rank system

May 2016

Supervisor: Radomír Černoch, MSc.

Acknowledgement

I would like to thank to my supervisor Radomír Černoch, MSc. for his guidance, patience and advice. I also would like to thank to our business partner Vojtěch Knyttl for access to the data, valuable suggestions and comments.

I would like to thank my family and my girlfriend for their support.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne .....

Tomáš Trnka

Abstract

The growth of information in recent years rendered the classical information retrieval unsatisfactory. It is no longer sufficient to retrieve relevant data. The retrieved data must be sorted to enable users to find the most relevant information as quickly as possible. More and more companies are dealing with large amount of data, and they are struggling with ordering it for the user queries. We partnered with a business partner and in our work we implemented a searching and learning to rank module for the *GoOut.cz* server. We were working with real world data based on the user inputs. For solving this task, the library providing the state of the art algorithms was used and we were mainly focusing on the features. In our work, we tried to enhance the commonly used features with relational data. Eventually, we identified algorithms that were able to benefit from the enhanced features. We also verified that even with the enhanced features, the most successful algorithms respond within a reasonable time, and therefore, the whole system is usable in practice.

Keywords

learning to rank, information retrieval, machine learning, feature extraction, GoOut.cz, real world data

Abstrakt

Značný nárůst množství dat v posledních letech ukázal limity klasického vyhledávání informací. Už nestačí relevantní informace pouze najít, je potřeba seřadit je tak, aby uživatel mohl získat nejdůležitější data co nejrychleji. Stále více společností se potýká s větším objemem dat a řeší tuto nesnadnou úlohu řazení informací pro své uživatele. Pro naši práci jsme se spojili s obchodním partnerem a vyvinuli modul pro hledání a řazení výsledků pro server *GoOut.cz*. Pracovali jsme s reálnými daty od uživatelů našeho partnera. Při řešení problému jsme použili knihovnu poskytující současné moderní algoritmy a soustředili se hlavně na vytváření příznaků pro dokumenty. V naší práci jsme se pokusili obohatit standardně používané příznaky o relační data. Na závěr se nám podařilo vybrat algoritmy, které byly schopné využít těchto obohacených příznaků. Také jsme ověřili, že i přes větší množství příznaků zvládnou nejlepší algoritmy odpovídat na dotazy v rozumném čase a jsou tedy použitelné v praxi.

Klíčová slova

učení se řazení, hledání informací, strojové učení, vytváření příznaků, GoOut.cz, reálná data

Contents

CHAPTER 1

Introduction

In recent years we have seen an enormous growth of the data being stored and transmitted over the internet. There are several things we have to care about considering such vast amounts of data. The first task is how to store them; the other one that we will be discussing in this work is how to retrieve the right data.

Over the time we were able to see many systems able to store and query documents according to the amount of data. The main changes in this area were mainly caused by world wide web. In the early days it was good enough to retrieve the relevant documents, but for working with databases as big as the internet, this was not longer sufficient. In today applications, the results need to be sorted so that the most relevant will be on top of the list and the user will not have to go through the whole result.

As a result when we nowadays solve the information retrieval (IR) task, in the first place we have to obtain documents that are relevant to the query, and then we would like to present them in a certain order that reflects their relevancy.

A simple information schema[1] that illustrates the IR procedure is shown in figure 1.1. IR can be more formally defined according to [10] as finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers). The IR evolved from its early days when such systems were used only in specific environments to its current daily usage. Today when someone uses 'search' basically in every domain, it means that some sort of IR is used no matter whether we use it when searching the whole web or when looking for a file in our computer.

The interesting part of information retrieval is that we are looking within the data where we cannot identify them by some unique ID; as mentioned before, the data are unstructured.

[1]The schema was created by X7q (talk) - I (X7q (talk)) created this work entirely by myself., Public Domain, https://en.wikipedia.org/w/index.php?curid=25449201

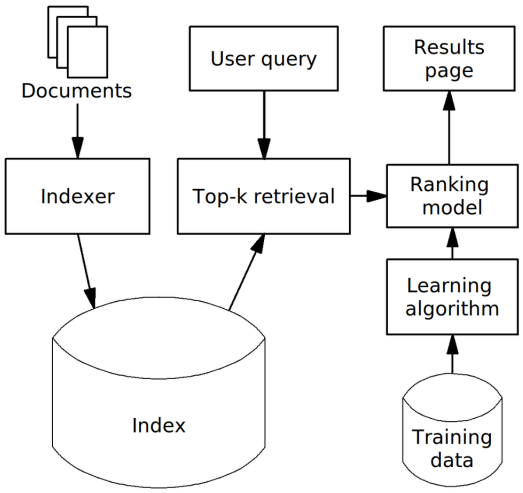

Figure 1.1: Simple IR schema

When we take a look at the 1.1, we can see that any IR system consists of several parts. At first we need the data, i. e. documents that in most cases contain text. We also need some facility that is able to store all these data and that can answer user requests. Most of the time is this part of the task handled by a database. If some results are obtained, we receive a list of results. The results are typically ranked, and their ranks carry the information how relevant each document for the query is. And this relevancy or some other comparable score determines the final result that is presented to the user. If the user is satisfied, then the task is done; otherwise he might wish to refine the query and reiterate.

With this schema in mind we can propose a simple *boolean IR* system. We can imagine such system as a collection of documents and sets of all words. We create a matrix of incidence of each word to the document and the perform queries with operators *AND*, *OR*, and *NOT* since all these queries can be translated into binary vectors and compare with the incidence matrix. This system would work but it is rather memory-consuming, and it would not tell us whether the word occurred in document just by accident or whether it is the topic of the document; moreover we have no idea how to sort the results.

As we can see, a straightforward approach would be infeasible and we are in need of other, more sophisticated information retrieval systems. We would like to be able to perform *ad hoc IR*, where we will be able to provide reasonable results even if a user is unable to specify a query in such manner that it exactly matches some of the keywords.

Unfortunately, for real world data we can build the term incident matrix in such a naïve way as described above. We will use some smart concepts; for example, one of the key concepts in the area of IR is called *inverted index*. This key principle of IR will be described later.

The main focus of this work was to implement a server module for our business partner. Our business partner represented by Vojtěch Knyttl is a cultural guide *GoOut.cz* which aggregates cultural events in a user's surrounding and it is a platform for ticket sales.

As many internet services that grew rapidly in recent years they provide much more content than at the beginning and they need to deal with the common situation. We agreed to build a standalone server application that will be able to serve as a standalone IR system providing understandable API and responding within a reasonable amount of time.

We will have to be able to deploy the application into the context where the current solution of a business partner system is running. This context consists of a Java server and uses a relational MySQL database. Therefore, our application is also written in `Java`, which is very well suitable for large server solutions, namely thanks to its just in time compiler (JIT) which can benefit from long-term running applications, and our system will use as a storage the current `MySQL` database.

The next chapter 2 focuses on the theory of learning to rank. We will discuss in more detail its key principles; the chapter will also present several states of the art algorithms and metrics that are used for LTR evaluation.

In the next chapter 3 we will take a closer look at the technologies used both by our business partner or by this work. We will examine the data format, how the data is preprocessed and how the core part of our work was implemented in technical detail. The feature extraction is also described in this chapter.

The following chapter 4 includes the discussion about additional relational features that were to the dataset. Furthermore, we will provide z description of the data sources and justify our assumptions why we used them.

In chapter 5 we will describe all the experiments we conducted with several different setups. The configuration concerning algorithms, metrics and features will be discussed and evaluated.

The last chapter 6 summarises the whole work, concludes the results and suggests a future work.

CHAPTER 2

Learning to rank algorithms and metrics

In this chapter the general overview about the learning to rank will be provided. Then, we introduce the metric how to evaluate ranking. Afterwards, the two basic approaches how to solve LTR task will be presented and in the rest of this chapter we will take a closer look at the algorithms implemented by *RankLib* library and used in our experiments.

The task of learning to rank or machine-learned ranking is defined as ranking documents for IR systems. We have a list of documents; each document is assigned either some ordinal commonly called relevance grade (in most used applications it spans form *0 – not relevant*, to *4 – highly relevant*) or binary rank *relevant* or *non relevant*. Between the relevance grades there exists total ordering. This unfortunately does not hold for the documents. Sometimes there are not enough data to assign a label to all the documents; therefore, the relation between documents is a partial ordering.

Definition 2.1 (Partial ordering)**.** A partial order is a reflexive, antisymmetric and transitive binary relation $\preceq$ over a set $\mathcal{S}$. A set with partial order is called a partial ordered set (poset). If for elements a and b from the set holds $a \preceq b$ or $b \preceq a$ we say that they are comparable, otherwise incomparable.

The learning to rank algorithms then try to create such permutation of the test set list that in some way resembles the train set. The vast majority of modern applications of LTR is working with the ordinal relevance grades although some big systems still relying on binary relevance survived till today. The intuition behind this task can be described as such that we would like to have the documents with the highest relevance grades (scores) as high on the resulting list as possible.

Definition 2.2 (Learning to rank, according to [8])**.** Let $\mathcal{Q}$ be query set and let $\mathcal{D}$ be document set. Suppose $\mathcal{G}$ is a label set with corresponding relevance grades. There exists total ordering between grades. Let $\mathcal{D}_\rangle$ be a list of documents belonging to query q_i. A feature vector $x_{i,j} = \phi(q_i, d_{i,j})$, where ϕ stand for feature function is created for each document query pair. We aim to train a ranking model $f(q, d) = f(x)$ that can assign score to a given query document pair. Let π_i be a permutation on D_i, then the ranking is to select such permutation π_i for given data with $f(q_i, d_i)$ that optimizes some criteria. Such criteria can be a partial ordering among documents or some specialized function.

In practice, all the algorithms as any other machine-learning application rely on features that describe all the documents. The algorithms learn how to order based on these features the list. The ordering is done in such a way that the algorithms transform the feature vectors into one number for each document, and according to this final score the list of results is ordered, i.e. the learning to ranks algorithms do not try to predict the relevance grade of the documents but rather their ordering.

For LTR we must rely on the IR part of the process that all the relevant documents were chosen. Then we process the document features with some algorithm and the underlying model predicts scores for items on the list. In the end, we know the ordering among the documents based on these scores.

The algorithms used for LTR can be split into two groups based on their approach to the data – how they create the model for ranking. There are a list-wise approach and a pair-wise approach. The pair wise approach tries to optimize the partial ordering among the pairs of the documents in the list. On the other hand, the listwise approach is concerned about the ordering of the whole list.

Before we begin with the algorithms, it is necessary to introduce metrics used in LTR. The modern algorithms try to optimize these functions directly, and thus we feel the obligation to explain what is being optimized and how the score reflects the ordering. Please note that in our work we use a metric for function evaluating the ordering of the list and not a metric in mathematical sense.

2.1 Metrics

There are several metrics that can be used to evaluate the performance of the rankers. We can divide them into two groups the older one developed in early days of IR works with binary classified data in relevant/non relevant meaning. The more recent metrics work with relevance grades that are assigned to each document in the result set. Most of the metrics use cut of parameter which mean that we look at the first N entries only in the result set. We will describe both groups although the historical metrics are nowadays not used very

often. However in our context they can be useful because for us, the binary classification can be viewed as a click or a non-click.

There are two models: the first is called position model, and it takes into account only the positions of the documents and their relevance. On the other hand, the cascade model tries to mimic the behaviour of a human going through the list. In position model we assume that for the document on i-th position the preceding documents are irrelevant. This is not true for the cascade models which suppose that the user is looking at the document because the previous failed to fulfil its needs. As we can see, the cascade models taking into account not only the relevance but also the relation between documents.

2.1.1 Discounted Cumulated Gain

Discounted cumulative gain is one of the most used criteria for ranking quality. With DCG we can measure usefulness of a document based on its position in the list. This metric assigns some value to each document, and the result is sum of document values in the result.

We can assume that we obtained a vector that describes the relevance grade of the documents. This metric is an extension of a Cumulative Gain which is defined as follows:

Definition 2.3 (Cumulated Gain by [9])**.** Let G be a list of relevance grades that corresponds to the list of results, then Cumulative Gain is computed as

$$CG = \begin{cases} G[1] & \text{if } i = 1 \\ CG[i-1] + G[i], & \text{otherwise} \end{cases}$$

This metric can provide some information but it completely ignores the ordering of the documents in the part of the list we are interested in. So the list cropped at 10th position with the most relevant document at the end would obtain the same score as the one where it would be on top of the list.

This lack of differentiation led to the new measure

Definition 2.4 (Discounted Cumulated Gain by [9])**.** Let G be a list of relevance grades that corresponds to the list of results and let CG be Cumulative Gain as defined before, then

$$DCG = \begin{cases} CG[1] & \text{if } i \leq b \\ DCG[i-1] + \frac{G[i]}{\log_b(i)}, & otherwise \end{cases}$$

This new measure introduces the discounting factor that grows as the position in the list rises. It is crucial not to discount the relevance grade too aggressively. Discounting by the

logarithm of the position is reasonable since the document on 1000th position would still obtain about one tenth of its relevance grade. The base of the logarithm b is usually chosen as 2.

The results may vary depending on the size of query; therefore it is convenient to introduce a measure that is cross query comparable.

Definition 2.5 (Normalized Discounted Cumulated Gain by [9])**.** Let $IDCG$ be DCG for ideally ranked list where DCG is defined as before, then

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

As we can see, the DCG is normalized by a value of the corresponding so called ideal measure $IDCG$. It is worth noting that in case of partial relevance feedback the ideal ordering may be unavailable.

For better understanding we provide an example how this metric is computed. Let us compute the value of above mentioned metrics for the list of relevance grades $[3, 2, 4, 3, 1, 2]$. The results are shown in table 2.1.

order	r.	i. r.	$\frac{\text{ranking}}{\log_2(i)}$	$\frac{\text{ideal r.}}{\log_2(i)}$	CG	DCG	Ideal DCG	NDCG
1	3	4	-	-	3	3	4	0.75
2	2	3	2	3	3+2	3+2	4+3	0.71
3	4	3	2.52	1.89	5+4	5+2.52	7+1.89	0.84
4	3	2	1.5	1	9+3	7.52+1.5	8.89+1	0.91
5	1	2	0.43	0.86	12+1	9.02+0.43	9.89+0.86	0.87
6	2	1	0.77	0.39	13+2	9.45+0.77	10.75+0.39	0.92
results					15	10.22	11.14	0.92

Abbreviations r., i.r. stand for relevance grade and ideally ordered list of relevance grades respectively.

Table 2.1: Computation of CG, DCG and NDCG

2.1.2 Expected Reciprocal Rank

According to the [3] the Expected Reciprocal Rank is a metric defined as the expected reciprocal length of time during which the user will find a relevant document. This metric can be viewed as an extension to standard Reciprocal Rank.

The standard Reciprocal Rank (RR) is used for IR, and it is defined in such way that the correct answer will receive a value which is proportional to inverted value of its order. The Reciprocal Rank over multiple queries is computed as a Mean Reciprocal Rank which is simple mean of their results.

The simple example in the figure 2.2 shows the principle of RR.

query	results	RR
query1	resultQ1_1, resultQ1_2, resultQ1_3, **resultQ1_4**	$\frac{1}{4}$
query2	resultQ2_1, **resultQ2_2**	$\frac{1}{2}$
query3	**resultQ3_1**, resultQ3_2, resultQ3_3	$\frac{1}{1}$
	$(1/4 + 1/2 + 1/3)$	3/8

Table 2.2: Computation of RR, the items in bold are the desired correct results.

The reciprocal metric is a subject of many discussions. It is particularly suitable for a system in which an only one response is correct. In our case this would exactly represent our data where we have only the information that user clicked on one particular query. But from a general point of view we can easily come up with a situation where exactly the same query should yield different 'most' relevant results either for only one user or for multiple users which would have been even easier. The ERR solves this problem.

In the first step we have to define the probability that a user will find the relevant document as a function of relevance grade. The authors suggest to choose similarly to NDCG mapping function as

$$R_i = \mathcal{R}(g_i), \text{ where } \mathcal{R}(g_i) = \frac{2^{g_i} - 1}{2^{g_{\max}}}, g_i \in \{0, \ldots, g_{\max}\}$$

In the previous equation the g_i represents the relevance grade of i-th document, the $g_{\max}$ value is typical 4 if a 5-point scale is used.

Definition 2.6 (Expected Reciprocal Rank as defined by [3])**.**

$$\text{ERR} = \sum_{r=1}^{n} \frac{1}{r} P(\text{user stops at position r}) = \sum_{r=1}^{n} \frac{1}{n} \prod_{i=1}^{r-1} (1 - R_i) R_r$$

There is a computational example for ERR as well but we needed the computation for justification of our approach while commenting the real data. Therefore the example is in the next chapter on page 17.

2.2 Pairwise algorithms

This type of ranker solves the approximation of the ranking task. It tries to optimize pairwise ordering among pairs. This task is more general then ordering within the list only. We can come up with a situation where such ordering holds

$$U1 \prec U2, U2 \prec U3, U3 \prec U1$$

and we would like the ranker to be able to learn such ordering. In reality we would not be able to learn the correct ordering entirely and the ranker itself learns an approximation of correct ordering. In general, every algorithm models this approximation differently; they will be described later.

2.2.1 RankNet

The main idea behind this ranker is that we need an easily differentiable function that models the ordering and can be optimized by its derivative at the same time. In this algorithm, a sigmoid function is used – it is very well-know and often used in area of neural networks. We demonstrate the learning procedure on two queries: example U_i, and U_j. Every query is by the model mapped to some score s_i, s_j. Let denote an event that U_i should precede U_j as $U_i \triangleright U_j$. The outputs of the model are mapped to learned probability via sigmoid function

$$P_{ij} \equiv P(U_i \triangleright U_j) \equiv \frac{1}{1 + e^{-\sigma(s_i - s_j)}}$$

As a cost function the cross entropy is used - it measures the deviation of the learned model from the desired. Let $S_{ij} \in \{-1, 0, 1\}$ model the ordering of examples U_i and U_i. The cost is than defined as

$$C = -\bar{P}_{ij} \log Pij - (1 - \bar{P}_{ij} \log(1 - P_{ij})$$

where the ordering is known, and therefore, $\bar{P}_{ij} = \frac{1}{2}(1 + S_{ij})$

We can put there two equations together and obtain the gradient

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

that can later be used for updating the weights of a model. The precise formulation of the weight updates is:

$$w_k \to w_k - \eta \left(\frac{\partial C}{\partial s_i} \frac{\partial s_i}{\partial w_k} + \frac{\partial C}{\partial s_j} \frac{\partial s_j}{\partial w_k} \right)$$

The update of the criteria:

$$\Delta C = \sum_k \frac{\partial C}{\partial w_k} \delta w_k = -\eta \sum_k \left(\frac{\partial C}{\partial w_k} \right)^2 < 0$$

These are all parts we needed to create a probability model, we obtained a function that can be optimised with a neural network which will eventually assign the scores.

2.2.2 RankBoost

The RankBoost algorithm is in general a boosting algorithm. It works on pairs of documents and tries to optimize the number of mismatched pairs.

Following definitions are direct quotes of [7]. 'For any pair of instances $x_0, x_1, \phi(x0, x1)$ is a real number whose sign indicates whether or not x_1 should be ranked above x_0, and whose magnitude represents the importance of this ranking.'

For the learning task the authors suggest to form a distribution D that is formed as

$$D(x_0, x_1) = c \cdot \max\{0, \phi(x_0, x_1)\},$$

where c is chosen so that the sum over x_0 and x_1 sums to 1. With this distribution being defined the main loop of the algorithm learns weak learners using this distribution D; then, the normalization and the update step take part. After T steps, the output is a linear combination of weak learners as expected for boosting algorithm.

2.3 Listwise algorithms

The main difference of Listwise algorithms from Pairwise ones is their ability to create a model and perform ranking in the scope of the whole list and not only pairs of documents. As usual, there are several advantages and disadvantages. The pairwise approach can be biased towards queries that have more pairs. This behaviour was experienced and described in [2]. On the other hand when dealing with the list as a whole we must consider all the possible permutations of the list. This would generally introduce intractability; therefore for example the authors of Listwise algorithms have to overcome this constraint.A detailed description of their and other solutions is provided in following subsections.

2.3.1 ListNet

This algorithm follows the basic ideas of RankNet. But the authors decided to model not a pairwise probability but a probability of ordering the whole list. This idea forced them to introduce two main concepts.

The first one is a permutation probability and the second one, used to overcome complexity issues is a top k probability.

Let π be a permutation, ϕ is an increasing and strictly positive function and $s(i)$ is a score of item i. Then according to the [2], the permutation probability is defined as

$$P_s(\pi) = \prod_{j=1}^{n} \frac{\phi(s_{\pi(j)})}{\sum_{k=j}^{n} \phi(s_{\pi(k)})},$$

where the $s_{\pi(j)}$ denotes the score of an object at position j of permutation π.

For the notion of top k probability, the top k subgroup is defined which collects all permutations where desired k items are among the top k elements. This reduce the size to $\frac{n!}{(n-k)!}$. Then the probability is computed from the permutations belonging to this subgroup.

The model for ranking is compound from following parts. The exponential function becomes ϕ function in our model; the cross entropy is used as a metric. The distribution is modelled by Neural network and its weights are updated with Gradient Descent.

2.3.2 AdaRank

This algorithm differentiates from the predecessors (in time of its invention) by its ability to optimize directly the performance measures such as NDCG or ERR. As the name suggest this algorithm belongs to the family of algorithms using boosting, which means that the ranker uses several weak classifiers. The learning follows the same pattern as AdaBoost algorithm. Once again, the final model is a linear combination of weak rankers.

This algorithm in general follows the structure of AdaBoost algorithm. It takes two meta parameters on the input: one is the number of rounds, i.e. how many weak rankers the algorithm will create, and the second one is an error function. The authors noted that we can directly use some IR metrics as such a function. The original implementation takes individual features as weak rankers. Each weak ranker is assigned its weight in the resulting linear combination based on the count of queries that it is able to improve.

The main difference from the original AdaBoost is that the distribution of the weights on training instance is not created from the current weak learner / ranker, but instead from the model created so far.

2.3.3 Coordinate Ascent

This whole algorithm using Coordinate Ascent was introduced by the author or the *RankLib* library and presented in [4]. Two major steps are sought during the run, the first is to reduce the number of features because sometimes the bigger model induces more complexity by considering useless features. The smaller model also tends to have a better performance. The second reason is feature extraction. The feature extraction is quite a common task in the area of machine learning but according to the author of paper [4], it was solved quite rarely in the ranking area.

The feature reduction is based on non-overlapping subset of the original set of features in greedy way. First step is to build an undirected graph of subsets of features that at most as big as the beforehand chosen size s. There exists an edge where one of the node can be created from another by adding exactly one feature. Then, this graph is traversed with Best-First-Search procedure and if in R last rounds the result was not updated, then the subset is returned. We repeat the procedure until the whole set of feature is covered.

After this step the Coordinate Ascent algorithm for optimizing multivariate objective function is applied. It cycles through these sets of features and tries to optimize one while keeping the others fixed. As a non-derivative this algorithm suffers from possibility of getting stuck in local optima. We can overcome this danger by random restarts because such greedy algorithms are quick.

2.3.4 MART

The MART algorithm's model belongs to category of boosted trees and its output is created as a linear combination of set of regression trees.

Regression tree is a rather simple model which is often appreciated because of simplicity and interpretability. The regression tree structure allows us to partition the feature space into partitions containing objects that are in some sense similar. We can apply multiple criteria to split the space into subspaces represented by nodes in a tree; the common criteria is Gini impurity measuring probability of mislabelling the data randomly chosen from the data set with probability p.

$$GI = \sum_j p_j(1 - p_j)$$

After the partitioning the models are fitted to the areas represented by leaves. The models are simple; partly because we want them to be simple, partly because of the partitioning that already constrained the prediction space. The model can be as simple as mean of the values belonging to the partition or linear regression.

MART's output function can be defined as

$$S_N = \sum_{i=1}^{N} \alpha_i f_i(x)$$

where a_i are coefficients belonging to the trees f_i; both of these parameters are learnt during the run of the algorithm.

Every other tree is constructed in such way that the algorithm optimizes the loss function, i.e. each tree models the gradient of the cost. We assign these gradients as leaf values of the tree.

2.3.5 LambdaMART

This model is compound from several previous algorithms. The first algorithm is called LambdaRank. It was not mentioned because it is not implemented in *RankLib*. From the general point of view it is a sped up version of RankNet. Its authors noticed that the gradients can be precomputed and they named them lambdas. In previous subsection we described MART algorithm. Since the algorithm also takes the advantage of differentiation, we clearly can enhance it by the same concept used in LambdaRank.

Let us examine LambdaRank algorithm in more detail. As we have seen before, the formal definition of the criteria and weight updates in the RankNet algorithm so we will update them to get to the main idea of the LambdaRank.

In RankNet we described how the weights of the model are updated; the authors of [1] noted that important part of the update can be factored out. More precisely

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

After this factorization we can define Δw_k as

$$\Delta w_k - \eta \sum_i \lambda_i \frac{\partial s_i}{\partial w_k}, \text{ where } \lambda_i = \sum_{i \text{ precedes } j} \lambda_{ij} - \sum_{j \text{ precedes } j} \lambda_{ij}$$

The aggregation of weights for updates can dramatically speed up the learning process.

The LambdaMART algorithm is a composition of above described LambdaRank and MART algorithm. We compute the lambdas and assign them as leaf values to the MART. Then we run the algorithm.

2.3.6 Random Forests

Random Forests are, as name suggests, ensembles of trees. The regression trees are constructed as mentioned before. This algorithm uses technique named bagging. Bagging can be described as averaging the results learned on a subset of learning data. The averaging the results increases the robustness because we train each part on different portion of input data, and indeed the random forests are known to be more prone to overfitting.

This model was used during Yahoo competition in 2010 as an alternative to much more computational demanding models with a reasonably good performance.

CHAPTER 3

Obtaining data and its preparation

To obtain real world data and to perform experiments we partnered with industrial partner GoOut.cz represented by Vojtěch Knyttl. They provided us logs of the search queries and the clicks done be their users.

In this chapter, the structure of the data is described; how the data were processed; which information we extracted from the logs. Then we present the infrastructure of our project, all the concepts and important libraries we used. The last part is dedicated to the feature extraction from the data.

3.1 Data format

The data that we obtained from industrial partner were in form of text logs created by standard logging tool `log4j`. Each log consists of several parts. Two examples of these logs are listed below in 3.1. Please note that the data are not very well formatted; they are simply the text output from the current application. This lack of structure was caused by the insufficient requirements we were able to demand from our business partner in the beginning of our work.

Our business partner recognises three types of documents. As a portal informing about cultural events they store informations about *performers*, *events* and *venues*. We will be working with three types of documents as well.

The first part is the date when the query was performed. Then after INFO there might or might not be the email of the user who performed the query. Then the part with the results comes. All the words between the keywords `SEARCHREMOTE` and a word form a set of keywords `performer, event, venue` succeeded by its `ID` from database forms the query. When some of these keywords appeared, then the user performed a click on a certain result.

```
2014-12-10 22:08:13.115 INFO john.doe@email.com 13064039501
SEARCHREMOTE: hold performer 2928 [{performer=2928}, {event=357632}]

2015-03-10 22:06:49.450 INFO 31312936378 SEARCHREMOTE: dj wich null null
[{performer=15296}, {event=367776}, {event=327671}, {event=349971},
 {event=231885}, {event=354197}]

2014-08-18 11:41:25.705 INFO user@goout.cz SEARCHREMOTE: null null null
```

Figure 3.1: Three logs of performed queries

As mentioned before the data are separated into three categories. In the case when either the user had not chosen anything or any other situation occurred, they are replaced with `null`. The rest of the log is the result set belonging to the query. It consists from types of the results and their corresponding IDs.

The data are parsed with a regular expression that by itself handles malicious logs. As a malicious log we consider such queries that were responded with an empty result set. We are also not concerned about queries where the user did not click on anything. We decided to ignore these cases because in our logging we cannot gain any information from such event, we unfortunately do not log the time spent on the page or the cursor position; these events would provide some at least additional info. When the log is considered malicious, it is simple dropped out and ignored for the rest of the process. Such logs might look as bad as shown in the third example in 3.1. These data can contain zero information about search.

Initially we iterated through the data and stacked all results that share the same query one after another. We were convinced that it is a reasonable assumption to have an only list of results per query. After conducting several experiments we were less convinced.

We realized that the queries are by their nature non-static. For example the search naturally does not respond with events that happened in the past. Therefore, we can easily obtain a totally different result of events for the same query in different point in time. The same holds for venues: they can emerge or disappear in course of time as well.

We recognized that while merging the queries together we are shifting the ordering of consecutive queries. All the results suddenly appear on their position plus length of all results retrieved earlier for the same query. This led to realization that by merging several queries together we artificially improve the results of selected metric for the baseline database search. For most of the metric it is a big difference whether the list contains only one relevant document or two documents. The following example of computation ERR will justify this reasoning.

In this example we are using the *ERR* metric described in previous chapter. Let the computation begin with a small query a:

$$\begin{aligned} ERR & \quad \text{for result with relevance grades } [3,2,4] \\ a &= 1/1 * 7/16 + \\ & \quad 1/2 * 3/16 * (1 - 7/16) + \\ & \quad 1/3 * 15/16 * (1 - 7/16) * (1 - 3/16) \\ &= 0.633 \end{aligned}$$

We continue with second query b with slightly different relevance grades

$$\begin{aligned} ERR & \quad \text{for result with relevant grades } [3,1,2] \\ b &= 1/1 * 7/16 * + \\ & \quad 1/2 * 1/16 * (1 - 7/16) + \\ & \quad 1/3 * 3/16 * (1 - 7/16) * (1 - 1/16) \\ &= 0.488 \end{aligned}$$

We concat the queries and compute *ERR* once again.

$$\begin{aligned} ERR & \quad \text{for result with relevant grades } [3,2,4,3,1,2] \\ c &= 1/1 * 7/16 * + \\ & \quad 1/2 * 3/16 * (1 - 7/16) + \\ & \quad 1/3 * 15/16 * (1 - 7/16) * (1 - 3/16) + \\ & \quad 1/4 * 7/16 * (1 - 7/16) * (1 - 3/16) * (1 - 15/16) + \\ & \quad 1/5 * 1/16 * (1 - 7/16) * (1 - 3/16) * (1 - 15/16) * (1 - 7/16) + \\ & \quad 1/6 * 3/16 * (1 - 7/16) * (1 - 3/16) * (1 - 15/16) * (1 - 7/16) * (1 - 1/16) \\ &= 0.637 \end{aligned}$$

$$\frac{0.633 + 0.488}{2} = 0.5605 \neq 0.637$$

From the results we can clearly see that the mean of the ERR for results a and b is not the same as the result for query c. And that by concatenating the result we can both improve and decrease performance of the baseline solution.

We also implemented the import into the MySQL database, the database structure managing this part of the whole business partner project is shown on schema 3.2. The database was chosen as a more suitable storage because of the server environment where the application is running. So far only one powerful machine is running but in future where scaling might be necessary we would have the logs scattered across multiple machines and it

Figure 3.2: Schema of part of the database

would be difficult to merge them. The database therefore provides a central storage with a natural remote access. Moreover, the database supports a concurrent access. The database also allows us to examine more thoroughly the queries and the results. We can easily select a subset of the logs based on their date, on the size of the result set, or on the query itself. These actions would be much less convenient and probably more time-demanding while working with the plain text file.

3.2 Used Technology and Frameworks

This section contains a list of tools and utilities used in our work. Each used library or a framework is followed by its short description and the implementation specifics details are provided.

3.2.1 Spring Framework

This open source framework is a popular alternative, replacement or enhancement to JavaBeans model. The JavaBeans is a concept that requires the class to have several properties. These properties of such class are:

- implement either Serializable or Externalizable
- to have a constructor which takes no parameters
- all properties should be private and must be accessible via public getter and setter methods.

This framework provides an inversion of a control container for the Java platform. The inversion of control is a design principle that inverts the classical paradigm where we have a program which calls libraries and delegates the tasks. For the inversion of control it is typical that all the events are invoked by the framework – typically graphical user interface – and the programmer fills blank spots, i.e. the actions that the framework will trigger after a certain action.

This approach is very useful when maintaining big projects that should be properly tested. It is quite impossible to trace an exception if one appears, for example while initialising a complex program. In this paradigm we leave the heavy work for the framework; the framework does not allow its parts to instantiate the dependencies among themselves and provides them rather as a service. In Spring framework this behaviour is controlled by annotation `@Autowired`.

To follow the requirements of the framework we must annotate every class that comply with the concept of bean with `@Component` annotation. In our concept most of the classes in out project are declared in such way because we would like to use them across the whole context of application.

We are aware of not using the full potential of Spring framework but we appreciated even the subset of features we are using. The main features we benefit from are a simple configuration with configuration files that enables using with ease the same application in different contexts - production server and development server. The next feature is handling the dependency injection - we do not have to take care about creation of the objects that interacts between themselves, the framework takes care of it. And at least it also formalizes the modular concept used by our application. This concept is essential while building a part of the running system and also provides easy maintainability or maybe even substitutability in the future.

3.2.2 Database Layer

Every application at a certain dimension needs a data storage. While working with a database as in our case, there is a lot of boilerplate code we have to write to connect to the database, to retrieve data and store the information. We also have to take care about the data consistency and other security issues. To overcome this overhead, the database layer and Object Relation Mapping (ORM) techniques are commonly used. There are frameworks and libraries that take care about the transactions, table creation that corresponds to the objects used in the applications and more. In our case we rely on the Hibernate framework and a custom layer written by our business partner Vojtěch Knyttl called Maite.

Hibernate

Hibernate is according to its web page[1] concerned with data persistence as it applies to relational databases (via JDBC). It can be used as a tool enabling to develop persistent classes following natural Object-oriented idioms including inheritance, polymorphism, association, composition, and the Java collections framework. Hibernate requires no interfaces or base classes for persistent classes and enables any class or data structure to be persistent.

In our work we use only a small fraction of its functionality, namely we use only escaping of the queries to the database and its internal ability to transform data types from the database into to Java ones, i.e. we do not have to convert from `MySQL` Date to Java Date object while reading and vice versa while writing.

Maite Database Layer

This library creates a thin layer between the programmer and the database. This layer was developed by our business partner and it mainly provides methods for more convenient handling the results form database and helps with the domain specific data. For example it simplifies creation of Java's `HashMaps` and introduces into the Java environment some features known from other languages such as continual dot notations where we can chain commands because the object's methods execute a certain action and return a reference to the object itself. This also saves time and space while writing the code that handles the communication with the database.

3.2.3 XMLRPC server

The core server part is taken from the standard library managed by Apache Foundation. It is a Java implementation of XML-RPC server[2]. This implementation is according to our business partner a current business standard and it will be able to handle the deployment load with thousands of requests per second in peak times as it is.

3.2.4 Remote Procedure Call

The whole system of our business partner is being developed as a server which is a common solution for a web service. As it is typical for a web service, it implements client-server architecture. It follows this pattern not only from the user's and provider's perspective but also from internal point of view. All the modules are also implemented as servers and communicate with each other.

[1] `http://hibernate.org/orm/`

[2] `https://ws.apache.org/xmlrpc/index.html`

Remote procedure call idiom is a logic extension of the local procedure call, what in general means that the function we are calling has not to be in the same program, same computer, system et cetera. We can call a procedure that resides somewhere else and we obtain a response from a remote location. The remote calls are usually in orders of magnitude slower than the local calls, and generally less reliable.

Such a call must be executed in a certain manner; therefore, we need to use a specified protocol. RPC follows the request-response schema. The connection is initiated by the client; client sends a specific request message to a server. The server must be known beforehand; the client has to know who he is communicating with.

The whole sequence that realises the RPC schema is shown in figure 3.3

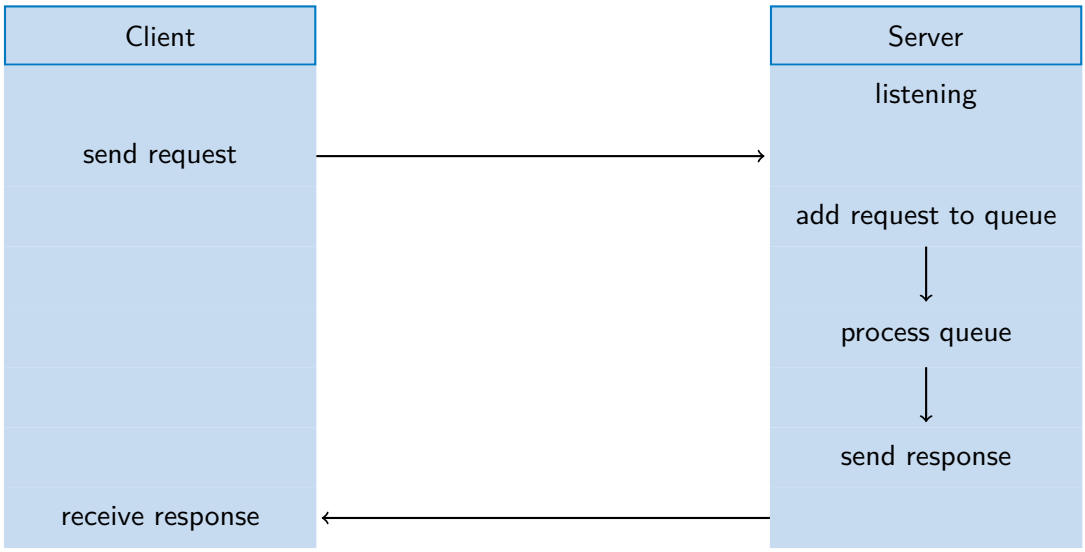

Figure 3.3: XML-RPC schema

The exact specification of RPC we used in our implementation is called XML-RPC. The idea behind adding the XML schema to the RPC is that this data format provides human readable and understandable structure compared to the binary protocols. The example of how such request looks like is in the figure 3.4

For such query the system would respond with a response as shown in the figure 3.5. Note the `tag` method response which determines the message type; the rest of the message looks similar to the request.

In our system there are basically two types of tasks – the ones that must be executed as fast as possible and return results to the user, and the others that can run asynchronously and take long time to execute. Example of the first query would be a response to the searching subsystem how to order the resulting list and to the second group such tasks belong as parsing the logs, invoking the learning and other preparation of the data.

```
<?xml version="1.0" ?>
<methodCall>
    <methodName>parseLogs</methodName>
        <params>
            <param>
                <value>
                    <string>/inputs/data_24_04_2015n1.txt</string>
                </value>
            </param>
            <param>
                <value>
                    <string>/inputs/RL_input.txt</string>
                </value>
            </param>
        </params>
</methodCall>
```

Figure 3.4: XML-RPC call example

```
<?xml version="1.0"?>
<methodResponse>
    <params>
        <param>
            <value>
                <int>15866607</int>
            </value>
        </param>
    </params>
</methodResponse>
```

Figure 3.5: XML-RPC request example

As we can see the XML version is stated in the header of each message, the tag specifies the request following this declaration. The first is the name of the method encapsulated between tags `methodName`. All the parameters for the demanded method are stated in section `params` with specified types. The types we can pass to methods are shown in table 3.6; this is important because it puts some limitation but it also enables communication for example between different programming languages.

name and description	example
Integer - standard integer value	`<int>2017</int>` `<i4>2017</i4>`

Array - array of values	`<array><data><value>`
	`<string>/inputs/data.txt</string>`
	`</value></data></array>`

Bolean - logical value, either 0 or 1	`<boolean>0</boolean>`
Double - floating point number with double precision	`<double>2017.4</double>`
Stirng - must comply with the encoding	`<string>parseLog</string>`

Base64 - base64 encoded data	`<base64>`
	`a2RvVG9Eb2NldGxBelNlbUplRnJhamVyCg==`
	`</base64>`

Struct - associative array	`<struct><member>`
	`<name>key</name>`
	`<string>value</string>`
	`</member></struct>`

Time - date formatted according to ISO 8601	`<dateTime.iso8601>`
	`20000101T00:00:00+01:00`
	`</dateTime.iso8601>`

Figure 3.6: XML-RPC data types

3.2.5 Representational state transfer (REST)

Previously introduced XML-RPC is one particular implementation of an architectural style called Representational state transfer. To quote one of the founder or REST idea Roy Thomas Fielding, he in his dissertation thesis [6] describes Representational State Transfer (REST) as '*an architectural style for distributed hypermedia systems. REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.*'

The criteria are as follows:

- Client–server (model) - This constraint ensures that both server and client are not concerned about each other else than through a specified communication channel. It means that the server and the client are independent on each other as long as the interface remains the same.

- Stateless - This requirement ensures that all the information needed for the server is stored in the request and it does not need to store any client context.
- Cacheable - This part of REST while correctly configured can decrease client-server interactions and improve performance. The crucial part is that the server responses must state its cacheability to prevent the client work with an obsolete version of response.
- Layered system - It must be undistinguishable for the client to say whether he communicates with the server or whether there are for example some load balancing mechanisms along the way.
- Uniform interface - The fundamental constraint separates the resources present on the server and the data returned to the client. All the resources in the request are identified and handled by its specification. Every message contains all the information how to deal with the request.

The XML-RPC interface we used for our application is an implementation of this architecture, and naturally it follows the REST constraints.

3.2.6 Our Implementation

We used all the above mentioned technologies and paradigms to create a system which schema is presented in picture 3.7. Let us briefly introduce the current solution of our business partner and our addition. The current solution follows the industry standard and implements variant of Model-View-Controller schema, and so do we.

In our particular case our module will use the current frontend as a view, the searcher that interacts directly with the database mimics the model part, while the Ranker calling to an external library and can be roughly considered as a controller. We decoupled the ranker into separate parts from several reasons. The frontend is already implemented, thanks to the REST API of our model it will only be necessary to change a few commands for communication.

Our main effort was to implement the searching and feature extraction part. We merged these together because they share a lot of code, especially regarding the feature extraction mechanisms which interacts a lot with the database.

We than wrapped the *RankLib* library to be able to cooperate with it directly. This library was intended as standalone executable, and therefore, it heavily relied on text inputs and output. We focused on the ranking and parsing methods to be able to communicate with them directly. This part is decoupled also because of computational resources which it takes. Such situations may occur (such as high load of servers) where we would like to disable the ranked and rather serve clients in due time than present the best possible order.

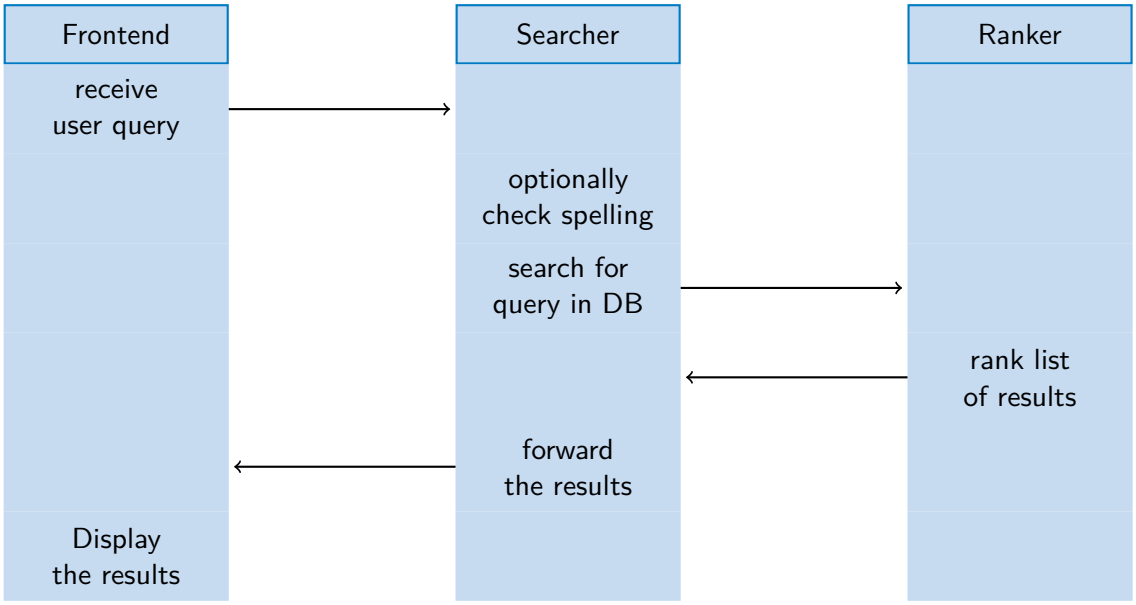

Figure 3.7: Infrastructure

3.2.7 RankLib

As a ranking module in our work we used the *RankLib* library[3]. This library implements several states of the art algorithms for learning to rank. The library works with several standards that established in the LTR community such as a data format for the ranking data or metric evaluating the rankers.

We feel necessity of changes because the *RankLib* is intended as a standalone executable. As the library is provided as an open source under BSD licence, we took the code and made a few structural rather then functional changes.

The main change crucial for our purposes was that we allowed by simple wrapper that the library methods can be invoked directly within the server to achieve the best response available rather then using the text file that needs parsing. Therefore, we work in our work with data structures that the RankLib uses, namely `RankList` and `DataPoint` to overcome the overhead that would arise from storing the data into the text files and loading them afterwards in the library. This design might be suitable for the command line application but it is rather impractical for the server side application. But this functionality was kept because our server must able to server in two different situations. It must be able to respond to query as quickly as possible, and on the other hand, it must be able to process quite large user data which we do not like to store even temporarily for longer period then absolutely necessary in RAM.

[3]The library can be found on this page https://sourceforge.net/p/lemur/wiki/RankLib/

The minor change we made was the output, which now uses standard logging tool `log4j` that allows us to control it with its configuration files and record the history of the results or errors that might occur.

3.3 Set of basic features

Before we start describing the new searching and ranking system we will briefly explain the current solution. The results for a query using the current system are presented in the figure 3.8. As we can see the result page is separated into three sections according to the type of result.

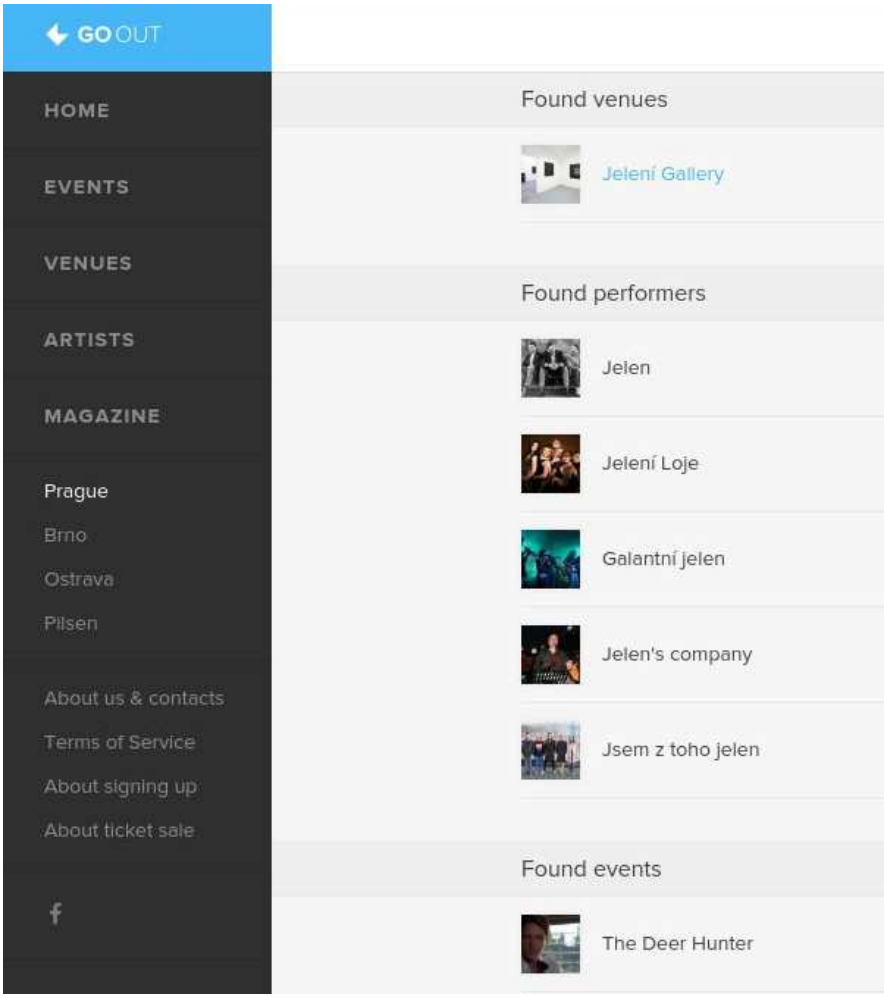

Figure 3.8: Current search results shown by server GoOut.cz

The previous version of searching was realised by *tokens*. Tokens were simply keywords which were stored in separate table and was shared among all the data types. When query was

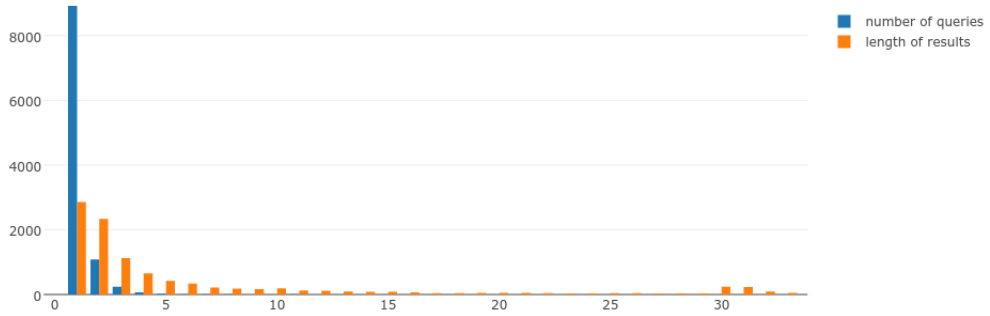

Figure 3.9: Histogram of the queries frequency and of the length of the results

performed the relevant tokens were found, this result was joined with the tables containing one type and the results were obtained. The join was performed three times as we are dealing with three types of documents. On top of the database result there was implemented a naive ranking algorithm. It worked as follows. If an exact match was found, then the result was put in the first place. Regardless the exact match, the other items were ordered by the length of their names. The ones with shortest names were listed in the first place. Such ordering is visible in the figure 3.8. Notice that this ordering was applied only within one document type.

We tried to reimplement the searching model. We wanted in the first place to avoid the costly `JOIN` operation, which interestingly did not perform that bad in reality supposedly thanks to *MySQL* caching. The results will be in future presented in one list only. The following part will describe the new search engine and the features we selected for the LTR.

3.3.1 Statistic of the data

We were working with data from Aug 18, 2014 to Apr 24, 2015. This dataset contained 65,982 search events, only 48,161 of them were parsed as a valid log and we encountered 18,126 distinct queries. As we expected, the distribution of the queries follows the power law. This can be seen in picture 3.9.

The average result length was 6.73 results. From the histogram in the picture 3.9 we can see that there is a lot of queries that were answered with only one result. The number of results then decreases, we can see a little peak around the number 30, this was probably caused by some general queries such as ‘theatre’ or ‘concert’.

3.3.2 Searching

As mentioned in chapter 1 there are several standard methods that are used in the IR.

We will describe our approach. The new search method is performed as a regular expression matched against the database full text columns, i.e. the query uses the `MATCH - AGAINST` construct in scope of full-text search.

```
WHERE MATCH (example_column) AGAINST ('some text' IN BOOLEAN MODE);
```

This `WHERE` clause returns all rows the columns of which stated in the parenthesis after `MATCH` keyword contains text specified in the parenthesis after `AGAINST`. The text being searched can be controlled with multiple operators; we use ones that MySQL documentation calls boolean. In our case we use two of them. Operator `+` indicates that the word must be in the result. We also use the `*` operator which represents truncation, i.e. the result contains the partial matches against the last part of the query. The resulting query enhanced by these operators is shown below.

```
WHERE MATCH (name, keywords) AGAINST ('+club +vago*' IN BOOLEAN MODE);
```

After this step the result set is obtained and can be ranked. For such ranking task we need features that can be used as an input for the learning algorithms. As we stated in the chapter 1 it is insufficient to have an incidence matrix which would contain the incidence matrix of all document and all world. We will provide a description of our implementation.

We also encouraged ourselves to implement a simple spell checking module. It is heavily inspired by Peter Norvig's blog post[4] and an implementation in Java[5] done by Dominik Schulz. The Schulz's implementation was slightly modified to fit our needs.

Building inverse document index

The inverse document index consists of several parts. The first part is dictionary of words; these words do not have to be all words from the data. How to select some relevant subset is not easy task and the approach how to solve it varies. In image retrieval the k-means algorithm is used to choose the number of such a dictionary, while working with text we can apply multiple steps in order to reduce the data and obtain only the relevant information.

The long texts are tokenized into separate tokens called *terms*, in text document this block is usually a word. The basic idea is to split the documents on white spaces and punctuation characters. Although this is a good idea in general, in some particular applications it can cause trouble. In such situation we have to take care about terms that can occur in our

[4] `http://norvig.com/spell-correct.html`

[5] `http://developer.gauner.org/jspellcorrect/`

domain and would produce unsatisfiable results, i.e. searching for flights between towns which names consist of multiple terms.

Next step deals with stop words. Stop words are such words that bring almost zero information. It is a reasonable assumption to drop these words. These words are often selected based on the statistic of the data or picked manually by someone with the domain knowledge. In general English text such words are verbs *to be*, *to have* and various prepositions such as *by*, *for*. In our context it would be *band* or *performer*.

Normalization is another process that should take place while creating the dictionary. We have to strip the text of all accents and diacritics. This holds for English but it is even more important for languages like French or Czech. The following step should be case folding, which means that we simple change all the characters into lowercase. This can sometimes produce ambiguity between normal terms and names that are derived from such terms, but nevertheless, the lowercasing is the most common approach.

The final transformations are stemming and lemmatization. These two processes try to conclude the base form of the terms, i.e. words are created from each other by prefixes and sufixes. Stemming is roughly similar to omitting the last part of the word with hope to derive a base form. Lemmatization uses the correct grammar model to achieve the same results. Describing both processes in further detail is out of scope of our work. More details can be found in [10].

In our application we depend on data provided by our partner. From this data we created our vocabulary directly. They used them successfully for a long period for searching by tokens. These will be substituted by keywords in the future. But in the end, we can consider the set of these terms as a human produced dictionary of all keywords and it would be hard to create a better vocabulary on our own, not mentioning that we would have to perform such task for two languages, Czech and English.

3.3.3 Domain independent features

TF-IDF

The holy grail of nowadays IR systems is created by two measures. The first is called ‘term frequency’ and the second one ‘inverse document frequency’. Indeed this metric proved itself very useful and can be found in many applications in originally not intended areas. One example for all would be image databases where the documents are described with ‘image words’. The whole framework of TF-IDF was used for searching the most relevant images. Such applications show the efficiency and robustness of this feature.

This feature or measure, depends on the point of view, but in our context we use it as a feature so it will be referred as such from now on; is used almost exclusively as a whole although it in fact puts two metrics together.

The first part is called *term frequency*, the name is quite self-explanatory - this metric provides information how often the term being searched occurred in document. For knowing whether the term is a random word or the main topic of the document, this information is not enough. For example we can search for term *the* which occurs many times in most of the documents but this is not very helpful for ranking the documents.

The need of distinguishing the relevant term from the irrelevant one introduced the Inverse document frequency (IDF) metric. We measure it as

$$\mathrm{IDF}(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|} ,$$

where N is the count of all the document; this N is divided by number of document in our database that contains term being searched. This fraction is then logarithmically scaled to prevent the rare keywords from reaching insane numbers. Sometimes we also would like to prevent the edge cases to occur when the other documents do not contain such term and when we would divide by zero by adding 1 to the denominator.

The resulting metric is computed as a product of these two previously described metrics, with all the symbols with the same meaning as before:

$$\text{TF-IDF}(t, d, D) = \mathrm{TF}(t, d) \cdot \mathrm{IDF}(t, D)$$

We consider understanding of this metric essential, and therefore, we feel the necessity to provide a small example. Let us consider data in table 3.1 as an excerpt from a larger database. There are two tables: the first one provides the frequencies of individual terms in documents and inverse document frequencies. The second table shows two examples of the computation of TF-IDF.

In our application we took two main sources as a vocabulary. For maintaining the backward compatibility we used the old tokens, and the set of all names and keywords. When a query is being processed, we assign to each document the word from dictionary that shares the longest common subsequence with the query and belongs to the particular document. This is done for every document in the result set and then the TF-IDF is computed.

BMI 25

This ranking function, commonly referred as BMI 25 Okapi, was in history often used as the only input for ranking the documents. Instead, we used its output as a feature for ranking algorithms. This function is defined as

$$\mathrm{score}(D, Q) = \sum_{i=1}^{n} \mathrm{IDF}(q_i) \cdot \frac{TF(q_i, D) \cdot (k_1 + 1)}{TF(q_i, D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\mathrm{avgdl}}\right)}$$

term	documents	IDF	frequencies	1	2	3	4	5
the	1, 2, 3, 4, 5	$\log \frac{5}{5} = 0$		10	8	11	1	6
oboe	1, 4	$\log \frac{2}{5} = 0.91$		10	0	0	2	0
cello	3, 4, 5	$\log \frac{3}{5} = 0.51$		0	0	8	2	7
xylofon	2	$\log \frac{1}{5} = 1.61$		0	1	0	0	0

query	cello	query	oboe
document 3	$8 \cdot 0.51 = 4.09$	document 1	$10 \cdot 0.91 = 9.16$
document 4	$2 \cdot 0.51 = 1.02$	document 4	$2 \cdot 0.91 = 1.83$
document 5	$7 \cdot 0.51 = 3.58$		

Table 3.1: TF-IDF example

As a feature we took the part suggested by [5]. Let us explain the notation: the k and b are meta parameters usually as well as in our implementation set to $k = 1.5$ and $b = 0.75$. The $TF(q_i, D)$ is a frequency of term q_i in the document D, $|D|$ is a length of a document in terms of words and $avgdl$ is an average document length among the whole database of documents.

This metric has an interpretation from information theory point of view. It basically measures the information content of statement 'document D contains query Q'; since we work with an assumption that the terms occur in documents independently, we can sum the logarithms of probabilities.

Cosine similarity

This feature was inspired by the paper [5]. Generally the cosine similarity is used to measure the distance between two vectors in a vector space; it was slightly modified to be able to measure the similarity of one result within the whole result set of a query.

Let DV_i represents the TF-IDF vector for document i. D_{Q_k} refers to all the documents for query Q_k, $|D_{Q_K}|$ stands for the size of the result set. Then cosine similarity is defined as:

$$\text{Cosine similarity}(D_i, Q_k) = \frac{1}{|D_{Q_k}| - 1} \sum D_j \in D_{Q_k}, j \neq \frac{DV_i \cdot DV_j}{|DV_i| \cdot |DV_j|}$$

As we can see, the last part of formula is a classical cosine similarity well known from geometry and other applications. The histogram can be seen in the picture B.1. As we can see, the documents tend to be similar within the query. Moreover, we can clearly see some peaks that belong to the special values of cosine, namely 0 and 0.5.

Query length

We measured the length of a query and used it as another feature. For example there it is nicely visible from the histogram of the queries that a minimal query is not cropped to some

reasonable length such as 3, and that there is a peak at length 5 and then the length of the query decreases rapidly. These trends can be observed in picture 3.10.

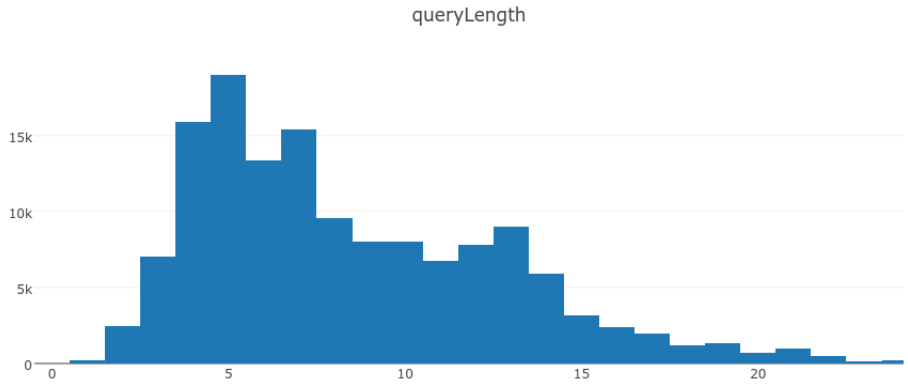

Figure 3.10: Histogram of feature length of a query

Text length

We assume that every document or any other object in particular domain can be measured in a suitable way. We can create a hypothesis that longer documents provide more information than the smaller ones. And therefore, we might prefer them in the results. This feature should capture this relation if it holds. In our case we take a long text description belonging to the `event`, `performer` or `query`, tokenize it and count the elements. The histogram of this feature can be seen in figure 3.12. As we can see there is a significant peak around zero, which signify that there is a lot of documents that have either no or a short description. There is a peak around 300 hundreds words of the description and then the length almost monotonically decreases.

First occurrence

While computing the TF-IDF we can extract another feature. We called it a first occurrence. We split the document description on the first occurrence of the *term*. As we know, the size of the description may vary and to preserve comparability between documents we count the size of the portion before the first occurrence and divide it by the length of the whole text. As a fraction, such number will be directly normalized.

In the figure 3.12 we can see that there are two significant peaks: one lies at 0 and second at 1 respectively. The peak around 0 means that there are lots of documents that start directly with the query word. The peak at 1 is more interesting - it signifies that there is a lot of documents in which the key word does not occur at all. This might look strange but it

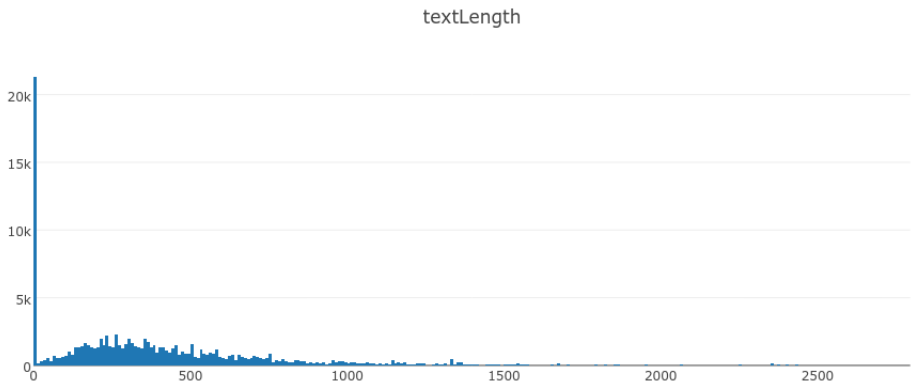

Figure 3.11: Histogram of the length of the description text

is a result of manually assigned keywords that does not have to follow in strictest sense the description.

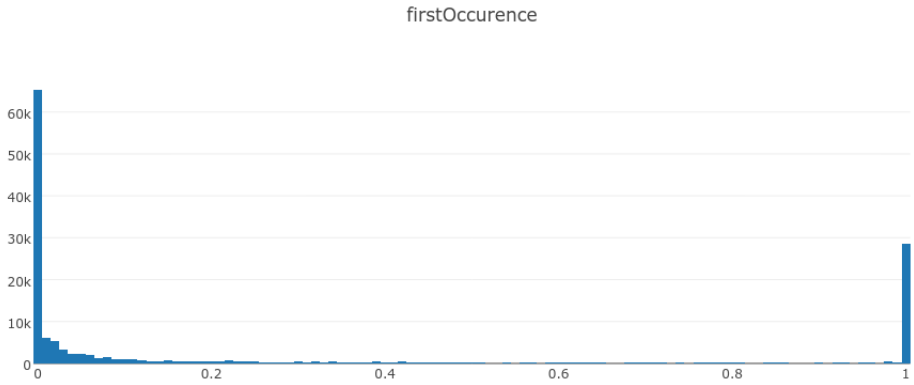

Figure 3.12: Histogram of the first occurrence distribution

3.3.4 Domain dependent features

Type feature

This feature is straightforward. As mentioned before, in our domain we are work with three types of data, namely `performer`, `venue` end `event`. Every item in the resulting list must belong to one and one only category. We create binary features from such relation as if the item is `event` then the feature `event` is set to 1 and the others to 0. This works analogically

for the other two types. We chose such a binary approach rather then enumeration because it is known that such algorithms can benefit from this configuration.

Video views

Many of the documents in the database refer to some type of multimedia content. Most of the performers have video or music video on video sharing server *YouTube*. The same holds for the events: the organizer usually wants to promote the event by linking it to the performer's previous events. There are two features extracted from the data, one is an aggregation over all videos – how many times they were played and the second one is sum of all videos played till the end. We were hoping that this measure can provide information about how much popular some videos and consequently the documents are, and whether the users were just curious or watched the whole videos.

Views

Our business partner stores the overall hits of current page displaying information about venue, performer or event we are interested in. This simple measure of popularity among the spectators of the web site should be certainly used as a feature for ranking.

Inner Events

This feature computes the events that belong to a certain venue or a certain event. For venues it is straightforward: we simply count all the events that took place in that venue. For events this values represent how many events belong to their 'parent' event. This might look useless but after discussion with the business partner, we realized that there are often events such as festivals or workshops that consist of several separate events and we might like to emphasise the parent event.

On the other hand, we decided to ignore this feature for performers mainly because we do not have all the data. The performer for example can perform a lot of events abroad but take part in only one Czech event, therefore, the data would be highly biased towards the Czech performers who are performing continuously at venues where our business partner keeps data about their appearance. Moreover, such query for performers would necessary involve joining tables in the database which would cost more time.

Freshness

Although even during the searching process we eliminate events that occurred in the past, we would like to preserve some notion of time. We use the time when the query was performed and compare it with the value when the corresponding item in the result list was edited for the last time. This gives us something what we call freshness, i.e. it could manage to capture 'trending' results.

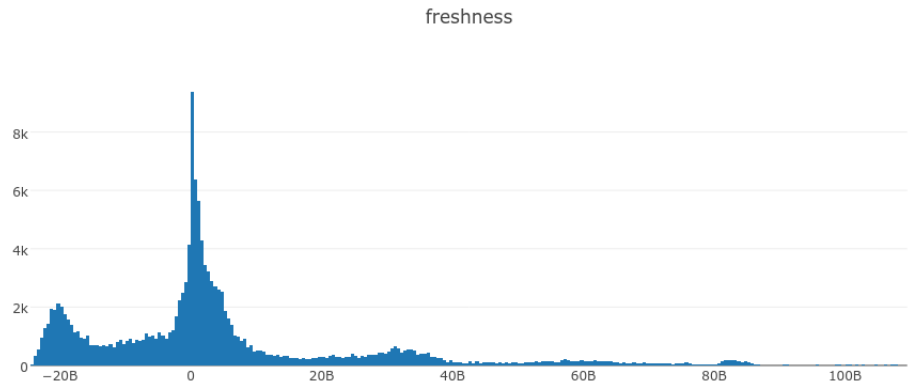

Figure 3.13: Histogram of freshness feature

The histogram of the data can be found in the picture 3.13. It is recognizable from the picture that there is a quite special shape of the distribution which suggests that users tend to search for queries that are either fresh or older. This corresponds with our intuition that people are looking for events that take place in near future or searching for venues or performers.

Although we were able to find the explanation, we have to think for a while. Most certainly in the time of the query the database entry cannot be younger. This is caused in our data by the situation that the database dump is younger than some queries. As this holds for the training, test and validation set it did not cause any trouble, but for the real life ranking we have to solve this. We tried different setups for our data to overcome this issue. The data was left intact, was cropped by zero and the absolute values were computed. The results of the experiment will be presented in the next chapter.

Email

We concluded that there might be a difference between queries performed by registered users and random visitors. In order to capture it we added a binary feature that indicates whether the user was logged in or not. In case that known user performed a query, then the 1 is assigned, and 0 otherwise.

Base feature set summary

As the reader may have noticed, every feature mentioned so far belongs to its separate distribution and can range in different intervals. This points to the need of normalization. The z-score normalization was done by the RankLib library and as result the algorithms performed much better with the normalized features.

3.3.5 Feature interaction analysis

We performed a simple statistical analysis of the linear correlation among features. The result is displayed in table B.1. As we can see there is some dependency between the performer and venue features. This is because whenever event is 1, then performer is 0 and vice versa.

The other features seem to be fairly independent, and thus we consider it as a reasonable assumption to work with all of them in the following work.

CHAPTER 4

Enhanced feature sets

This chapter will present our effort to enhance the basic set of domain dependent and domain independent features by some relational data. We have tried two different data sources. At first we used data obtained form Google's Word2Vec library, and consequently we worked with data which were extracted from the DBpedia project.

4.1 Word2vec

In recent years the increase of popularity of neural networks was noticed. There were enormous achievements with so called Deep Networks, for example beating the world best Go player and image recognition. Word2vec is in contrary a shallow two layer neural network developed at Google team led by Tomas Mikolov [11]. For sake of brevity we will address Word2Vec as w2v from now on.

4.1.1 Continuous Bag-of-Words (CBOW) and Skip-gram

In w2v there are two theoretical concepts being used. They are called continuous Bag-of-Words and Skip-gram. We will provide brief a introduction. Continuous Bag-of-Words share similarities with the neural net language model although CBOW approach was developed to overcome the computational complexity arising from the number of hidden layers in the neural net.

The continuous Bag-of-Words is learned to predict values describing a word given its context, i.e. words occurring before and after the word we are interested in. On the other hand, the continuous Skip-gram model is able to predict words in a certain distance from the given word. We refer more interested readers to [11] or other works by Tomas Mikolov.

This output of a layer, before the concrete words are looked up in sophisticated dictionary served us as an input for the ranking algorithms. We decided to limit the size of the output

to 100 features. We experimented with higher values but in the end, we concluded that these models were more biased to overfitting and their computational complexity as well as the memory requirements during learning increased dramatically.

4.1.2 Our implementation

From our data we extracted all the text descriptions belonging to the events, venues and performers. This text file has about 140 MB and in terms of size it is comparable with the dataset supplied by the w2v. The model was trained with the same parameters that were used in the demonstration example script also attached to original w2v implementation. Then we used a short piece of code from a web page[1] that allowed us to convert the binary data stored by the C implementation of w2v into a plain text `csv` file.

This rather large, over 100 MB, `csv` file is loaded into a HashMap in our Java implementation; this situation is not ideal but we had to overcome the technical difficulties that emerged while trying other and more straightforward implementations.

Initially we tried to use Java implementation of w2v provided by deeplearning4j[2] package. Although we were able to compile the application with the all necessary packages, the work with it was infeasible. We were trying to learn the network with our text database in text form, but the deeplearning4j w2v implementation took more then 12 GB of RAM and was killed by the operation system. We could consider this as a limitation of the developing machine, but on the other hand, our business partner was also not willing to spent so much resources on just one module. Therefore, we used the original implementation provided by Google and modified it to satisfy our needs.

The remaining step was to assign the values obtained from w2v to our data. As we know every document in our database is described by several names and keywords. For every description keyword we queried the dictionary learned by w2v and retrieved either the vector with specific values or filled with zeros depending on whether the keyword was or was not in the dictionary. For every document we stored all the values returned from w2v and summed them to one vector in the end. We decided to use this because it has been shown that w2v is able to preserve the relation between words in the vector space.

4.2 DBpedia

The other enhancement we decided to use is based on project called DBpedia[3], which describes itself as '*a community effort to extract structured information from Wikipedia and to make this information available on the Web*'.

[1] `http://stackoverflow.com/questions/27324292/convert-word2vec-bin-file-to-text`

[2] `http://deeplearning4j.org/`

[3] `http://wiki.dbpedia.org/`

The DBpedia provides all the information that is stored in the side boxes on the whole wikipedia. Such amount of data must be structured in some way. DBpedia provides colourful pallet of data formats that can be used. The most descriptive is an ontology. Such ontology can be queried by f.x. SPARQL language. For our purpose we decided not to use SPARQL or any other query language. The SPARQL language is very expressive, and therefore not easy to process. Moreover, for work with it we would need to embed some ontology reasoner, for example Pellet, which would further increase the complexity of our programme and more importantly it would increase the computational time for a response to the query.

We used another type of data, more specifically the relation called in terms of DBpedia *instance type* dataset for both languages we are interested in, i.e. in English and Czech. We downloaded the `nt` files, which contain the triples. An example of two lines is listed bellow. We formatted the text to fit the width of the page so indent lines are just continuations of previous lines.

```
<http://dbpedia.org/resource/Autism>
        <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
        <http://dbpedia.org/ontology/Disease> .
<http://dbpedia.org/resource/Martin_Scorsese>
        <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
        <http://dbpedia.org/ontology/Person> .
<http://dbpedia.org/resource/Anarchism>
        <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
        <http://www.w3.org/2002/07/owl#Thing> .
```

Figure 4.1: Three DBpedia triples

From this data we have to parse only the information we are interested in and do some basic normalization. From every term between `<` and `>` we take the part after the last slash. While parsing all these words are lower cased. And whenever the `_` (underscore) character (which in the dataset stands for space) occurred we split such term and worked with the original term as well as with all its parts. We also discovered that the dataset contains some types that looks like `Q1546`, i.e. letter `Q` followed by some number. Because we want to have our additional features interpretable we simply left these entries out.

Since every term can belong to multiple types we created a HashMap where we used terms as a key and the set of their types we considered as a value.

Let us provide some statistic of the data contained in *types instance* dataset for English and Czech. As we can see in the table 4.1 the number of terms decreased compared to the original files. This is an understandable result of normalization and leaving out the data we are not able to interpret.

feature name	value
number of terms in English file	5647975
number of terms in Czech	64039
number of terms in HashMap	4965816
number of distinct terms	1243873

Table 4.1: Basic information about the DBpedia database

4.2.1 Using the DBpedia data

There are over 650 distinct types in the database. When the human uninterpretable were filtered out we still have 480 types left.

We performed a simple experiment with text dump of our TF-IDF vocabulary to obtain the coverage, i.e. for how big portion of our data we are able to find the type. We ended up with a number around 30 %. Once again in order to keep computational cost under control we selected only a subset of most common types among the data. We also considered stop words and omitted the most general type *Thing* because it will not bring any information.

type	frequency
person	10758
album	10446
film	9860
band	7968
musical artist	7188

Table 4.2: Frequency of DBpedia types

As we can see in the table 4.2 there are several types in the top five most common types that look very promising ranking. The table shows that there are a lot of types that refer to the artistic area. There are even more of such 'suitable' types when we list for example the first one hundred most frequent types. We decided to take only a subset of the types limited by their support for creating a features for our documents.

We create a binary relation for each document. This relation work as follows. We initialized a vector of length of selected types, then if we were able to find in a look up dictionary some type for a corresponding keyword, we added one to the particular column belonging to this specific type. We iterated over all the keywords belonging to one particular document and summed up the values for the matches into one vector.

4.2.2 Enhanced features summary

We enriched our feature set by quite number of very different features. We added 100 features that are and output of the w2v neural network and are quite impossible to interpret. On

document	keyword	person	album	band	musical artist	...
Iggy Pop	iggy	1	0	0	1	
	pop	0	1	0	0	
	punk	0	1	0	0	
DBpedia vector		1	2	0	1	...

Table 4.3: Example of DBpedia features

the other hand, we have very easily understandable data that simply categorize out data into roughly 100 categories or more precisely expressing the relation of belonging to multiple categories.

We thought that our assumptions are reasonable that to add them to these types of features will be a possibility to improve the results. The w2v proved itself very useful in many applications where capturing of relations between words was required. Therefore, we hoped that these vectors will play a similar role used in our case as input for the ranking algorithms. The second idea was to use the data created by internet users. These data are usually considered valuable. They can provide information about what is popular among the internet users, what the users are interested in. We assume that if they are interested that Wikipedia pages are created, and therefore captured by DBpedia project.

To provide a theoretical justification is certainly out of scope of this work. Even considering just w2v, there were several attempts to provide a rigorous explanation why the algorithm works. So far the effort was not fulfilled with the definite answer. For the DBpedia data we are not aware of any similar attempts nor justifications. All the assumptions about usefulness of extended features and about their influence on overfitting will be considered after conducting experiments. The results of the experiments will be described in the next chapter.

CHAPTER 5

Experimental results

In previous chapters, a description of LTR algorithms was given. We introduced and discussed features used for ranking. Afterwards, new features were used to enhance the feature set. This chapter is devoted to an experimental evaluation of algorithms and a comparison of the performance of the feature sets.

5.1 Input data

Our input data is not ideal, at certain points we had to make a decisions how to create data, how to solve missing values et cetera. For having our assumption justified, we conducted experiments in order to refute or confirm them. The conclusion will be provided in this chapter and some figures with the results will be presented in appendix B.

5.1.1 Relevance grades modelling

Our baseline relevance grades data form a very sparse relevance grades distribution. We consider the data from the users as a ground truth. All the documents that were clicked for a given query were assigned the maximum relevance grade, and the other results were assigned to zeros.

We were not sure what will happen with the results when we tried to modify our model for predicting the relevance; therefore, we set the relevance grades randomly in such a way that each document was assigned with a random relevance grade, strictly smaller than the maximal relevance; the documents which were clicked were assigned the maximal relevance grade as before. We observed that the results were almost the same; therefore, we decided to use the straightforward approach which considers non-clicks irrelevant.

5.1.2 Feature ambiguity

Earlier, we promised results regarding the freshness feature. The table B.2 for the MART algorithm shows that with absolute values the algorithm was able to achieve a slightly better score on a training set, but it generalized worse and was more likely to overfit. Therefore, we decided to adopted the scheme where all negative values are limited by 0.

5.2 Machine learning and RankLib

Basic machine learning concept is to strictly split the data into 3 folds. These folds are commonly named training set, testing set and validation set respectively. The naming convention regarding the testing and validation set may vary depending on the author of the publication or algorithm. The main distinction between these two sets is that one is used for the model evaluation during learning, and therefore, some information from it may leak into the learning process and we can overfit our model and the second one is never seen during the learning process.

We will follow the naming scheme used by the RankLib library. The training set is straightforward, the split used for evaluation during learning RankLib is called validation and the last split, never seen by the algorithm before, assessing the performance of fully learned model, is the test set. This configuration is shown in schema 5.1.

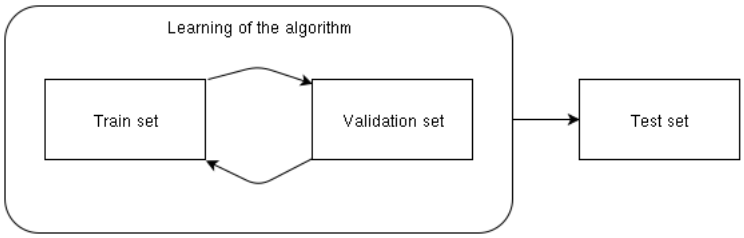

Figure 5.1: Machine learning schema

5.3 Algorithm comparison

We decided not to use all algorithms that RankLib provides. We abandoned the Coordinate Ascent algorithm. Its learning times were in order of magnitude slower than for the other algorithms, and its results were certainly not the best ones.

In the first place we are aiming to choose the most appropriate model for our data which in practice means we have to choose the right algorithm. We use the classical 10 fold cross-validation scheme provided by the RankLib library.

In accordance with the standards in machine learning as described earlier, the data are divided into three sets. We will measure all results in terms of ERR metric as defined in 2.1.2.

We use the standard cut off parameter which limits the number of documents. We set the cut off parameter so that we compute ERR metric on at most 20 documents; this is denoted as `ERR@20`.

Let us report on achieved results. The first figure 5.2 shows the performance of the algorithms in terms of `ERR@20` metric. The dashed line shows the current baseline algorithm preferring exact matches and shortest names. As we can see, all the algorithms were able to beat the baseline solution on training set while some of them struggle to achieve this result also on test data. Among the best ones there were ListNet, MART and Random Forest algorithms.

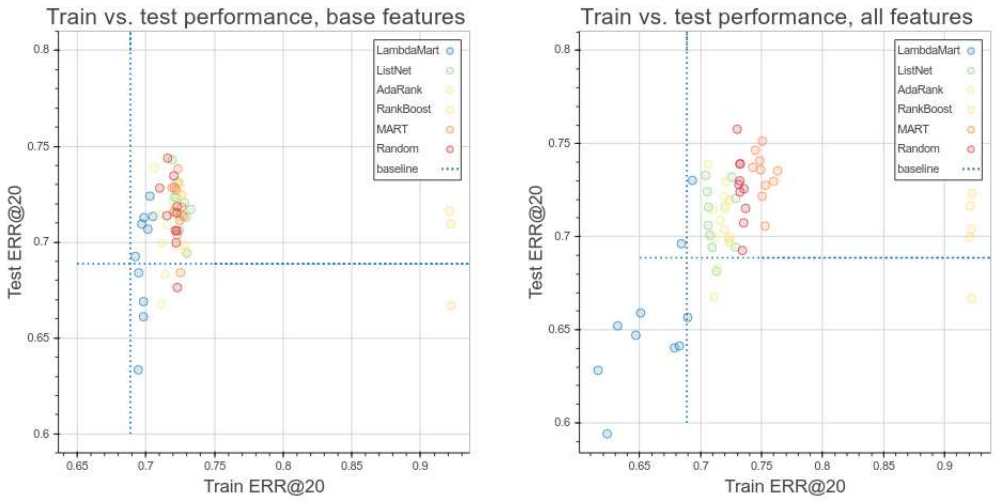

Figure 5.2: Score of algorithms with base and all features

We also wanted to examine the learning times of the algorithms. The learning times are shown in the figure 5.3. As we can see, certain algorithms run for quite a long time but their results are not convincing - such as LambdaMART. Random Forest and RankBoost shares longer running times but they are able to beat the baseline search by a significant margin and might prove itself usable in the future.

We also performed the same experiment with all features (under assumption that the other features will help, which will be examined later). As we can see, the performance of the algorithms increased, as well as the learning times. The comparison of the figure 5.3 with the figure B.2 tells us that for the time intensive algorithms the learning time grew linearly with the number of features. We can see roughly a 10-fold increase which is similar to roughly 12 times bigger feature vector.

In terms of ERR performance there are no dramatic changes; the algorithms that performed well even with the basic set of features performed with more features also well. The only thing that changed was ordering.

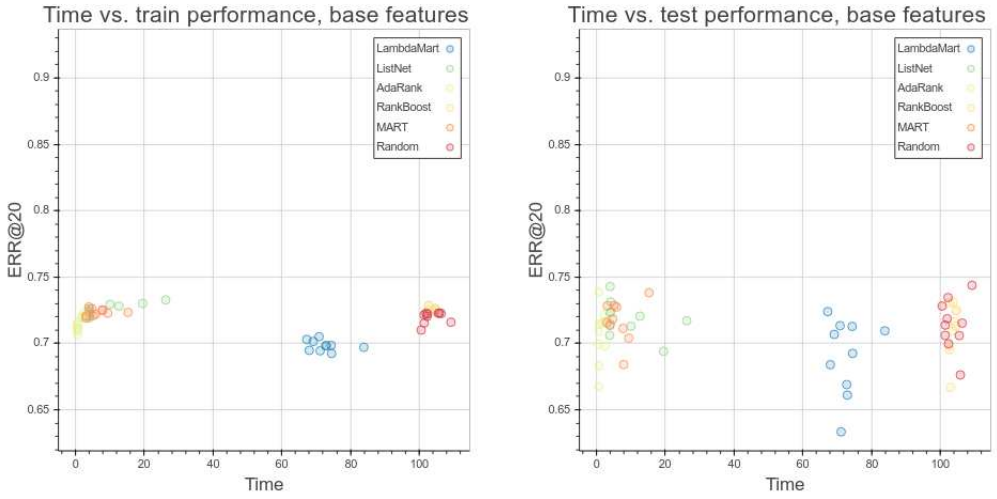

Figure 5.3: Learning time for algorithms with base features

As we can see in the figure B.2 the algorithms such as AdaRank and LambdaMART are not performing very well; moreover, LambdaMART is very time consuming to learn as well as RankBoost. Because of this unfortunate combination of bad performance and time complexity we will focus from now on on the Random Forests and MART algorithms.

5.4 Features analysis

While comparing the feature set, four interesting events occurred. This section will examine these events in detail.

The first observation is, that for certain algorithm the enhanced features were not able to improve the results. In the figure 5.4 we can clearly see that the algorithm was not able to benefit from these data even on training set. Therefore, is not surprising that the results of the test set are almost the same.

For the ListNet algorithm the results with more features are sometimes even worse then with the base set. It was not because of the overfitting as we can see in figure B.3. This event is caused by the inability of the algorithm to explore a bigger state space and find better solution.

On the other hand, while considering the RankBoost algorithm an expected event occurred. As depicted in figure B.4, the performance with enhanced feature was better while training but the lack of improvement on the test data signifies that the algorithm overfit the training data.

We were most interested in cases where the algorithms were helped by enhanced features. Such algorithms that were able to benefit from larger feature vectors are based both based on the regression trees. This is expectable behaviour since this type of algorithm scales well

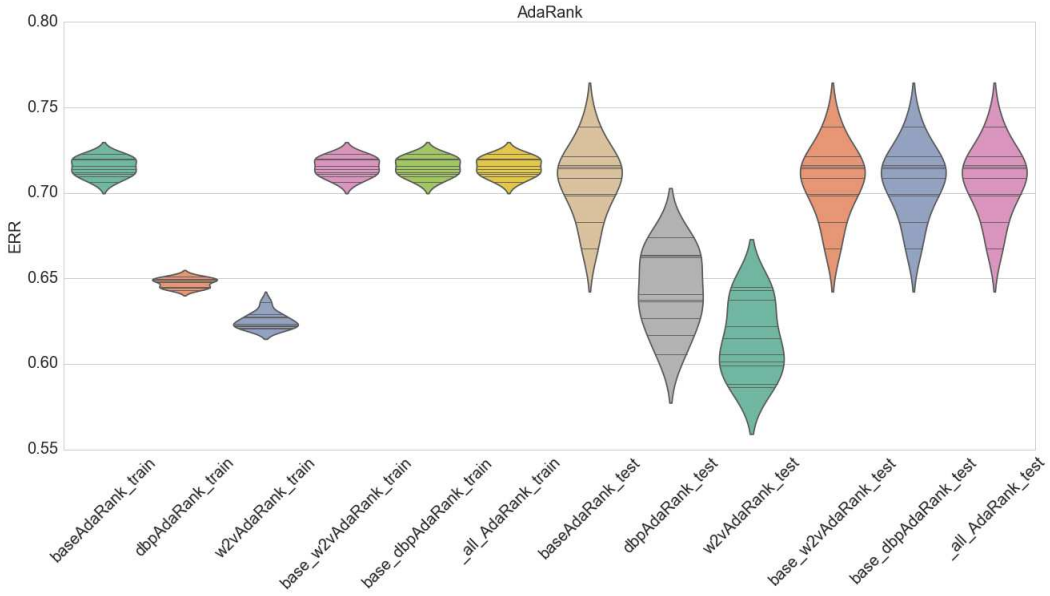

Figure 5.4: Feature sets for AdaRank

with the number of features and they implicitly select the most relevant features for the splits of the feature space.

In the figure B.5 we can see that both w2v and DBpedia features alone caused increase in performance, and combined these enrichments together, the result was even better. We can see from the figure B.5 that this was the most successful option for the Random Forest algorithm.

Similar behaviour was observed for the MART algorithm. Its behaviour is presented in the figure 5.5. When we take into account the graph showing learning times B.2 we consider as a most usable the MART algorithm. It performed among the best algorithms compared to the others. We can also see a notable improvement with more features, though it certainly shares the risk overfitting as well as the other algorithms. In favour of this algorithm speaks also its learning times which were among the quickest, therefore we consider it as a best candidate. 5.5.

5.5 Query response time

The last measurement we were interested in is query response time. We can consider our system as usable if it is able to respond within hundreds of miliseconds, otherwise the results are useless since the users are not willing to wait too long even for better results.

As we can see in the figure 5.6, the response time is independent on the model and even on the feature set. From the second figure we can conclude that the only parameter that

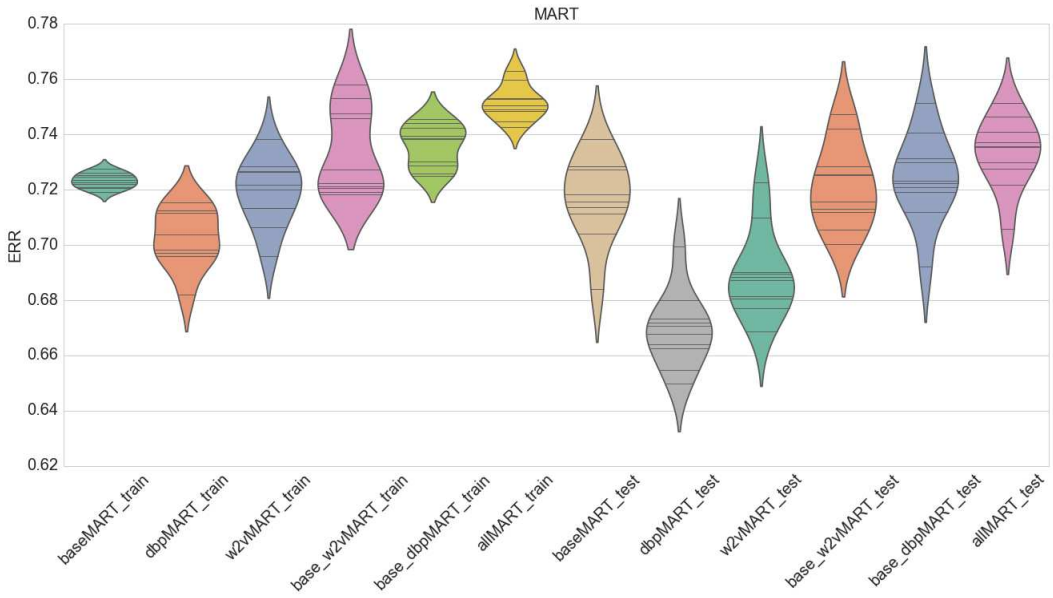

Figure 5.5: Feature sets for MART

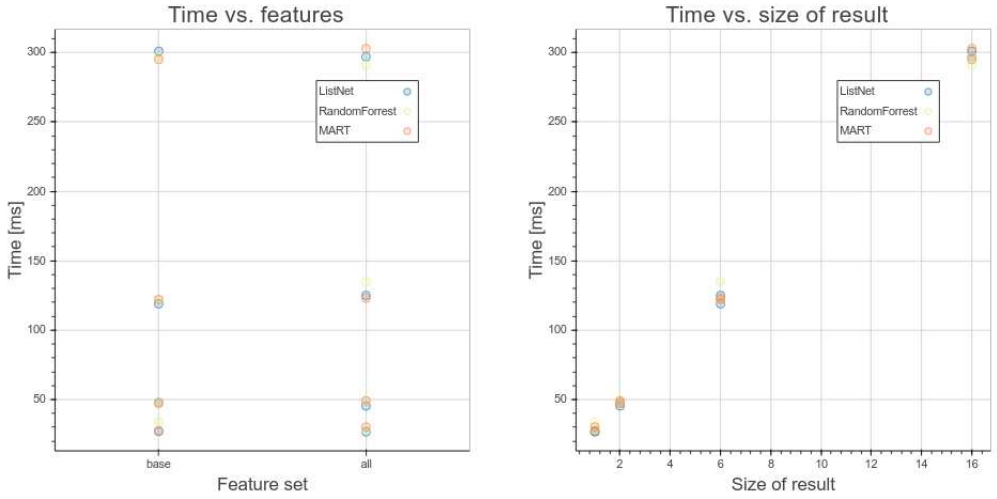

Figure 5.6: Query response time

influences the response time is the number of documents retrieved as a result for a given query. Therefore, it would be reasonable to limit the number of results to avoid too long response times.

CHAPTER 6

Conclusion and Future work

In our work we presented the IR problem with its development in the recent past. Then, the theoretical background of the state of the art algorithms was provided as well as the description of several metrics commonly used to measure the performance of LTR algorithms.

Then, we introduced the libraries and tools used in our work. Next, the implementation details of our work were provided. In chapter 4, we discussed several feature sets used for ranking. Afterwards, the algorithms and different feature sets were evaluated. The comparison showed that there are algorithms more suitable for our task. These algorithms were able to profit from the higher number of features and produced a better ranking. Eventually, we observed that the query response time depends merely on the number of results and we concluded that the query response time of our system is suitable for practical needs.

In future work, we would like to come up with a more thorough feature analysis and then to perform a feature selection. This could help to improve the performance of the algorithms that were not able to handle the enhanced features. Of course, it would be very interesting to observe the performance of our implementation in the real application and improve the model from its previous results.

It also might be interesting to implement LTR task using the Hibernate Search that internally rely on Lucene – a well known and efficient indexing and searching tool. It would be interesting to see how this out-of-box solution would perform in comparison with our model.

APPENDIX A

Contents of DVD

A.1 Directory structure

`/2016-Tomas_Trnka-Diploma_Thesis.pdf`	Electronic version of the thesis
`/tex`	LaTeX sources of the thesis
`/GoOutSearcher`	Sources of the application
`/RankLib`	Sources of the edited *RankLib* library
`/Example`	Folder containing example setup

A.2 GoOutSearcher

We have implemented a searching and learning to rank server module that uses the ideas presented in the thesis. An IntelliJ IDEA project, that can be also compiled with `maven`, is contained in the directory `/GoOutSearcher`.

A.2.1 Libraries and used software

- RankLib (originally implemented by Van B. Dang)
  Java library providing learning to rank algorithms mentioned in this thesis, the edited version of this library is attached as a separate project. For original see: `https://sourceforge.net/p/lemur/wiki/RankLib/`, for our fork see: `https://github.com/trnkatomas/RankLib`

- Spring Framework
  The Spring Framework is an application framework and inversion of control container

for the Java platform. The dependency is stated in the maven dependency tree.
See: `https://projects.spring.io/spring-framework/`

- Hibernate
  Domain model persistence for relational databases.
  See: `http://hibernate.org/`

- Jupyter, Numpy, Matplotlib, Plot.ly, Bokeh
  Various extension which were used to conduct the experiments and plot the results. These can be installed by the build-in package manager `pip` included in Python 3. The libaries needed for the example are already prepared in the virtual machine.

- Word2Vec (Mikulov et al.)
  This tool provides an efficient implementation of the continuous bag-of-words and skip-gram architectures for computing vector representations of words. We used it as a one-time preprocessing tool for our data. See: `https://code.google.com/archive/p/word2vec/`

- Google Gson, Google Guava Collections
  Google Gson, a Java serialization/deserialization library that can convert Java Objects into JSON and back was used for the analysis.
  The Guava project contains several of Google's core libraries that we rely on in our project. Only collections and some data primitives were used.
  See: `https://github.com/google/gson`, `https://github.com/google/guava`

- Maite (Vojtěch Knyttl)
  This library is attached in compiled version of our application with all the other libraries. The source code can be obtained from Vojtěch Knyttl upon request.

A.3 GoOutSearcher API

The basic API of our server is shown in A.1. We list only the most important subset of the API calls. We specify for every method its name and parameters, the return value is always the `HashMap` which contains field *status* that uses HTTP codes to inform the client. The other fields may vary depending on the method. The detailed description of the API is provided in the `READ.me` file in the `/Example` directory.

A.3.1 Example

In the the folder `/Example` there is prepared a working environment with our server. The `VirtualBox` image is running Linux, Debian 8. This image contains all the necessary prerequisites. There is an instance of `MySQL` server with the populated database, a compiled

method name	parameters
parseLogs	String inputFile, String outputFile
parseLogs	String query, Date start, Date end, String outputFile
readFileImportToDB	String file
getRunningTasks	
killTask	int id
callRankLibLearn	String fileName, String modelName, int rankerType, String normalize, String metric, String features, String validationFile
callRankLibLearnCV	String fileName, String modelName, int rankerType, String normalize, String metric, String features
callRankLibEvaluate	String modelName, String testFile, String scoreFile
setModelFile	String modelFile, int rankType, String metric
searchForQuery	String query, HashMap<String, String> params
storeLogToDb	String timestamp, String query, String clickedType, int clickedId, String results
correctQuery	String query
prepareDBpedia	
prepareW2V	
createIDFTable	

Table A.1: GoOutSearcher API

version of this thesis and a simple implementation of XML-RPC client written in Python. All the data are precomputed and stored in binary files. Most importantly, in the `/Example` folder there is also a `READ.me` file which provides further guidance how to use the system, full API and hints how to query the server from the client.

APPENDIX B

Figures and Graphs

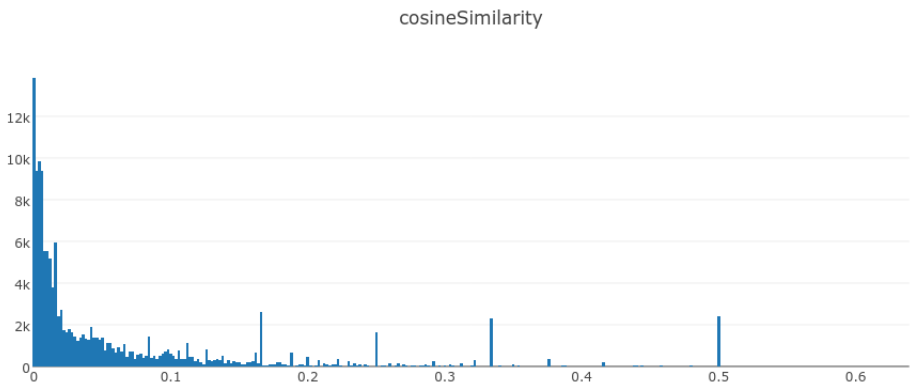

Figure B.1: Cosine similarity histogram

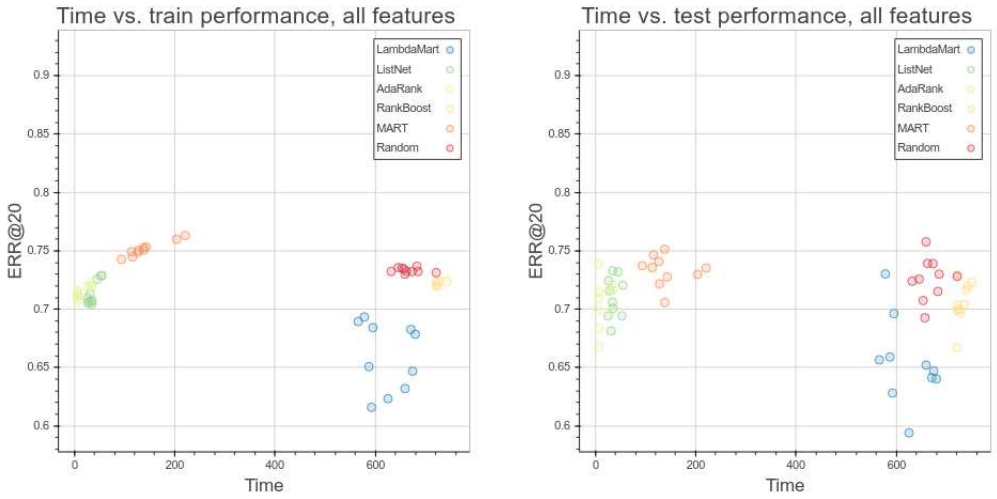

Figure B.2: Learning time for algorithms with all features

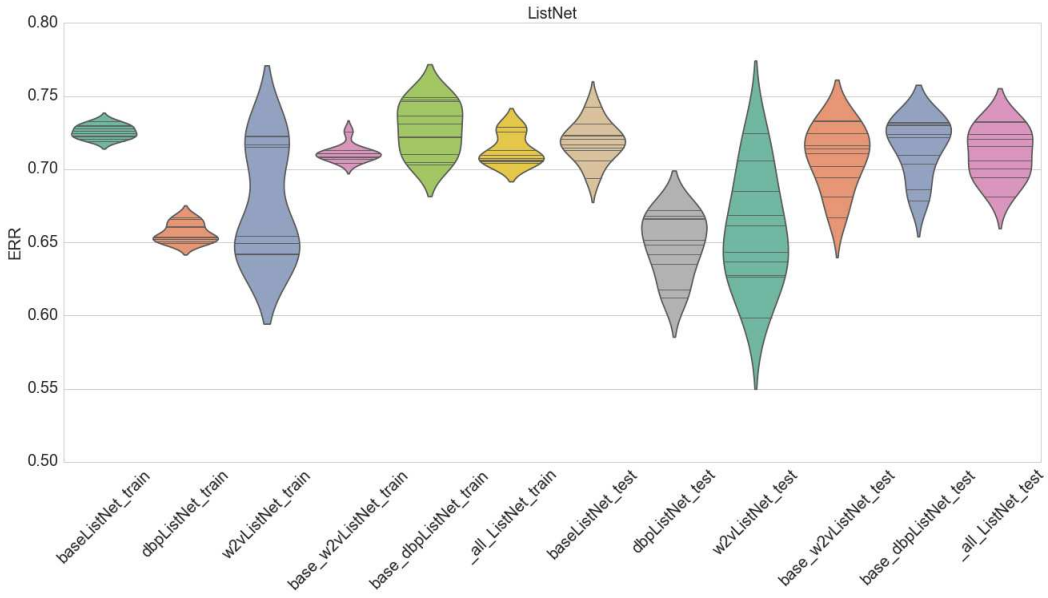

Figure B.3: Feature sets for ListNet

	event	performer	venue	tfidf	cosSim	BM25	qL	tL	fOcc	video	videoV	views	innerE	freshness	email
event	1.00	-0.78	-0.53	-0.07	-0.32	-0.31	0.24	0.04	0.19	-0.28	-0.28	-0.21	-0.11	-0.42	-0.02
performer	-0.78	1.00	-0.11	0.05	0.31	0.21	-0.19	-0.02	-0.16	0.37	0.36	-0.07	-0.02	0.39	0.01
venue	-0.53	-0.11	1.00	0.05	0.09	0.20	-0.12	-0.03	-0.09	-0.06	-0.05	0.43	0.21	0.15	0.02
tfidf	-0.07	0.05	0.05	1.00	0.06	0.62	0.01	0.47	-0.18	0.04	0.04	0.08	0.02	0.04	0.01
cosSimi	-0.32	0.31	0.09	0.06	1.00	0.16	-0.11	0.02	-0.08	0.18	0.18	0.08	0.03	0.09	0.03
BM25	-0.31	0.21	0.20	0.62	0.16	1.00	-0.04	0.31	-0.24	0.12	0.11	0.16	0.08	0.16	0.04
queryLength	0.24	-0.19	-0.12	0.01	-0.11	-0.04	1.00	0.08	0.05	-0.03	-0.03	-0.02	-0.01	-0.25	-0.04
textLength	0.04	-0.02	-0.03	0.47	0.02	0.31	0.08	1.00	0.16	0.02	0.02	0.06	-0.00	0.04	0.04
firstOccurence	0.19	-0.16	-0.09	-0.18	-0.08	-0.24	0.05	0.16	1.00	-0.03	-0.04	-0.04	-0.02	-0.04	0.00
video	-0.28	0.37	-0.06	0.04	0.18	0.12	-0.03	0.02	-0.03	1.00	0.78	0.03	-0.01	0.05	-0.01
videoViews	-0.28	0.36	-0.05	0.04	0.18	0.11	-0.03	0.02	-0.04	0.78	1.00	0.02	-0.01	0.04	-0.01
views	-0.21	-0.07	0.43	0.08	0.08	0.16	-0.02	0.06	-0.04	0.03	0.02	1.00	0.11	-0.06	-0.01
innerEvents	-0.11	-0.02	0.21	0.02	0.03	0.08	-0.01	-0.00	-0.02	-0.01	-0.01	0.11	1.00	0.05	0.01
freshness	-0.42	0.39	0.15	0.04	0.09	0.16	-0.25	0.04	-0.04	0.05	0.04	-0.06	0.05	1.00	0.01
email	-0.02	0.01	0.02	0.01	0.03	0.04	-0.04	0.04	0.00	-0.01	-0.01	-0.01	0.01	0.01	1.00

Table B.1: Correlation coefficients between features

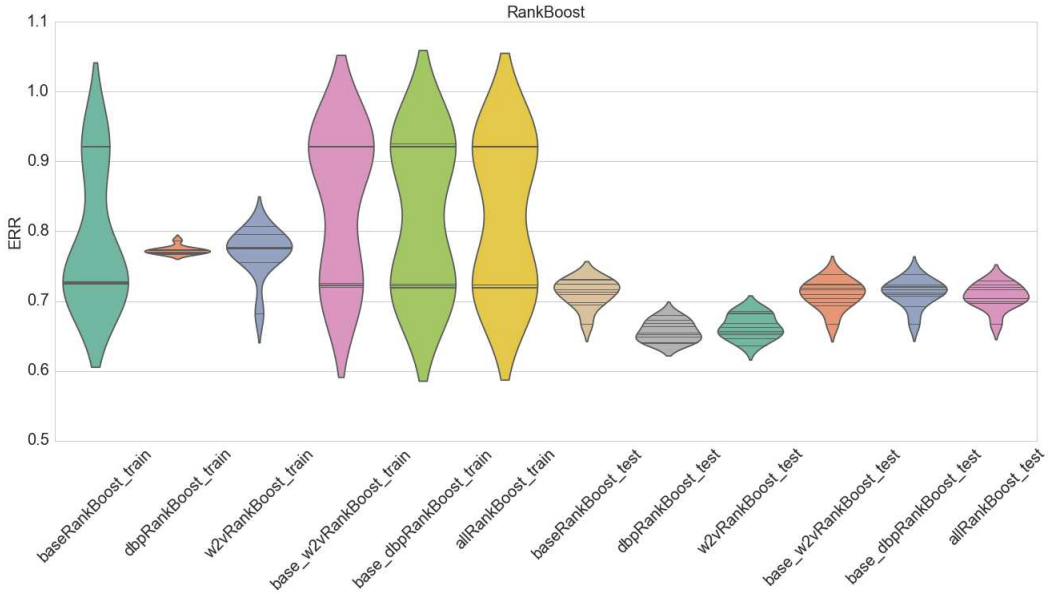

Figure B.4: Feature sets for RankBoost

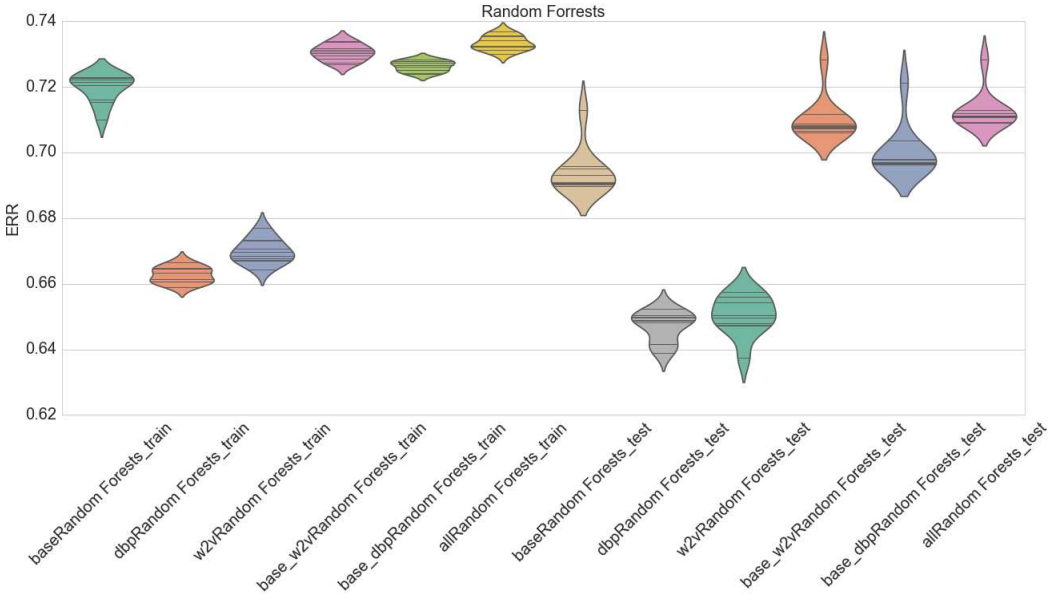

Figure B.5: Feature sets for Random Forests

	zero_test	abs_test	zero_train	abs_train
base_features	0.717	0.7178	0.7233	0.7224
dbp	0.6694	0.6694	0.7025	0.7025
w2v	0.6896	0.6896	0.7191	0.7191
base_w2v	0.7215	0.7193	0.7334	0.7285
base_dbp	0.7244	0.731	0.7358	0.7441
all	0.7332	0.7329:q	0.7515	0.751

Table B.2: Comparison of freshness values zeroed and modified by absolute value

List of Figures

List of Tables

Bibliography

[1] Christopher JC Burges. "From ranknet to lambdarank to lambdamart: An overview". In: *Learning* 11 (2010), pp. 23–581.

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

[3] Olivier Chapelle et al. "Expected reciprocal rank for graded relevance". In: *Proceedings of the 18th ACM conference on Information and knowledge management*. ACM. 2009, pp. 621–630.

[4] V Dang and B Croft. "Feature selection for document ranking using best first search and coordinate ascent". In: *Sigir workshop on feature generation and selection for information retrieval*. 2010.

[5] Yajuan Duan et al. "An empirical study on learning to rank of tweets". In: *Proceedings of the 23rd International Conference on Computational Linguistics*. Association for Computational Linguistics. 2010, pp. 295–303.

[6] Roy Thomas Fielding. "Architectural styles and the design of network-based software architectures". PhD thesis. University of California, Irvine, 2000.

[7] Yoav Freund et al. "An Efficient Boosting Algorithm for Combining Preferences". In: *J. Mach. Learn. Res.* 4 (Dec. 2003), pp. 933–969. ISSN: 1532-4435. URL: `http://dl.acm.org/citation.cfm?id=945365.964285`.

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

[9] Kalervo Järvelin and Jaana Kekäläinen. "Cumulated gain-based evaluation of IR techniques". In: *ACM Transactions on Information Systems* vol. 20.issue 4 (), pp. 422–446. ISSN: 10468188. DOI: `10.1145/582415.582418`. URL: `http://portal.acm.org/citation.cfm?doid=582415.582418`.

[10] Christopher D Manning, Prabhakar Raghavan and Hinrich Schütze. *Introduction to information retrieval*. New York: Cambridge University Press, 2008. ISBN: 0521865719.

[11] Tomas Mikolov et al. “Efficient estimation of word representations in vector space”. In: *arXiv preprint arXiv:1301.3781* (2013).

[12] Mu Zhu. “Recall, precision and average precision”. In: *Department of Statistics and Actuarial Science, University of Waterloo, Waterloo* 2 (2004).