FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Improving Learning to Rank Algorithms
Student:	Huy Hoang Vu
Supervisor:	doc. Ing. Pavel Kordík, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of winter semester 2019/20

Instructions

Survey learning to rank algorithms and collaborative filtering based algorithms. Implement a baseline algorithm for learning to rank on top of Yandex Personalized Web Search Dataset. Design and implement an advanced learning to rank algorithm utilising colaborative filtering, compare the performance to the baseline and other submissions in the Yandex learning to rank challenge. Analyse a computational complexity of implemented algorithms and demonstate scalability on training data set of increasing size.

References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 22, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Improving Learning to Rank Algorithms

Vu Huy Hoang

Department of Theoretical Computer Science
Supervisor: doc. Ing. Pavel Kordík Ph.D.

May 14, 2018

Acknowledgements

I'd like to thank my supervisor for pointing out mistakes in my thesis and helping me in correcting them. I'd like to thank my parents for their support. I'd like to thank Martin Bobek for his countless constructive criticisms. I'd also like to thank my friends Michal Cvach, Miroslav Sochor, Jan Uhlík, Josef Erik Sedláček, Matyáš Křišťan, Tung Anh Vu for their moral support and for the lessons they taught me.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 14, 2018 ……

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Vu, Huy Hoang. *Improving Learning to Rank Algorithms*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstract

In this thesis I explore existing approaches to the learning to rank problem and collaborative filtering methods, and apply them to Yandex's dataset provided in the Personalized Web Search Challenge competition on Kaggle.com. I build on the existing submissions by replicating the top competitor's feature extraction from the dataset. Then I implement and apply ES-Rank and matrix factorization on these features and test if matrix factorization based collaborative filtering significantly increases the overall performance of the algorithm. Then I compare the performance of the implemented algorithms to other submissions on Kaggle. Lastly I analyze the time complexity of my solution.

Keywords information retrieval, learning to rank, collaborative filtering, matrix factorization, evolutional strategy

Abstrakt

V této práci se zabývám existujícími algoritmy pro úlohu přeřazení URL podle relevance na základě uživatelského dotazu do vyhledávače a metodami kolaborativního filtrování, které uvádím v rešerši. Vybrané algoritmy, což jsou ES-Rank a maticová faktorizace, pak implementuji a použiji na dataset poskytnutý společností Yandex v rámci soutěže Personalized Web Search Challenge na Kaggle.com. Poté porovnávám přesnost řazení s ostatními řešeními na Kaggle.com. Následně testuji, jestli kolaborativní filtrování metodou maticové faktorizace významně zvyšuje přesnost řazení. Nakonec analyzuji časovou složitost svého řešení.

Klíčová slova získávání informací, učení se řadit, kolaborativní filtrování, maticová faktorizace, evoluční strategie

Contents

List of Figures

List of Tables

Introduction

In recent years, a lot of research has been done on algorithms solving the learning to rank problem because companies collect and want to utilize massive amounts of data about their clients or users. Correctly reranking URLs or products, which are presented to users, increases sales and user satisfaction. Therefore, the need for high quality information retrieval and ranking algorithms arises.

In the year 2014 the Russian company called Yandex, that runs it's own search engine, held a competition on Kaggle.com. The goal of the competition was to personalize web search for users by reranking search engine results based on relevance and the users' personal preferences. The winners of this competition used Simon Funk's matrix factorization as a collaborative filtering feature in their solution, but reported a marginal increase in their model's overall performance. They also used LambdaMART, a learning to rank algorithm, which training is not parallelizable.

In the following chapters I analyze existing learning to rank algorithms and collaborative filtering algorithms. I also analyze the competition winner's solution and heavily use their ideas in my own solution. I then implement a baseline model without collaborative filtering for later comparison with my final solution that uses collaborative filtering to see if this technique can significantly increase performance.

Chapter 1

Learning to Rank

1.1 Problem description

Let $\mathcal{Q} = \{q_1, q_2, \ldots, q_n\}$ be the set of n given queries, $\mathbf{d_i} = \{d_1^i, d_2^i, \ldots, d_{m(q_i)}^i\}$ is the set of $m(q_i)$ documents retrieved upon query q_i, $\mathbf{y_i} = \{y_1^i, y_2^i, \ldots, y_{m(q_i)}^i\}$ is the set of relevance labels, where label y_j^i is the relevance label of j-th document d_j^i with respect to query q_i. The type of labels is chosen manually by experts. Usually a label takes on the value of either 0 or 1 indicating if the document is relevant or not, or an integer ranging from 0 up to a small integer l is used as a relevance grade, i.e. $y_j^i \in \mathcal{R} = \{0, 1, 2, \ldots, l\}$, the higher the grade the more relevant the document is. The training dataset becomes

$$\mathcal{S}_{train} = \{(q_i, \mathbf{d_i}, \mathbf{y_i})\}_{i=1}^{n}. \tag{1.1}$$

A feature vector $\vec{x}_j^i \in \mathbb{R}^d$ is specified for each query-document pair (q_i, d_j^i). An example of a feature might be how many times a search term from query q_i occurs in document d_j^i.

Typically a scoring function $f : \mathbb{R}^d \to \mathbb{R}$ is used as the ranking model. The documents are then sorted by $f(\vec{x}_j^i)$ in descending order creating a permutation $\pi_i(\mathbf{d_i}, f)$ which maps documents to a position (rank). Instead of ordinal regression a classifier can be used, i.e. a classifying function $f : \mathbb{R}^d \to \mathcal{R}$ which maps a document to a relevance grade. The documents are then sorted by their grade in descending order.

The objective is to find a ranking function f that maximizes a performance measure (see section 1.2) directly or one that minimizes a loss function.

The notation above was taken from [2].

1.2 Performance measures

1.2.1 Assumptions about user behavior

To even measure the performance of a ranking function, a few assumptions about the user interaction with documents is made as described in the following frequently used models.

Position model Users tend to look at and click the first few documents. To cope with this bias, the position model assumes that users examine the documents in a linear fashion, from top position to bottom position, and only click on documents that are a) examined and b) relevant to the user. The lower a document is ranked, the lower is its probability to be examined and thus it has a low probability to be clicked. Measures Normalized Discounted Cumulative Gain and Mean Average Precision work under this model.

Cascade model The cascade model is an extension of the position model in that it assumes that documents are examined from top to bottom and the user stops when he/she is satisfied, but it also takes into account relevances of higher ranked documents. If for example a highly relevant document is ranked at position 3 and documents at positions 1 and 2 are also highly relevant, then it has a lower probability to be clicked than in the case where documents at positions 1 and 2 are not relevant. Expected Reciprocal Rank is an example measure that uses this model.

1.2.2 Notation used in following subsections

- $\mathcal{Q} = \{q_1, q_2, \ldots, q_n\}$ is the set of queries with each q_i associated with documents $\mathbf{d_i}$ and labels $\mathbf{y_i}$
- f is the ranking function that scores each document based on its relevance effectively assigning it a position
- $\mathbf{d_i} = \{d^i_1, d^i_2, \ldots, d^i_{m(q_i)}\}$ is the set of $m(q_i)$ documents retrieved upon query q_i
- $\mathbf{y_i} = \{y^i_1, y^i_2, \ldots, y^i_{m(q_i)}\}$ is the set of relevance labels, where each label y^i_j is associated with document d^i_j

1.2.3 Winner Takes All

Winner Takes All (WTA) measure works only with binary relevance grades, that is $y_j^i \in \{0, 1\}$. WTA is defined like this

$$WTA(f, \mathbf{d_i}, \mathbf{y_i}) = \begin{cases} 1, & \text{the document at top position is relevant} \\ 0, & \text{the document at top position is not relevant} \end{cases} \tag{1.2}$$

The WTA measure is simple, it returns 1 if the document at position 1 has a relevance of 1 (relevant), otherwise it returns 0.

1.2.4 Precision

Precision (P) is the ratio between relevant documents and all retrieved documents.

1.2.5 Mean Average Precision

Average Precision (AP) is defined as

$$AP(f, \mathbf{d_i}, \mathbf{y_i}) = \frac{1}{n_{rel}^i} \sum_{p:\ y_p^i=1} \frac{\sum_{j=1}^{p} y_j^i}{p}, \tag{1.3}$$

where y_j^i is the binary relevance label of document d_j^i, p is the position of document d_p^i, and n_{rel}^i is the number of relevant documents retrieved upon query q_i. In other words take a relevant document with its position p, divide the number of relevant documents with position $j \leq p$, and sum these fractions over all relevant documents, and divide by n_{rel}.

Unlike previously mentioned measures AP takes into account the position of relevant documents and rewards correctly sorted documents. If all documents $\mathbf{d_i}$ are sorted perfectly according to their relevance, then $AP(f, \mathbf{d_i}, \mathbf{y_i}) = 1$.

Mean Average Precision (MAP) is then the AP averaged over all n queries $\mathcal{Q}$, i.e.

$$MAP(f, \mathbf{d_i}, \mathbf{y_i}) = \frac{1}{n} \sum_{i:\ q_i \in \mathcal{Q}} AP(f, \mathbf{d_i}, \mathbf{y_i}) \tag{1.4}$$

1.2.6 Normalized Discounted Cumulative Gain

Discounted Cumulative Gain (DCG) is a measure that takes into account positions of documents like MAP, but it can also handle multigraded relevances,

that is $y_j^i \in \{0, 1, 2, \ldots, l\}$. It is usually defined as

$$DCG(f, \mathbf{d_i}, \mathbf{y_i}) = \sum_{j=1}^{m(q_i)} \frac{2^{y_j^i} - 1}{log(j+1)}, \tag{1.5}$$

where y_j^i is the relevance label of the document ranked at position j and $m(q_i)$ is the number of documents retrieved upon query q_i. The position discount function $log(j+1)$ penalizes the gain function $2^{y_j^i} - 1$ with increasing position j, so to maximize this measure it is advantageous to have highly relevant documents at top positions so that they're the least discounted.

Normalized Discounted Cumulative Gain (NDCG) returns a real number between 0 and 1. NDCG is calculated as

$$NDCG(f, \mathbf{d_i}, \mathbf{y_i}) = \frac{DCG(f, \mathbf{d_i}, \mathbf{y_i})}{IDCG(f, \mathbf{d_i}, \mathbf{y_i})}, \tag{1.6}$$

where $IDCG(f, \mathbf{d_i}, \mathbf{y_i})$ is the ideal DCG, that is the DCG of documents that are perfectly sorted according to their relevance grade in descending order, i.e. $(\forall j, k)(j < k \implies y_j^i \geq y_k^i)$.

1.2.7 Expected Reciprocal Rank

Proposed in [3], the Expected Reciprocal Rank (ERR) follows the cascade model.

Let R be a function of relevance grades, y_j^i be the relevance grade of document at j-th position, and y_{max} be the highest possible grade. R_j is the probability that the document at position j is relevant and is defined as

$$R_j = \frac{2^{y_j^i} - 1}{2^{y_{max}}} \tag{1.7}$$

ERR is then defined as

$$ERR(f, \mathbf{d_i}, \mathbf{y_i}) = \sum_{r=1}^{m(q_i)} \frac{1}{r} R_r \prod_{j=1}^{r-1} (1 - R_j) \tag{1.8}$$

Similarly to NDCG, it has a gain function and a discount function. In this case the gain function is R_r and the discount function is $\frac{1}{r} \prod_{j=1}^{r-1} (1 - R_j)$. Notice that the discount function is the probability that the user is not satisfied with top $r-1$ documents and is satisfied with the r-th document. The discount function takes into account the positions of documents as well as relevance of higher ranked documents.

1.2.8 Loss function

A loss function can be used instead of the measures described above. A loss function that is differentiable might be desirable. An example of such loss function is mean squared error is Meas Squared Error

$$MSE(f, \mathbf{d_i}, \mathbf{y_i}) = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{m(q_i)} (y_j^i - f(\vec{x}_j^i))^2. \tag{1.9}$$

1.3 Categories of LTR algorithms

There are three main approaches to learning to rank. They differ in the way each of them handles data samples and how they measure performance.

Point-wise The point-wise approach treats each query-document pair as a single data sample and uses regression or classification to predict its relevance. It does not look at relationships between documents, instead it just takes a document as a single sample and predicts it's relevance e.g. by assigning it a relevance grade. It also measures the performance document by document.

Pair-wise The pair-wise approach treats a pair of documents as a data sample and tries to classify it as correct if the first document is more relevant than the second or incorrect (inversion) otherwise. Unlike the point-wise approach, pair-wise captures the relationship between two documents.

List-wise The list-wise approach takes the whole document list $\mathbf{d_i}$ as a sample and finds a permutation that optimizes a performance measure or loss function. This method fully captures the relationships between documents in the list.

1.4 Learning to Rank algorithms

1.4.1 Random Forest

Random Forest [4] is an ensemble of decision trees. An ensemble is a group of models that combines the predictions of these models into a single prediction, thereby having a better performance than an individual model. Typically regression trees are used for ranking. Bootstrap aggregating (bagging) is used to create the ensemble, where the dataset is randomly sampled with replacement creating a subsample of same size. The dataset in the case of ranking is a set of query-document pairs each associated with a feature vector $\vec{x}_j^i$. Regression trees or classification trees are then fitted onto subsamples and the forest's prediction is a weighted or unweighted average or majority vote of individual trees. However a tree doesn't use all available features, instead a

random subset of features is selected for each tree during training. This is called random subspace method or feature bagging [5]. It is used to reduce correlation between trees because the same highly explanatory features are often selected for best split in many trees. The problem where many trees use the same features and therefore are correlated is avoided.

This algorithm is not only quick in prediction but also quick in training because trees can be trained in parallel.

1.4.2 MART

MART or Multiple Additive Regression Trees [6] is an ensemble of gradient boosted regression trees. Boosting refers to the technique that is used to build ensembles. In boosting each partial model is added to the ensemble iteratively to correct the ensembles "flaws". By flaw I mean the incorrectly classified or predicted data points from the training set.

MART training is described in the following steps

1. Fit a fixed size regression tree on the dataset according to ground truth labels $(y_j^i)_0$ creating an ensemble of one tree $F_0(\vec{x}_j^i) = \alpha_0 f_0(\vec{x}_j^i)$, where α_0 is a real valued weight and f_0 is the prediction of the tree

2. For each data point calculate the residual (difference) $(y_j^i)_m - F_m(\vec{x}_j^i)$ and use this residual as new label $(y_j^i)_{m+1}$ in the next iteration $(m+1)$

3. Create a new ensemble $F_{m+1}(\vec{x}_j^i) = F_m(\vec{x}_j^i) + \alpha_{m+1} f_{m+1}(\vec{x}_j^i)$ by fitting a new regression tree f_{m+1} on the dataset according residuals $(y_j^i)_{m+1}$ calculated in step 2

4. Terminate if enough trees are created, otherwise set m to $m+1$ and go to step 2

Notice that in the case of MART the "flaws" are residuals $(y_j^i)_m - F_m(\vec{x}_j^i)$ of the ensemble. At each step a loss function is minimized. The loss function is the squared error

$$\frac{((y_j^i)_m - F_m(\vec{x}_j^i))^2}{2}. \tag{1.10}$$

The derivative of this error with respect to $F_m(\vec{x}_j^i)$ is exactly

$$\frac{d\frac{((y_j^i)_m - F_m(\vec{x}_j^i))^2}{2}}{dF_m(\vec{x}_j^i)} = -((y_j^i)_m - F_m(\vec{x}_j^i)) \tag{1.11}$$

By training a new tree on these residuals the error is minimized by essentially subtracting this derivative from the ensemble prediction with learning step α.

By the nature of boosting that requires iterative improvements, this algorithm cannot be parallelized.

1.4.3 RankNet

RankNet [7] is a pairwise method, that is it's trained on pairs of documents and minimizes the number of inversions. However it's output is a mapping function f that maps documents to real values because ranking by comparing all pairs of documents is time consuming. Then the ranking is ordinal regressions, i.e. f maps to a real value and the documents are sorted according to those values. The underlying model of this algorithm is a neural network that is trained to minimize the cost function C described bellow.

The training of this algorithm works as follows. Each query-document pair (q_i, d_j^i) has a feature vector $\vec{x}_j^i$ and f maps this vector to a real value $f(\vec{x}_j^i)$. Let A and B be documents (q_1, d_1^1) and (q_1, d_2^1) for shorthand. Let $A > B$ denote that A should be rank higher than B. $P(A > B)$ is the probability that A should be ranked higher than B and is defined as

$$P(A > B) = \frac{1}{1 + e^{-\sigma(f(\vec{x}_A) - f(\vec{x}_B))}}, \tag{1.12}$$

where σ is a parameter of the neural network.

The cost function C is defined as

$$C = -\bar{P}(A > B)log(P(A > B)) - (1 - \bar{P}(A > B))log(1 - P(A > B)), \tag{1.13}$$

where $\bar{P}(A > B)$ is the known probability that A ranks higher than B. $\bar{P}(A > B)$ is set to 0 if A ranks lower than B, 0.5 if A and B have the same relevance, and 1 if A ranks higher than B. Since C is differentiable, it is minimized with backpropagation.

1.4.4 AdaRank

AdaRank [8] is a boosting algorithm that iteratively trains T weak rankers and then linearly combines them. A weak ranker is a ranking algorithm that doesn't rank well on its own but adds performance if used in an ensemble. AdaRank is a listwise algorithm so it treats queries as a single data sample.

The training of AdaRank works as follows: $for\ t = 1, ..., T$

1. Create a weak ranker h_t with weighted distribution P_t
2. Choose $\alpha_t = \frac{1}{2} ln \frac{\sum_{i=1}^{n} P_t(i)(1+E(h_t, \mathbf{d_i}, \mathbf{y_i}))}{\sum_{i=1}^{n} P_t(i)(1+E(h_t, \mathbf{d_i}, \mathbf{y_i}))}$
3. Create $f_t = \sum_{k=1}^{t} \alpha_k h_k(\vec{x})$
4. Update $P_{t+1}(i) = \frac{e^{-E(f_t, \mathbf{d_i}, \mathbf{y_i})}}{\sum_{j=1}^{n} e^{-E(f_t, \mathbf{d_j}, \mathbf{y_j})}}$

$P_t(i)$ is the coefficient of query q_i that measures how difficult q_i is to rank at iteration t. $E(h_t, \mathbf{d_i}, \mathbf{y_i}))$ is the performance measure (NDCG, MAP, WTA) of h_t on query q_i. Notice that P_t changes at every addition of a new weak ranker as difficult queries get correctly ranked over time. Also notice that as the performance E for query q_i increases, $P_{t+1}(i)$ decreases. The importance of an individual weak ranker is captured with its coefficient α_t. α_t increases based on performance of h_t weighted with $P_t(i)$, in other words α_t increases as h_t's ranking performanc increases on difficult queries.

In [8] the authors use a single feature as the weak ranker so the final ensemble is a linear combination of αs and features. The weak ranker h_t is chosen such that $\sum_{i=1}^{n} P_t(i)E(h_t, \mathbf{d_i}, \mathbf{y_i})$ is maximized.

1.4.5 ES-Rank

Since I implement this algorithm I describe it in more detail.

ES-Rank or Evolutional Strategy Ranking was proposed in [9]. It uses the optimization technique called Evolutional Strategy (ES).

The objective is to optimize the scoring function

$$f(\vec{x}_j^i) = \vec{w}, \vec{x}_j^i = \sum_{k=1}^{d} w_k x_{jk}^i \tag{1.14}$$

by finding the best weight vector $\vec{w} \in \mathbb{R}^d$ such that f scores document a with a higher number than document b if document a is more relevant than document b.

ES-Rank uses the simplest form of Evolutional Strategy $(1+1)$-ES to find $\vec{w}$. As seen in algorithm 1, ES-Rank stores two weight vectors (genotypes) $parent\_gen$ and $offspring\_gen$. These two vectors are initially set to zero or otherwise initialized (see chapter 3). Then these weights are optimized over a number of generations by mutating the offspring. If the mutation of the offspring genotype yields an increase in its fitness compared to the parent's fitness, then the same mutation on the same weights is applied in the next generation and the offspring replaces its parent, otherwise the offspring stays the same as the parent and a new mutating scheme is created in the next generation. A new mutation scheme is created by randomly choosing a number r of genes to be mutated and then r times randomly choosing a weight from the offspring and adding $mutation\_step_m$ to it. $mutation\_step_m$ is defined as

$$mutation\_step_m = Gaussian(0,1) * e^{F(x;0,1)}, \tag{1.15}$$

where $Gaussian(0,1)$ is a random number from normal distribution with mean 0 and standard deviation 1, x is a random number between 0 and 1 uniformly distributed, e is Euler's number, and $F(x;0,1)$ is the cumulative distribution

function of Cauchy distribution with 0 location and 1 scale applied on x, that is $F(x; 0, 1) = \frac{1}{\pi} arctan(x) + \frac{1}{2}$.

Algorithm 1 ES-Rank

Input: Query q_i, documents $\mathbf{d_i}$, labels $\mathbf{y_i}$, feature vectors $\vec{x}_j^i$ of each document d_j^i

Output: A linear ranking function $f(\vec{x}_j^i)$ that maps a real valued score to document d_j^i

$successful\_mutation \leftarrow$ false
for $parent\_gen_i \in parent\_gen$ **do**
    $parent\_gen_i \leftarrow 0.0$
$offspring\_gen \leftarrow parent\_gen$
for $g \leftarrow 1$ to $max\_generations$ **do**
    if $successful\_mutation =$ true **then**
        mutate $offspring\_gen$ using $mutation\_step$ from generation $g-1$
    else
        $r \leftarrow$ random number from 1 to length($offspring\_gen$)
        for $m = 1$ to r **do**
            $i \leftarrow$ random number from 1 to length($offspring\_gen$)
            $offspring\_gen_i \leftarrow offspring\_gen_i + mutation\_step_m$
        $mutation\_step \leftarrow$ sequence of mutations $mutation\_step_m$
    if Fitness($offspring\_gen$) > Fitness($parent\_gen$) **then**
        $successful\_mutation \leftarrow$ true
        $parent\_gen \leftarrow offspring\_gen$
    else
        $successful\_mutation \leftarrow$ false
        $offspring\_gen \leftarrow parent\_gen$

Chapter 2

Collaborative filtering

Collaborative filtering (CF) is a collective term for algorithms from recommendation systems in which items are recommended to users based on their similarity in preferences with other users. The prediction of a user's interest or the recommendation of items to him/her is based on behavior of other similar users.

2.1 Usage in Learning to Rank

The notion of similar users can be used in LTR as well. When ranking documents for user A based on his/her preferences and search history, there can be a document D that A has never seen. If a user B that has similar preferences to A has an opinion on document D (he/she clicked on the document or not), then the same kind of opinion on D can be expected from A and therefore accordingly ranked. For example if B wasn't interested in document D (didn't click it), then D would be ranked lower for A because B has similar tastes to A.

2.2 Data representation

The data is usually represented in a matrix of ratings called *utility matrix* of *users* and *items*. Each user-item pair is a rating that represents the users degree of preference of that item. Ratings are from an ordered set of values typically integers ranging from 1 to 5. An example of such ratings could be how many stars a user gives to a movie as illustrated in table 2.1.

In table 2.1 an observation can be made that since both users A and B rated movie HP1 highly, they share similar interests and a recommendation of movie HP2 to user A can be made because user B also rated HP2 highly.

Table 2.1: Utility matrix of users and movies from [1]

	HP1	HP2	HP3	TW	SW1	SW2	SW3
A	4			5	1		
B	5	5	4				
C				2	4	5	
D		3					3

Notice that some ratings can be blank and in practice blank ratings are even greater in number. The utility matrix is usually very sparse. Out of millions of movies an average user might have rated fifty of them.

2.3 Similarity measurement

Suppose that a user's interest is represented by his/her row vector in the utility matrix. In [1] cosine similarity is recommended as similarity measure. Cosine similarity of vectors $\vec{a}$ and $\vec{b}$ is defined as a ratio between their dot product and the product of their magnitudes

$$sim_{cos}(\vec{a}, \vec{b}) = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} a_i^2}\sqrt{\sum_{i=1}^{n} b_i^2}} \tag{2.1}$$

To calculate this distance, blanks in the utility matrix have to be filled first. A straightforward way would be to fill them with zeroes but that would mean that the user rated the movie very poorly. A better way would be to subtract the user's average rating from his/her ratings and then filling in the blanks with zeroes. By subtracting the average, ratings will be centered around zero. A positive rating would mean that the user liked the movie more than average and a negative rating would mean the user liked the movie less than average [10]. This also solves the problem of high raters and low raters, i.e. users who tend to give high ratings or users who tend to give low ratings. A similarity measure that does this is called Pearson's correlation and is defined as

$$sim_{pearson}(\vec{a}, \vec{b}) = \frac{\sum_{i=1}^{n} (a_i - \bar{a})(b_i - \bar{b})}{\sqrt{\sum_{i=1}^{n} (a_i - \bar{a})^2}\sqrt{\sum_{i=1}^{n} (b_i - \bar{b})^2}}. \tag{2.2}$$

2.4 Predicting ratings

Let N be the the set of k users that are the most similar to user u and have rated item i, and $\vec{r}_u$ be the ratings vector of user u. The prediction for user u and item i could simply be the average of ratings of item i by users N

$$r_{ui} = \frac{1}{k} \sum_{y \in N} r_{yi} \tag{2.3}$$

or the average can be weighted with similarities of users N with user u

$$r_{ui} = \frac{\sum_{y \in N} sim_{cos}(\vec{r}_y, \vec{r}_u) r_{yi}}{\sum_{y \in N} |sim_{cos}(\vec{r}_y, \vec{r}_u)|}. \tag{2.4}$$

2.5 Matrix factorization

Another method for CF, described in [11], is the factorization (or decomposition) of the utility matrix into two long and thin matrices.

Let M be the utility matrix of users and items with m rows and n columns. The matrix factorization of matrix M are matrices U with m rows and k columns and V with k rows and n columns. Matrix M can then be reconstructed by multiplying matrices U and V, that is $M_{ij} = U_i V_j = \sum_{l=1}^{k} U_{il} V_{lj}$.

Each value in matrix U and V represents a so-called latent variable of users or items respectively. Suppose that M is a utility matrix of ratings for movies and U_i is the row vector of matrix U representing latent variables of user i. The concrete meaning of these latent variables is not known. For example, U_{i1} might mean how much user i likes action in movies and V_{1j} might mean how much action is in movie j but it is not known for certain. A rating prediction for user i and movie j is made by calculating the dot product of row $U_{i:}$ and column $V_{:j}$. If individual elements (latent variables) of vector $U_{i:}$ highly correlate with corresponding elements of vector $V_{:j}$, then a high rating is put out.

Matrix factorization is not true singular value decomposition (SVD) which can be used but has some problems associated with it. In SVD, missing values (blanks) have to be filled with a value. Secondly, storing such a utility matrix in memory with a large dataset is not feasible. Typically, ratings of millions of users and millions of items are kept. The utility matrix is very sparse so an incremental decomposition algorithm has been developed [12] (see section 2.6).

2.6 Incremental computation of matrix factorization

The incremental method, proposed by Simon Funk in [12], only stores known ratings (non-blank elements of the utility matrix), so it is memory efficient. Elements of matrices U and V can be initialized to anything but [12] shows that a head start can be achieved by initializing them to the average of all ratings. Then latent variables are trained one by one iteratively, that is at iteration f the values U_{if} and V_{fj} are changed for every known rating M_{ij}. This is done by minimizng the mean squared error (MSE) between known ratings and predictions.

As described in [11], the error function that is to be minimized is

$$E = \frac{1}{2}(M_{ij} - U_{i:}V_{:j})^2. \tag{2.5}$$

Gradient descent is used to minimize this error function and according to [12] it has no problems with local minima. The gradient of this function is

$$\frac{\partial E}{\partial U_{i:}} = -(M_{ij} - U_{i:}V_{:j})V_{:j} \tag{2.6}$$

$$\frac{\partial E}{\partial V_{:j}} = -(M_{ij} - U_{i:}V_{:j})U_{i:} \tag{2.7}$$

Because the error is being minimized, this gradient is subtracted from the values of matrices U and V during each iteration of this algorithm. The update step is performed like this

$$utemp = U_{if} \tag{2.8}$$

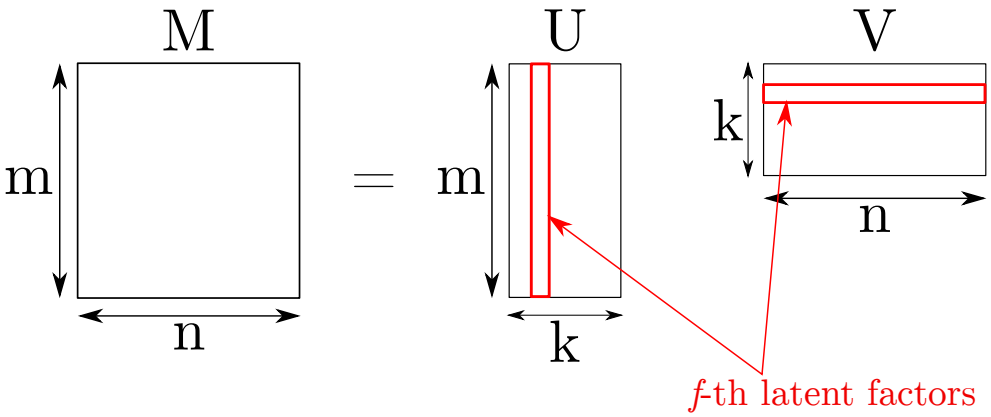
$$vtemp = V_{fj} \tag{2.9}$$

$$U_{if} = U_{if} + \alpha(M_{ij} - utemp \cdot vtemp)vtemp \tag{2.10}$$

$$V_{fj} = V_{fj} + \alpha(M_{ij} - utemp \cdot vtemp)utemp, \tag{2.11}$$

f is the latent variable being updated, α is the learning rate, and M_{ij} is the known rating of user i and item j.

Figure 2.1: Matrix factorization matrices



Chapter 3

Implementation

3.1 Yandex's dataset

The dataset, provided by Yandex in the Personalized Web Search Challenge competition, is 30 days worth of search sessions, each associated with a user. There are one or more queries in each session and search terms are specified for each query. Additionally, 10 URL-domain pairs are given with each query and click information is also given, that is the URLs which the user clicked on are known. Each query and click are marked with a timestamp that indicates when the action occurred from the beginning of the search session. These 30 days of sessions are split into 27 days of training sessions and 3 days of test sessions as illustrated in figure 3.1. In the Sessions selection phase seen in figure 3.1, a session randomly chosen for each user occurring in the 3 days of sessions. Each of the selected sessions must have a query that has a URL with relevance label of at least 1 (described below), this query is then considered as a test query and its session is considered as a test session. Information about clicks is removed from the test queries and is not disclosed by Yandex. Then the goal of the competition is to rerank the URLs of the test queries.

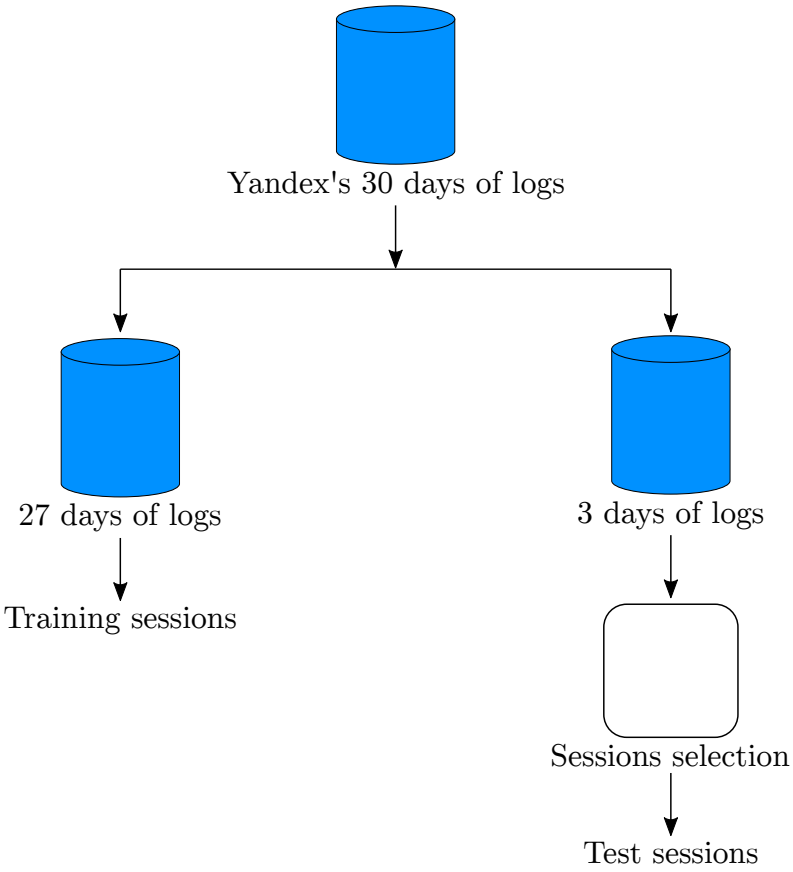
Every bit of the data is replaced with a meaningless IDs, all of the data is heavily anonymized this way, even the unit of timestamps is not given, so complex language analysis is not possible.

Yandex uses NDCG to measure performance in this competition. The relevance labels are dependent on dwell time (the duration between the user clicking on a URL and performing the next action or ending the search session). The relevance grade for a URL is defined as

- *0 (irrelevant)*, if the user doesn't click the URL or the dwell time was less than 50 units of time
- *1 (relevant)*, if the user clicked the URL and the dwell time was between 50 and 399 inclusively

- *2 (highly relevant)*, if the user clicked the URL and the dwell time was greater or equal to 400 or the user ended the session after the click

Figure 3.1: Yandex's data splitting process



Here are some noteworthy characteristics of this dataset:

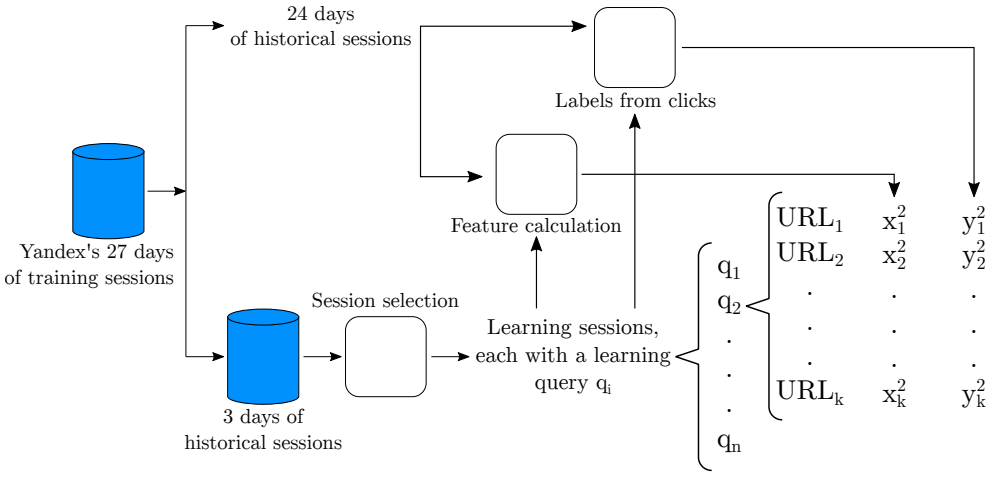
- Unique queries: 21,073,569
- Unique URLs: 70,348,426
- Unique users: 5,736,333
- Training sessions: 34,573,630
- Test sessions: 797,867
- Clicks in the training data: 64,693,054
- Total records in the log: 167,413,039

See these links (logs format, data description) for a more detailed description of the dataset.

3.2 Workflow

I split the 27 days of training data, that were provided by Yandex, into 24 days of historical sessions and 3 days of learning sessions similarly to team Dataiku Data Science Studio (Dataiku). Like Yandex, I selected learning queries (equivalent to Yandex's test queries) from the learning sessions that have at least one click with relevance label 1. Then I calculated a feature set for each of the URLs within the learning queries (see section 3.3). This resulted in query-URL pairs (q_i, d_j^i) each associated with a vector of features $\vec{x}_j^i$. The labels y_j^i for these query-URL pairs were relevance labels based on clicks which are described in section 3.1. Finally I applied algorithms described in the following sections on these features. To visualize the workflow, see figure 3.2.

Figure 3.2: Data splitting diagram



3.3 Features

For feature calculation, I followed the process team Dataiku described in [13] and I will briefly describe it in this section.

Before calculating features, Dataiku further divided the relevance label 0 into three more categories and in the end worked with 5 categories:

- *skip*, if the user examined the URL but didn't click it, i.e. every URL that was positioned above a clicked URL
- *miss*, if the user didn't examine the URL, i.e. every URL that was positioned after the last clicked URL
- *click0*, if the user clicked the URL and the dwell time was below 50

- *click1*, if the user clicked the URL and the dwell time was between 50 and 399 inclusively
- *click2*, if the user clicked the URL and the dwell time was greater or equal to 400

In the following features, two queries are considered the same if they have all terms (words) in common and their terms have the same order.

3.3.1 Non-personalized rank

This is the position of the URL as it was displayed to the user before reranking. It is the most important feature according to [13] because of the bias of users' tendency to click URLs at top positions.

3.3.2 Aggregate features

Aggregate features are defined as conditional probabilities

$$agg(l, \mathcal{P}) = P(outcome = l|\mathcal{P}) = \frac{count(l, \mathcal{P}) + p_l}{count(\mathcal{P}) + 1}, \tag{3.1}$$

where $l \in \{miss, skip, click0, click1, click2\}$, $\mathcal{P}$ is a predicate which can be a conjunction of the following predicates

- $URL = url_0$
- $Domain = domain_0$
- $User = user_0$
- $Query = query_0$

The conjunction must contain predicate $URL = url_0$ or $Domain = domain_0$. The zero subscript url_0 denotes that it is a concrete URL, $count(l, \mathcal{P})$ is the number of occurrences of outcome l given $\mathcal{P}$, $count(\mathcal{P})$ is the number of occurrences satisfying predicate $\mathcal{P}$, p_l is 1 if $l = miss$ and 0 otherwise. This is equivalent to the assumption that every URL has been missed at least once.

3.3.3 Mean Reciprocal Rank

The Mean Reciprocal Rank (MRR) is defined as

$$MRR(l, \mathcal{P}) = \sum_{r \in \mathcal{R}_{l,\mathcal{P}}} \frac{\frac{1}{r} + 0.283}{count(l, \mathcal{P}) + 1}, \tag{3.2}$$

where $\mathcal{R}_{l,\mathcal{P}}$ is the list of ranks (positions) of all occurrences matching outcome l and predicate $\mathcal{P}$.

3.3.4 User click habits

User click entropy is $-\sum_{r=1}^{10} P_{click}(r) \cdot log_2(P_{click}(r))$, where $P_{click}(r)$ is the probability of user $user_0$ clicking a URL displayed at position r.

User click counters are counters click12, click345, click678910 that count the number of times user $user_0$ clicked on URLs with positions {1,2}, {3,4,5}, {6,7,8,9,10} respectively.

User number of queries is the number of queries issued by user $user_0$ in the whole dataset.

User query length average is the average length (number of words) of queries issued by user $user_0$.

User session number of terms average is the number of words used by used by user $user_0$ in a session averaged over all of his/her sessions.

3.3.5 Query specific features

Query click entropy is $-\sum_{r=1}^{10} P_{click}(r) \cdot log_2(P_{click}(r))$, where $P_{click}(r)$ is the probability of query $query_0$'s URL positioned at r being clicked.

Query average position is the position of query $query_0$ within a session averaged over all sessions with $query_0$.

Query length is the number of terms in $query_0$.

Query MRR is the mean reciprocal rank of clicks of $query_0$, for example there was a click on a URL with rank (position) r upon $query_0$, the reciprocal rank of that click is $\frac{1}{r}$. The MRR is is the average of these reciprocal ranks.

Since there are a lot of predicate combinations, a lot of features can be calculated. I have only used those that Dataiku selected to be the most explanatory and they are listed in [13].

3.4 Baseline algorithm

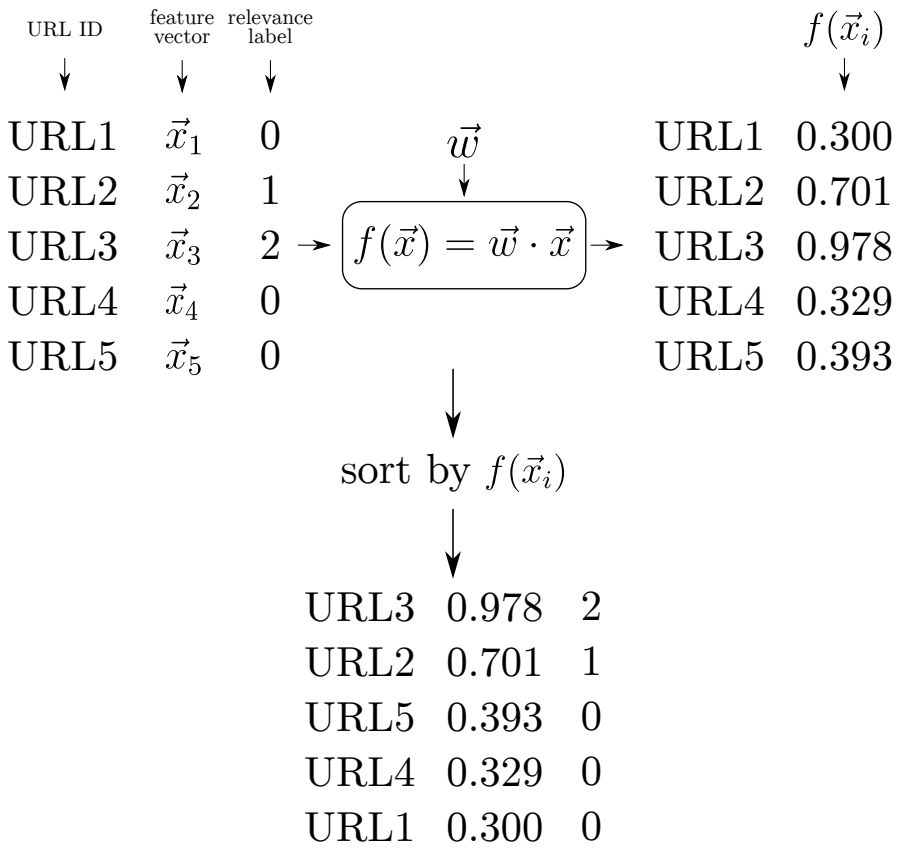
For the baseline model, I chose to implement ES-Rank (see section 1.4.5) because it is relatively straight forward to implement and has promising results according to [9].

The most ambiguous part of algorithm 1 is perhaps the fitness evaluation of weight vectors. It is also the part where most computation occurs. The fitness can be calculated with any measure described in section 1.2. Since Yandex uses NDCG, I will be using it as well. To calculate the NDCG for a weight vector, simply iterate over all queries and for each query, score it's URLs by

calculating the dot product of the URLs' feature vectors and the weight vector, then sort the URLs by these scores in descending order. Finally calculate the NDCG for the sorted URLs of each query and average the NDCG over all queries.

There are two implementation details that are noteworthy. One of them is that each query has exactly 10 URLs so a sorting network can be used to speed up the sorting process in the fitness evaluation. The second is that the fitness evaluation can be trivially parallelized by computing the NDCG of a fraction of the queries on seperate threads, then averaging the results over the number of threads. No thread synchronization is required. The full implementation is provided in the file esrank.cpp in apendix B.

Figure 3.3: An example of how ES-Rank ranks



As seen in figure 3.3, ES-Rank first scores each URL with a real number. The scores are calculated as a dot product of the feature vector $\vec{x}$ and the weight vector $\vec{w}$. The URLs are then sorted according to these scores. ES-Rank mutates $\vec{w}$ until the stop criterion is met which could be a certain number of generations or an NDCG threshold.

3.5 Advanced algorithm

For the advanced algorithm, I improved the baseline ES-Rank and I also used collaborative filtering to improve the NDCG score.

3.5.1 Performance variance improvement

During the search in weight space, the algorithm tends to get stuck in local minima which is absolutely normal, but it remains in a different local minimum every run due to the random mutations. The problem is that the fitness varies a lot across the minima.

To correct this problem, I have introduced an initialization phase in ES-Rank, where the initial weight vector $\vec{w}$ is chosen, during the first few generations of the algorithm, such that the loss

$$\frac{1}{N}\sum_{j=1}^{N}|y_j - \vec{w}\cdot\vec{x}_j|, \tag{3.3}$$

is minimized. N is the number of query-URL pairs, y_j is the relevance label of the j-th pair, and $\vec{x}_j$ is the feature vector of the j-th pair. This was proposed in [9] but the author used linear regression for this initialization. In this implementation, the weights are chosen with the same mutation process as in algorithm 1. This not only reduces the NDCG variance but it also raises overall training performance of the algorithm.

3.5.2 Unsuccessful mutation improvement

In the original ES-Rank, any number of weights can be mutated. I have restricted the number of weights that can be mutated to 3 at most, that is 1, 2, or 3 weights can be mutated every generation. The idea behind this is that with an increasing number of mutated weights, there is a decrease in the probability that all of those mutations change the weights such that the overall performance increases.

I have tested this change empirically and it led to faster convergence to better solutions, i.e. the increase in training performance was larger per generation.

3.5.3 Collaborative filtering feature

Team Dataiku applied Funk's Matrix factorization on a utility matrix of users and domains but reported a marginal increase in performance. I tried to apply the same algorithm on three utility matrices (described bellow) to see if it makes any difference in performance.

The three added features were predictions from matrix factorization of three utility matrices:

- $Users \times URLs$
- $Queries \times URLs$
- $Terms \times URLs$

I have used matrix factorization in a non-traditional way. Instead of Users and URLs the utility matrix was for Queries and URLs. The idea is that there are also certain relationships between queries which could be captured with latent variables. This doesn't have much to do with collaborative filtering, but it improved the performance.

The values (ratings) in the utility matrices mentioned above were scores:

- 0 if the URL was skipped
- 1 if the relevance grade of the URL was 0
- 2 if the relevance grade of the URL was 1
- 3 if the relevance grade of the URL was 2

There is no point in considering the missed URLs because the user didn't examine them and therefore had no opinion about them.

3.5.3.1 How is the CF feature important and how does it help ES-Rank?

Here is the full weight vector of 107 weights averaged over a few runs of ES-Rank:

-0.29, -1.20, -9.85, -8.35, -0.30, 1.60, 0.30, -3.59, -2.50, 0.00, -0.40, 0.00, 7.63, 4.42, 6.28, -2.98, 0.25, -0.74, 2.42, 0.00, -0.05, -1.13, -7.49, -3.07, -2.36, 0.32, -1.82, -3.09, 1.03, 0.01, -0.27, 0.44, -2.32, -0.89, 5.87, 7.19, -4.49, 0.09, -1.29, -0.71, 0.00, 0.05, 2.05, -0.06, 0.00, 0.00, -0.03, -1.34, 0.00, 2.17, 1.89, 3.55, -2.46, -0.42, 0.73, -1.43, 0.00, 0.00, 0.01, 0.00, 0.27, 0.40, 0.55, -1.61, 0.74, 4.62, -0.71, 2.44, 7.21, -2.95, 0.00, 4.76, 5.18, -32.32, 6.06, -3.82, -4.42, -4.64, -8.72, -1.06, 0.04, 0.01, -0.22, 0.09, -0.70, 0.06, 4.48, 8.51, 48.03, 1.31, 3.23, 2.92, 0.84, 0.00, -4.52, **15.07, 893.56, 3.40**, -0.07, 0.26, -5.67, 0.12, -0.05, -0.50, -0.02, -0.45, -0.50

The features were normalized before the weights were trained by ES-Rank so the weights directly reflect how each feature is important in linear ranking. The weights 15.07, 893.56, 3.40 belong to features which are predictions of utility matrices $Users \times URLs$, $Queries \times URLs$, and $Terms \times URLs$ respectively. From the magnitude of these weights, an observation can be made that the prediction of matrix $Terms \times URLs$ didn't do much, the prediction of matrix $Queries \times URLs$ is the most explanatory, and the prediction of matrix $Users \times URLs$ is more explanatory than most of the other features.

3.5.4 Hyperparameter tuning

ES-Rank During the tuning of ES-Rank parameters, I have found that the NDCG hardly increases by a substantial amount after the 7000-th generation, so I set the number of generations to 7000. I set the number of generations that were dedicated to weight initialization (described in subsection 3.5.1) to 1500 since the error convergence stopped around that point. The optimal number of mutated weights per generation is 3 by empirical tests.

Matrix factorization I have chosen the number of latent variables that gave the smallest MSE. I set the number of latent variables to 40 which reduced the MSE from initial 0.89 to 0.68. Any setting of the learning rate above 0.02 led to gradient explosion (the MSE went to infinity).

3.6 Performance evaluation and comparison

I didn't use cross validation to measure the performance of the implemented models, because I could submit my solution to the competition page on Kaggle.com and measure the performance directly on Yandex's test sessions.

I also used Random Forest Classifier from the python library scikit-learn for a better comparison of scores.

Table 3.1: Performances of individual algorithms on Dataiku's feature set with and without collaborative filtering evaluated on Kaggle

	no CF	with CF
ES-Rank baseline	0.80141	-
ES-Rank improved	0.80149	0.80302
Random Forest	0.80263	0.80426

Table 3.2: Kaggle leaderboard (higher NDCG is better)

Place	eam name	NDCG score
1	pampampampam (ooc)	0.80724
2	Dataiku Data Science Studio	0.80714
3	LR	0.80636
4	learner	0.80475
5	DenXX	0.80425
6	YS-L	0.80390
7	lucky guy	0.80322
8	Gábor S	0.80320
9	Vermillion Team 3	0.80289
10	Ruslan Mavlyutov (ooc)	0.80191

As seen in tables 3.1 and 3.2, the performance of ES-Rank is not great compared to other submissions on the leaderboard though ES-Rank came 9th. There could be a few possible reasons for the low score of ES-Rank

1. The training performance of ES-Rank improved with CF was 0.80668 which is considerably higher than the test performance 0.80302 and indicates overfitting.

2. I inaccurately replicated the features described by Dataiku in [13] and therefore had bad quality data going into the models.

3. ES-Rank is a linear model so it is a weak learner and cannot capture non-linear relationships in the data.

Number 1 is unlikely to be true because a linear model hardly overfits data especially on large data with $1.2 \cdot 10^6$ queries. I have also tried to reduce the number of generations in ES-Rank to not overfit the data but that led to an even lower score on Kaggle.

Number 2 is very likely because Dataiku have also tried Random Forests and reached a score of 0.80458 but only with 30 trees. I reached 0.80426 but had to use 128 trees so the base features without CF are of low quality.

Number 3 is also likely because Random Forest which is a more complex predictor than ES-Rank and easily reached a higher NDCG score.

3.6.1 Collaborative filtering performance increase significance test

Here are the NDCG values of ES-Rank with and without CF measured over 10 runs each and evaluated on Kaggle.

```
nocf <- c(0.80123, 0.80137, 0.80152, 0.80132, 0.80130,
.80138, 0.80157, 0.80129, 0.80142, 0.80143)
withcf <- c(0.80249, 0.80250, 0.80266, 0.80274, 0.80238,
0.80251, 0.80302, 0.80290, 0.80260, 0.80100)
var.test(nocf, withcf)
t.test(nocf, withcf, paired=F, alternative="less",
var.equal=F, conf.level=0.999)
```

Listing 1: Measured NDCGs in R

I conducted a simple two sample t-test. The hypothesis H_0 is that the mean performance with and without CF is the same and the alternative H_A is that ES-Rank with CF significantly increases performance. As seen in listings 1 and 2, the test in R gives a p-value that is lower than 0.001 which indicates that CF increased the NDCG with 99.9% confidence.

```
F test to compare two variances

data:  nocf and withcf
F = 0.036064, num df = 9, denom df = 9, p-value = 3.279e-05
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
 0.008957842 0.145194337
sample estimates:
ratio of variances
        0.03606422


Welch Two Sample t-test

data:  nocf and withcf
t = -6.1275, df = 9.6483, p-value = 6.486e-05
alternative hypothesis: true difference in means is less than 0
99.9 percent confidence interval:
          -Inf -0.0003463374
sample estimates:
mean of x mean of y
 0.801383  0.802480
```

Listing 2: Two sample t-test output

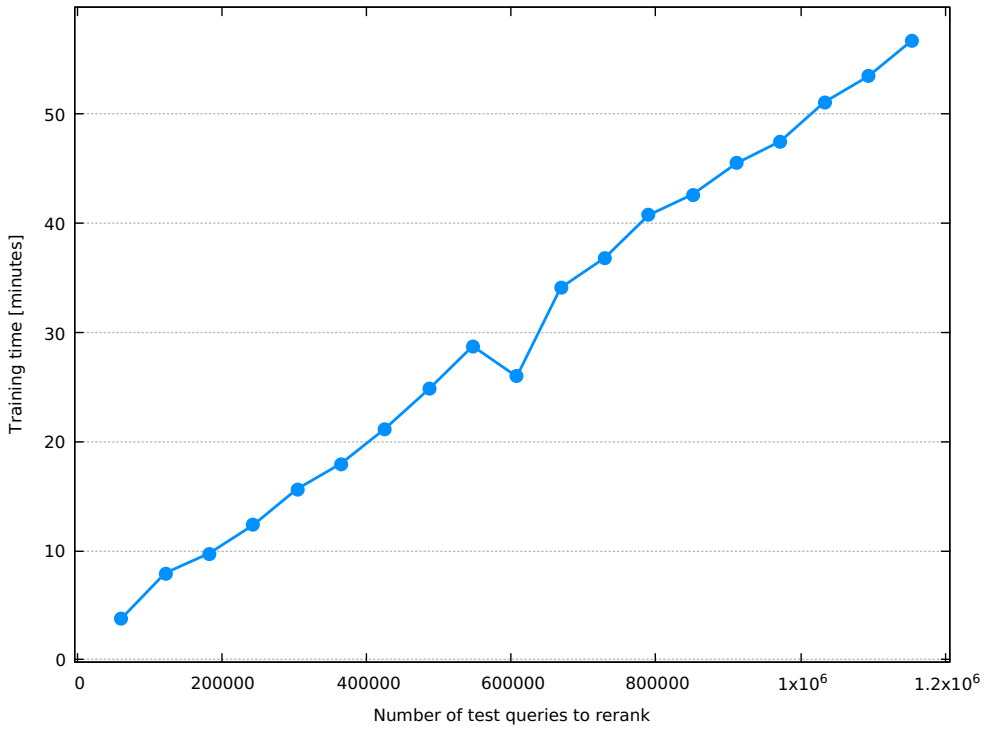
3.7 Computational complexity and scalability

3.7.1 ES-Rank

The time complexity of ES-Rank is $O(G \cdot Q \cdot (F \cdot U + Ulog(U)))$, where G is the number of generations, F is the number of features of query-URL pairs, Q is the number of queries to be ranked, and U is the maximum number of URLs in a query. The mutation of the weight vector takes $O(F)$ time, the NDCG calculation takes $O(Q \cdot (F \cdot U + Ulog(U)))$ because a dot product is performed for each URL in a query which takes $O(F \cdot U)$ time and then the URLs are sorted in $O(Ulog(U))$ time, this is done for all queries over G generations hence $O(G \cdot Q \cdot (F \cdot U + Ulog(U)))$.

For this particular dataset, it is given that each query has 10 URLs and the number of features doesn't change during ES-Rank ranking. Then the time required to rank scales linearly to the number of queries to be ranked. This is confirmed by empirical measurements plotted in figure 3.4. These measurements were made on a 8-core machine. The fitness evaluation can be computed in parallel. The number of operations per core would be $O(G \cdot \frac{Q}{t} \cdot (F \cdot U + Ulog(U)))$, where t is the number of threads (or cores).

Figure 3.4: Training time of ES-Rank in relation to number of queries

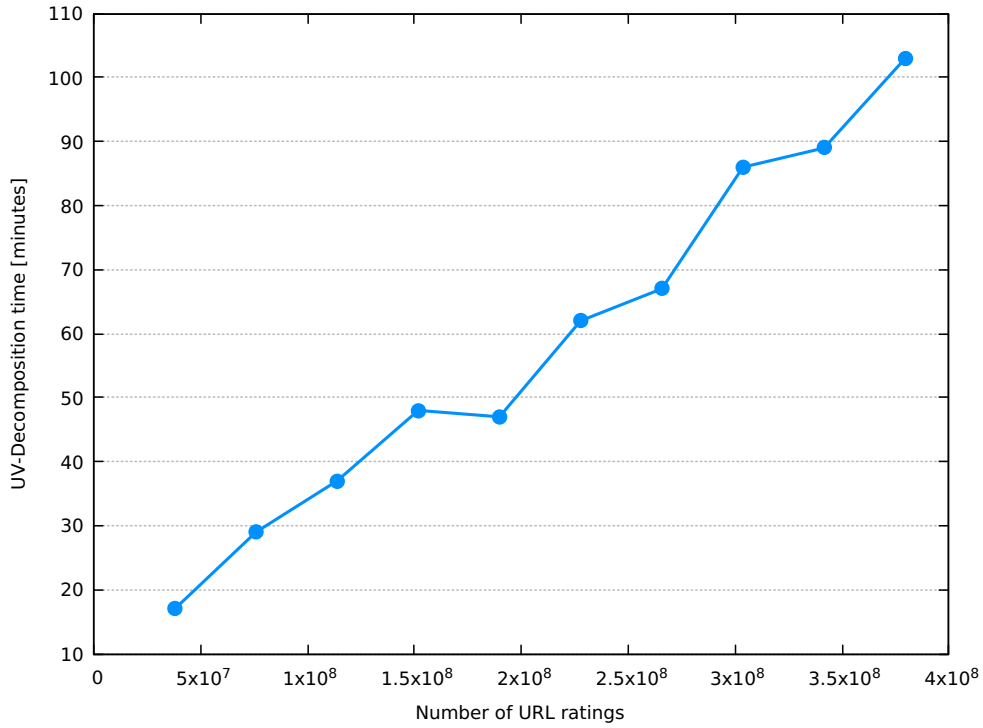


Note that $1.2 \cdot 10^6$ is the number of queries in the full test dataset (3 days of logs).

3.7.2 Matrix factorization collaborative filtering

The time complexity of matrix factorization is $O(k \cdot e \cdot R)$, where k is the number of latent variables per user or per URL, e is the number of epochs per latent variable, and R is the number of URL ratings. k and e are chosen empirically to minimize the MSE and don't change during training. Given that k and e are constants, the algorithm scales linearly with the number of URL ratings R as shown in figure 3.5.

Figure 3.5: Training time of matrix factorization in relation to the number of ratings



Conclusion

I have examined learning to rank and collaborative filtering algorithms and chose to implement ES-Rank and incremental matrix factorization. For performance evaluation, I have applied those algorithms on Yandex's dataset provided in the Personalized Web Search Challenge competition held on Kaggle.com. To calculate features from the dataset, I replicated team Dataiku's feature aggregation which was challenging because their methods were described perhaps too succinctly. Hyperparameter tuning was a challenging task as well because the runtime of the algorithms mentioned above was long on this large dataset. I have improved the baseline ES-Rank algorithm with an initialization step, a better mutation procedure, and matrix factorization features which led to an increase of NDCG score from 0.80149 to 0.80302 (the higher the better). I have also analyzed the computational complexity and scalability of the implemented algorithms and compared their performance to other submissions on Kaggle.com. The implemented algorithms placed 9th in the competition which is decent considering that they are linear models. By doing these tasks I have completed the assignment.

For future improvements, bagging could be used to further reduce the variance of ES-Rank and to improve its performance.

Bibliography

[1] Leskovec, J.; Rajaraman, A.; et al. *Mining of massive datasets*. Cambridge university press, 2014.

[2] Modrỳ, M. *Algoritmy pro učení se řadit*. Master's thesis, Czech Technical University in Prague, Czech Republic, 2014.

[3] Chapelle, O.; Metlzer, D.; 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] Breiman, L. Random forests. *Machine learning*, volume 45, no. 1, 2001: pp. 5–32.

[5] Ho, T. K. The random subspace method for constructing decision forests. *IEEE transactions on pattern analysis and machine intelligence*, volume 20, no. 8, 1998: pp. 832–844.

[6] Friedman, J. H. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, 2001: pp. 1189–1232.

[7] Burges, C. J. From ranknet to lambdarank to lambdamart: An overview. *Learning*, volume 11, no. 23-581, 2010: p. 81.

[8] Xu, J.; Li, H. Adarank: a boosting algorithm for information retrieval. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, ACM, 2007, pp. 391–398.

[9] Ibrahim, O. A. S.; Landa-Silva, D. An evolutionary strategy with machine learning for learning to rank in information retrieval. *Soft Computing*, 2018: pp. 1–15.

[10] Rajaraman, A. Collaborative filtering. 2016. Available from: https://www.youtube.com/watch?v=h9gpufJFF-0

[11] Percy, M. Collaborative Filtering for Netflix. 2009. Available from: https://classes.soe.ucsc.edu/cmps242/Fall09/proj/mpercy_svd_paper.pdf

[12] Funk, S. Netflix Update: Try This at Home. 2006. Available from: http://sifter.org/simon/journal/20061211.html

[13] Masurel, P.; Lefèvre-Hasegawa, K.; et al. Dataiku's solution to yandex's personalized web search challenge. *Dataiku's Solution to Yandex's Personalized Web Search Challenge, WSDM*, volume 14, 2014: pp. 285–294.

Appendix A

Acronyms

AP Average Precision

CF Collaborative Filtering

ES Evolutional Strategy

ERR Expected Reciprocal Rank

LTR Learning to Rank

MAP Mean Average Precision

MRR Mean Reciprocal Rating

NDCG Normalized Discounted Cumulative Gain

WTA Winner Takes All

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `implementation` ..... the directory of implementation sources codes
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format