A Comparative Study on Hyperparameter Optimization for Recommender Systems

Pawel Matuszyk
Knowledge Management and Discovery
Otto-von-Guericke University
Magdeburg, Germany
pawel.matuszyk@ovgu.de

Renê Tatua Castillo
Knowledge Management and Discovery
Otto-von-Guericke University
Magdeburg, Germany
rene.tatua@st.ovgu.de

Daniel Kottke
Knowledge Management and Discovery
Otto-von-Guericke University
Magdeburg, Germany
daniel.kottke@ovgu.de

Myra Spiliopoulou
Knowledge Management and Discovery
Otto-von-Guericke University
Magdeburg, Germany
myra@iti.cs.uni-magdeburg.de

ABSTRACT

Hyperparameter optimization is essential for researchers and practitioners as the quality of many machine learning and data mining algorithms is sensitive to parameter settings. In this comparative study we investigate the applicability of these optimization strategies to recommender systems. We compare nine different algorithms in an extensive evaluation on real-world datasets. To our knowledge, this is the first such study in the domain of recommender systems. Lastly, we give recommendations regarding the best-performing optimization algorithm.

CCS Concepts

•**Computing methodologies** → **Discrete space search; Continuous space search; Randomized search;** *Machine learning algorithms;* •**Information systems** → **Collaborative filtering;**

Keywords

Recommender Systems; Matrix Factorization, Hyperparameter Optimization

1. INTRODUCTION

Hyperparameter optimization is an indispensable tool for researchers and practitioners working with machine learning and data mining algorithms. Many of those algorithms are highly sensitive to setting of parameters, which is considered a task of a human expert. An example is the k-means algorithm that requires setting of the parameter k, i.e. the number of centroids. Setting a non-optimal number results in clustering of a bad quality. Often human experts have no means to know in advance what a good parameter setting is. Therefore, they try several settings blindly until an acceptable setting is found. Hyperparameter optimization algorithms offer a solution to this problem.

For researchers hyperparameter optimization is essential, for instance, when comparing two algorithms. Let A and B be two algorithms with different parameters that need to be tuned. After experimental evaluation we obtain the error measure H of the two algorithms: H_A and H_B. Let $H_A < H_B$. When the tuning is done manually by a human, it is unknown if the difference in the measured error is due to the algorithm A being better than B, or is it because the human expert tuned A better than B. A reliable conclusion is not possible. To marginalize the influence of the human expert, an objective and fair tuning component is required. Hyperparameter optimization methods play the role of the objective component.

Formally, we define the hyperparameter optimization task as follows. Let A be the target algorithm with n number of parameters to be tuned. Each parameter θ_i can be a value taken from an interval $[a_i, b_i]$ in parameter configuration space $\Theta = [a_1, b_1] \times \ldots \times [a_n, b_n]$. Let the vector $\overrightarrow{\theta} = [\theta_1, \theta_2, \ldots, \theta_n]$ represent a parameter configuration and $H : \Theta \to \mathbb{R}$ be a performance metric function that maps $\overrightarrow{\theta}$ to a numeric score computed over a set of instances. Therefore, the optimization problem aims to find $\overrightarrow{\theta} \in \Theta$ that minimizes $H(\overrightarrow{\theta})$. In our problem definition we partially adopted the notation proposed by Lindawati et al. [16].

In this comparative study we focus on hyperparameter optimization for a specific type of algorithms i.e. recommender systems. Recommender systems (RS) alleviate the problem of information overload. Their goal is to recommend relevant items to users in a personalized way. They are applicable not only to commercial products, but also to web sites, scientific articles, medications, etc.

As many other algorithms, RS are also sensitive to setting parameters correctly. Therefore, in this work we investigate which hyperparameter optimization method is the most appropriate for them. Our **main contribution** is a **comparison of nine optimization algorithms on four real-world, public, benchmark datasets**. As result of our research, we give clear recommendations to practitioners and researchers in the RS domain (cf. Sec. 6).

To our knowledge, this is the first such comparative study for recommender systems. While hyperparameter optimiza-

tion has been investigated for classification or regression problems, recommender systems pose additional challenges that do not occur in conventional regression or classification problems. In contrast to regression problems, the challenge in RS is often to predict a real-valued sparse matrix of ratings that express the degree of preference of users towards items. This results in a different response surface in the optimization problem than in the case of regression or classification. Consequently, optimization methods known to perform well for the conventional regression or classification do not necessarily perform well in recommender systems. Our goal is to investigate the performance of several hyperparameter optimization methods in this specific setting.

In this work, we focus on matrix factorization (MF) as recommendation algorithms. MF algorithms are one of the most successful types of algorithms in recommender systems. In the current research they are considered state-of-the-art, since they have shown their superiority in terms of predictive performance and runtime in numerous publications [20] [15]. Hence, in this work we use a generic representative of those algorithms, the BRISMF algorithm by Takács. et al. [20], which requires optimization of three parameters: learn rate η, regularization λ and the number of latent dimensions k (cf. Sec. 2 for more details). While there are numerous MF methods, most of them are based on the same principle that the BRISMF algorithm also uses. Since it is not possible to experiment with countless variants of MF algorithms, we focus on BRISMF, which is representative to most of them and, therefore, allows for generalization.

The remainder of this paper is structured as follows. Section 2 describes the related work in hyperparameter optimization techniques and recommender systems. In Section 3, we describe the implemented hyper-parameter optimization algorithms. In Section 4 we explain our evaluation protocol. Experiments and results are presented in Section 5. In Section 6 we conclude the work.

2. RELATED WORK

Two main classes of algorithms in the hyper-parameter optimization are model based (MB) and derivative-free (DF) approaches. MB approaches approximate the response surface with another function by sampling points using the current model. DF approaches make use of heuristics in order to achieve the best parameter combination.

Recently, MB approaches are gaining popularity over DF approaches due to the fact that evaluating a surrogate model instead of the response surface is computationally cheaper. Jones et al. proposed the Efficient Global Optimization algorithm (EGO), which computes the expected improvement using a combination of a linear regression model and a noise-free stochastic Gaussian process model (also known as DACE model)[13]. Many Sequential Model Based Optimization algorithms (SMBO) are based on EGO [3],[19], among others also the SMAC algorithm [11] (cf. Sec. 3.3). The main competitor of SMAC is a meta-heuristic called genetic algorithm GGA [2], which combines the power of heuristics and model-based approaches. A recent direct search approach is CALIBRA. It uses a simple local search algorithm combined with fractional factorial design [1]. Frank Hutter et al. (the authors of SMAC, which we used in our experiments, cf. Sec. 3.3) implemented another prominent local search algorithm called PARAMILS[12], and proved its superiority over CALIBRA.

There is also a wide range of stochastic optimization algorithms. However, most of them depend on the availability of a gradient (e.g. stochastic gradient descent, alternating least squares, etc.), making them inapplicable to our problem. The gradient can be, however, finely approximated by heuristic algorithms or DF approaches. Commonly used heuristics encompass genetic algorithms (Sec. 3.2), simulated annealing, particle swarm optimization.

In this work we focus on hyperparameter optimization for recommender systems (RS). RS are an active research filed that started with a simple collaborative filtering algorithm for text documents that helped users find relevant documents in a big document collection [7]. During recent years of research, matrix factorization algorithms developed into a state-of-the-art in recommender systems. They have shown their superiority in several real-life applications (e.g. the Netflix competition) and in numerous studies on this topic [20], [15]. Therefore, we use those algorithms in our study.

Matrix factorization (MF) decomposes a matrix of user ratings (e.g. 5 stars rating for relevant items) into two latent matrices. Those matrices are then used for predictions of missing values in the original matrix. Internally, MF uses stochastic gradient descent (SGD) to approximately decompose the rating matrix. Therefore, we need to optimize additional parameters that are characteristic to SGD (e.g. the learn rate η). The difficulty of setting parameters for MF has been recognized by Zeng et al. [21]. They proposed MF with scale-invariant parameters that once set, can be transferred to matrices of different size. While this makes the task of model selection easier, finding the optimal parameter values remains a challenge.

To find the optimization methods that tackles this challenge in the best way, in this paper we use the BRISMF algorithm (Biased Regularized Incremental Simultaneous MF) [20] as a representative of MF algorithms. Is uses the core idea of matrix factorization that can be found in several other papers (e.g. work by Koren et al. [15]). Due to the space constraints in this paper we cannot explain this method in details and refer to the paper by Takács et al. [20].

3. HYPERPARAMETER OPTIMIZATION ALGORITHMS

In this section, we provide descriptions of the implemented hyperparameter optimization algorithms. In the following, we assume without loss of generality that the goal of the algorithms is minimization of an error function (and not maximization of a quality function).

3.1 Random Walk

In random walk (RW), the entire search space can be seen as a (high-dimensional) grid, where each point represents a specific hyperparameter setting $\overrightarrow{\theta} \in \Theta$. The RW algorithm is an iterative method. First, it selects a random point on the grid and considers it a central grid point $\overrightarrow{\theta}_c^t$. Then, it computes the performance of each neighbour surrounding that specific grid point. Using a selection mechanism which is described later, we select one of the neighbours and make a step towards this neighbour. This neighbour then becomes the new central grid point $\overrightarrow{\theta}_c^{t+1}$. The whole process is repeated until a stopping criterion is satisfied which is either a maximum number of steps t (iterations) or a maximum number of experiments.

Data: R storage of all runs and their performances; N the current neighbours; Θ the configuration space for target algorithm A; S the max. number of steps; E the max. number of experiments
Result: Optimized parameter configuration $\overrightarrow{\theta}_{inc}$
$[\overrightarrow{\theta}_{inc}, R]$= $Initialization(\Theta)$;
while S *and* E *are not reached* **do**
  $[N]$=$GetCurrentNeighbors(\overrightarrow{\theta}_{inc}, \Theta)$;
  for $i := 1, ..., length(N)$ **do**
    $[R]$=$RunExperiment(N(i), R)$;
  end
  $[\overrightarrow{\theta}_{inc}]$= $GetCurrentGridPointbyRoulette(\overrightarrow{\theta}_{inc}, R)$;
end

Algorithm 1: Random Walk Algorithm

In each iteration of RW a selection of a neighbour is made. We apply a roulette selection algorithm. Neighbours with high fitness values have a higher chance to be selected. The fitness value is computed as follows. We have K neighbours $\overrightarrow{\theta}_1, \ldots, \overrightarrow{\theta}_K$ and their error measures $H(\overrightarrow{\theta}_1), \ldots, H(\overrightarrow{\theta}_K)$. We sort them and obtain a ranking $H_1, \ldots, H_K$, so that $H_1 \leq \ldots \leq H_i \leq \ldots \leq H_K$. Then, the fitness value f_i, assigned for the neighbour at the position i in the ranking is $f_i = \frac{1}{i \cdot H_i}$.

The error measure H_i is multiplied with its ranking i to give more priority to higher ranked settings. Otherwise, the roulette selection would be nearly equivalent to a random selection, since $H_1, ..., H_n$ are very similar to each other. Consequently, the probability p_i, for selecting a neighbour at the position i is:

$$p_i = \frac{f_i}{\sum_{i=1}^{n} f_i} \quad (1)$$

The pseudo-code for RW is shown in Alg. 1. In our algorithms we used the naming convention proposed by Hutter et al. [11].

3.2 Genetic Algorithm

Genetic Algorithms (GA) are a widely used technique for solving optimization problems by exploiting the process of natural selection present in evolution. Over the last decades, many GA have been proposed by the research community. One of the best known ones is the Simple Genetic Algorithm (SGA) by Holland [10], which lays the foundations of GA. GA aims to optimize a function by evolving a population of candidate solutions (called phenotypes). The information of each candidate solution is encoded into an array (called chromosomes) which can be mutated and altered. The symbols that form the array are called genes.

The GA evolution process is iterative. Usually the first population is generated randomly, and at each generation the fitness of each member of the population is evaluated. Only those individuals with high fitness are likely to transfer their information to the next generation. The process continues until it reaches a stopping mechanism. For instance, when a maximum number of generations is reached or by obtaining a target value within a given threshold. More details about how GA works can be found in [10]. In Alg. 2 we show the pseudo-code for the GA.

Data: R storage of all runs and their performances; P the current population; si the original population size; parameter configuration space Θ; G maximum number of generations; E maximum number of experiments
Result: Optimized parameter configuration $\overrightarrow{\theta}_{inc}$
$[P, R]$= $Initialization(\Theta)$;
while G *and* E *are not reached* **do**
  for $i := 1, ..., length(P)$ **do**
    $[R]$=$RunExperiment(P(i), R)$;
  end
  SortPopulation(P);
  TrimPopulation(si);
  $[\overrightarrow{\theta}_{inc}]$=P(0) ;
  $[P]$= $GetNextPopulation(P)$;
end

Algorithm 2: Genetic Algorithm

Data: P is a population; C is a chromosome
Result: New population P_{new}
for $i := 1, ..., length(P)$ **do**
  $P_{new}add(P(i))$;
end
for $i := 1, ..., length(P)$ **do**
  $C = GetChromosomebyIndex(i)$;
  $C_{mutated} = MutateChromosome(C)$;
  $C_{random} = GetRandomChromosome(P)$;
  $[C_{cross1}, C_{cross2}] = Crossover(C, C_{random})$;
  $P_{new}add(C_{mutated})$;
  $P_{new}add(C_{cross1})$;
  $P_{new}add(C_{cross2})$;
end
return P_{new} ;

Algorithm 3: Get Next Population

A selection mechanism is required to evaluate, who are the fittest members in the population. Since the differences in the fitness values are not high we use selection by ordering. It sorts the candidate solutions in decreasing order of fitness value. The population is then trimmed to the original population size. Therefore, only the best candidate solutions (those with high fitness value) are kept and passed to the next generation. In our experiments we used the population size of 4. The pseudo-code for this process is shown in Alg. 3. This version of genetic we call "GA-ordering" in our experiments. Additionally, we use another version "GA-roulette", where the selection is performed by a roulette algorithm.

The Recombination (also known as chromosomal crossover) is a convergence operation that is intended to pull the population towards a local minimum/maximum by combining genes. On the other hand, mutation is a divergence operation that acts as a source of diversity. The GA requires a trade off between exploitation and exploration similarly to other optimization algorithms.

In our application a chromosome is the parameter vector $\overrightarrow{\theta}$ (cf. the definition in Sec. 1). Mutation in the chromosome $\overrightarrow{\theta}$ is equivalent to changing a parameter value at a random position m from this vector into a random value θ_m^{random} from the predefined interval $[a_m, b_m]$. The resulting, mutated chromosome is $\overrightarrow{\theta} = [\theta_1, \ldots, \theta_m^{random}, \ldots, \theta_n]$.

Crossover is a binary operation that swaps parts of two chromosomes with each other. Let $\overrightarrow{\theta}_y$ and $\overrightarrow{\theta}_z$ be chromosomes of length n. A result of crossover on $\overrightarrow{\theta}_y$ and $\overrightarrow{\theta}_z$ look as follows: $\overrightarrow{\theta}_y = [\theta_{y_1}, \ldots, \theta_{y_m}, \theta_{z_{m+1}}, \ldots, \theta_{z_n}]$ and $\overrightarrow{\theta}_z = [\theta_{z_1}, \ldots, \theta_{z_m}, \theta_{y_{m+1}}, \ldots, \theta_{y_n}]$, where m is a random number from $[0, \ldots, n]$.

3.3 Sequential Model-based Algorithm Configuration

Sequential Model-based Algorithm Configuration (SMAC) is a state-of-the-art, model-based algorithm using Sequential Model-Based Optimization (SMBO). SMBO tackles optimization problems by using four components: a initialization mechanism, a surrogate model, a selection mechanism, and a intensification phase. As described in [11], the first step in SMBO algorithm is the "initialization mechanism" which aims to find the best initial configuration $\overrightarrow{\theta}_{inc}$. Random or default selection is commonly used in this step. In the next step a model M is fitted in order to characterize the response surface H. M uses as training data all target algorithm runs performed so far. In other words, a training set of K instances can be seen as $\{(\overrightarrow{\theta}_1, H(\overrightarrow{\theta}_1)), \ldots, (\overrightarrow{\theta}_K, H(\overrightarrow{\theta}_K))\}$ where a parameter configuration is $\overrightarrow{\theta}_i$ and $H(\overrightarrow{\theta}_i)$ is the corresponding target algorithm's observed performance. The selection mechanism's main goal is to select the most promising parameter configurations based on M and the configuration's expected improvement EI. $EI(\overrightarrow{\theta})$ tells us how promising a configuration $\overrightarrow{\theta}$ could be by giving a trade off between exploration and exploitation. More detail information about the calculation of expected improvement can be found in [13]. Finally, the intensification mechanism compares a list of promising configurations P against the current incumbent $\overrightarrow{\theta}_{inc}$ in order to select the new incumbent for the next iteration. The SMBO algorithm is shown in Alg. 4. This pseudo-code comes from [11] and was minimally adjusted to our problem definition.

Data: R storage of all runs and their performances; M a model; P list of promising configurations, and t_{fit} and t_{select} are the run times required to fit the model and select configurations; c the cost metric.
Result: Optimized parameter configuration $\overrightarrow{\theta}_{inc}$
$[\overrightarrow{\theta}_{inc}, R] = Initialize(\Theta)$;
while *total time for configuration not exhausted* **do**
  $[M, t_{fit}] = FitModel(R)$;
  $[P, t_{select}] = SelectConfigurations(M, \overrightarrow{\theta}_{inc}, \Theta)$;
  $[R, \overrightarrow{\theta}_{inc}] = Intensify(P, \overrightarrow{\theta}_{inc}, M, R, t_{fit}, t_{select}, c)$;
end
$return(\overrightarrow{\theta}_{inc})$;
Algorithm 4: Sequential Model Based Algorithm [11]

Internally, SMAC models are based on Random Forests (RF) [5], a machine learning technique for classification and regression tasks. RF models are, in this case, an ensemble of Regression Trees (RT). SMAC uses RF models to compute EI and implements its own simple multi-start local search for finding configuration $\overrightarrow{\theta}$ with large $EI(\overrightarrow{\theta})$. In [11], SMAC is explained in more detail. In our experiments we used the SMAC software developed by Hutter et al.[1].

3.4 Full Enumeration

Full enumeration (also known as **grid search**) attempts to calculate all possible parameter settings. It works only with discrete parameters, since the number of possible combinations with continuous parameters is infinite. Therefore, it often requires a discretization of parameters. Even then, it is highly inefficient, because the number of possible combinations grows combinatorially with the number of parameters.

This method iterates over the whole multi-dimensional grid starting from a randomly selected dimension. For the sake of feasibility and of a fair comparison to other methods, we cap this process after 50 experiments (cf. Sec. 4). This capping is performed equally with all tested methods.

This method serves to us as a comparison baseline that simulates behaviour of a user, who starts an optimization, intending to test all combinations, but interrupts the process due to the excessive computation time and uses the best result found so far.

3.5 Random Search

We use two versions of random search. Random search with discretized parameters is equivalent to the random grid search. In our experiment we call it "Random discrete". The second version works in the continuous search space and is, therefore, called "Random continuous" hereafter.

Despite its simplicity the random method has several advantages. It is highly parallelizable, in contrast to sequential methods, such as SMAC. This feature of random search is particularly useful when a computational cluster or multicore processors are available.

Furthermore, it is possible to specify theoretical guarantees on the goodness of the found optimum, as Bergstra and Bengio did [4]. They define a hyper-cube around the optimal point in the search space. Let v be volume of the hypercube and V volume of the search space. Then the likelihood of finding a point in this hypercube after T trials is equal to:

$$1 - (1 - \frac{v}{V})^T \tag{2}$$

If we define $\frac{v}{V}$ as 5%, then the likelihood of finding a setting within this hypercube after 50 trials amounts to 0.9231.

3.6 Greedy Search

Greedy search optimizes only one dimension/parameter at a time, while keeping other dimensions fixed. To optimize one dimension, it sets the value of the corresponding parameter to a random value. For each parameter it selects m random samples. m is determined according to a heuristic rule $m = \sqrt{E/n}$, where E is the maximal number of experiments (so called budget) and n is the dimensionality of the search space. Then, the value of the parameter is fixated to the best setting from the random samples and the procedure is repeated with all remaining parameters. A pseudo-code explaining this procedure in detail in shown in Alg. 5.

3.7 Simulated Annealing

Simulated Annealing (SA) is a method motivated by the cooling processes observed in metals. SA uses temperature as a control variable that determines the probability of acceptance of worse solutions than the current one. As temper-

[1] http://www.cs.ubc.ca/labs/beta/Projects/SMAC/

Data: R storage of all runs and their performances; E the maximum number of experiments
Result: Optimized parameter configuration $\overrightarrow{\theta}_{inc}$
$[\overrightarrow{\theta}_{inc}, R] = RandomInitialization(\Theta)$;
while *E is not reached* **do**
  for $\forall \theta_i \in \overrightarrow{\theta}_{inc}$ **do**
    for $i := 1, ..., \sqrt{\frac{E}{|\overrightarrow{\theta}_{inc}|}}$ **do**
      $\theta_i^{new} = random \sim \mathcal{U}\{a_i, b_i\}$;
      $\overrightarrow{\theta}_{inc} = [\theta_1, ..., \theta_i^{new}, ..., \theta_n]$;
      $[R] = RunExperiment(\overrightarrow{\theta}_{inc}, R)$;
    end
    $\theta_i = R.\text{getBest}\,\overrightarrow{\theta}.\text{get}(\theta_i)$;
  end
end

Algorithm 5: Greedy search algorithm

ature goes down, the probability of accepting worse solutions also decreases.

SA systematically compares the current incumbent (parameter setting) with random configurations (so called neighbours). Then, a difference Δ in the error measure between the incumbent and each neighbour is computed. The probability of accepting a neighbour as new incumbent depends on the current temperature and Δ. It is calculated using the formula $e^{-\left(\frac{\Delta}{Temp}\right)}$. Finally, the temperature is updated using a cooling schedule (e.g. linear or geometric). The algorithm stops when it reaches a maximum number of experiments or when a temperature threshold is reached. The pseudo-code in Alg. 6 describes SA.

In our experiments we use two versions of SA with a different definition of a neighbourhood. The first version "SA-discrete" uses a discrete grid and considers adjacent nodes in the grid neighbours. The second version "SA-Gaussian" uses a Gaussian distribution with mean equal to current parameter value and standard deviation adjusted so that 98% of points lay in the range of parameter values. According to this variant of SA, a neighbouring value is a sample from this distribution. For more details and the initial publication on SA we refer to [14].

3.8 Nelder-Mead

This algorithm, also called downhill simplex method, minimizes a function by spanning a simplex in the parameter space. This simplex has $k+1$ vertices, where k is the number of dimensions of the parameter space. In two dimensional space, for instance, the simplex is a triangle.

The position of the vertices of the first simplex is determined randomly. After that, the simplex is transformed iteratively by using a set of operations: reflection, contraction and expansion [17]. For every new vertex determined by those operations the function value is calculated. Based on its value, further transformations are carried out iteratively to minimize the values associated with vertices.

An important feature of the Nelder-Mead algorithm is that is does not require the function to be differentiable, in contrast to e.g. gradient descent. Its transformations are based solely on the function values evaluated at the vertices of the simplex. Due to space constraints in this paper, we refer to the work by Nelder and Mead for details about those

Data: R storage of all runs and their performances; E the maximum number of experiments; T_{ini} the initial temperature; T_{min} the minimal temperature; N current neighbours; T_{rate} temperature cooling rate.
Result: Optimized parameter configuration $\overrightarrow{\theta}_{inc}$
$[\overrightarrow{\theta}_{inc}, R] = RandomInitialization(\Theta)$;
$Temp = T_{ini}$;
while *E is not reached and* $Temp > T_{min}$ **do**
  $[N] = GetCurrentNeighbors(\Theta)$;
  for $i := 1, ..., lenght(N)$ **do**
    $[R] = RunExperiment(N(i))$;
    $\Delta = R_{N(i)} - R_{\overrightarrow{\theta}_{inc}}$;
    $P = e^{-\left(\frac{\Delta}{Temp}\right)}$;
    if $P \geq random \sim \mathcal{U}\{a_i, b_i\}$ **then**
      $\overrightarrow{\theta}_{inc} = N(i)$;
    end
  end
  $Temp = Temp * T_{rate}$;
  $return(\overrightarrow{\theta}_{inc})$;
end

Algorithm 6: Simulated annealing algorithm

transformations [17].

3.9 Particle Swarm Optimization

Particle swarm optimization (PSO) is an optimization method proposed originally by Eberhart and Kennedy [6]. It is inspired by behaviours of swarms in nature. Accordingly, this optimization methods maintains a population of particles (parameter settings in our case) called swarm.

First, the positions of the particles in the swarm are initialized randomly. Each particle maintains additionally its velocity vector and its best position. The entire swarm also stores its best position. Subsequently, guided by the optimum of the swarm and their local optima, each particle in the swarm adjusts its position using the previous position and its velocity. The velocity is derived from the distance of a particle from its optimum and from swarm's optimum. Additionally, updating the velocity contains a random component.

4. EVALUATION FRAMEWORK

Our evaluation framework provides a fair and realistic comparison of different hyperparameter optimization methods for recommender systems. This includes that:

- the hyperparameter optimizer cannot access information from a hold-out evaluation set (it provides a parameter configuration using only the given training data),
- we repeat the experiments several times to exclude random effects,
- in every repetition each algorithm uses the same data.

This evaluation framework is given in Alg. 7. As input, it uses a list of different hyperparameter optimization algorithms to be evaluated, a dataset, a recommender system algorithm, and the number of repetitions of experiments.

Input: $\mathcal{HO}$ (list of hyperparameter optimizer)
X (dataset)
RS (recommender system)
M (number of repetitions, e.g., 100)
N (number of experiments, e.g., 50)
for $i \in \{1, \ldots, M\}$ **do**
  $res_* \leftarrow \{\ \}$;
  for $j \in \{1, \ldots, N\}$ **do**
    $(X_{RSTr}, X_{RSTe}, X_{eval}) \leftarrow \text{randSplit}(X)$;
    for $HO \in \mathcal{HO}$ **do**
      `/* Training phase */`
      $\theta \leftarrow \text{nextParam}(HO, res_{HO})$;
      $rs_{tr} \leftarrow \text{trainRS}(RS, \theta, X_{RSTr})$;
      $perf_{tr} \leftarrow \text{evalPerf}(rs_{tr}, X_{RSTe})$;
      $res_{HO} \leftarrow res_{HO} \cup \{(\theta, perf_{tr})\}$;
      `/* Evaluation phase */`
      $\theta^* \leftarrow \text{getBestParam}(res_{HO})$;
      $rs_{ev} \leftarrow \text{trainRS}(RS, \theta^*, X_{RSTr} \cup X_{RSTe})$;
      $perf_{ev}^{(i,HO,j)} \leftarrow \text{evalPerf}(rs_{ev}, X_{eval})$
    end
  end
end

Algorithm 7: Evaluation Framework

The outer loop repeats the whole experiment a given number of times (in this paper: $M = 100$). To decide on the best parameter setting, each optimizer has to store all tested parameter configurations and the corresponding performances. This is done in res_* which is initialized with an empty set ($*$ indicates that all cells are initialized accordingly). The optimizer is allowed to conduct a given number of experiments (in this paper: $N = 50$) with different parameters.

In each experiment, we split the dataset into three non-overlapping subsets (40%, 27%, 33%). The first two of them (X_{RSTr}, X_{RSTe}) are training sets for the hyperparameter optimizer. More specific, the first one is used to train a recommender system with a chosen parameter setting and the second one to test the recommender. The evaluation set is not accessible to the optimizer to ensure unbiased results.

After splitting the dataset, each hyperparameter optimizer HO selects its next parameter setting θ. The recommender RS is trained on the training set X_{RSTr} with the chosen parameters, which are then evaluated on X_{RSTe} to obtain a performance score. Each parameter-performance-tuple $(\theta, perf_{tr})$ is added to the optimizer's result set res_{HO}.

In the evaluation phase, we determine the best parameter setting based on the calculated parameter-performance-tuples, i.e. the one with the best training performance. Using this parameters θ^*, we train a recommender on all available training data $(X_{RSTr} \cup X_{RSTe})$, and evaluate it on X_{eval} to obtain the evaluation performance.

To evaluate the quality of a recommender system, we use the RMSE (Root Mean Squared Error), which is an error measure used commonly in the literature. Formally, it is calculated as follows [18]:

$$RMSE = \frac{1}{|T|} \sum_{r_{u,i} \in T} (r_{u,i} - \widehat{r}_{u,i})^2 \qquad (3)$$

where T is the test dataset, $r_{u,i}$ is a rating of user u to item i (the true, observed value) and $\widehat{r}_{u,i}$ is the estimation of the rating performed by the matrix factorization. Since RMSE is an error measure, our goal is its minimization.

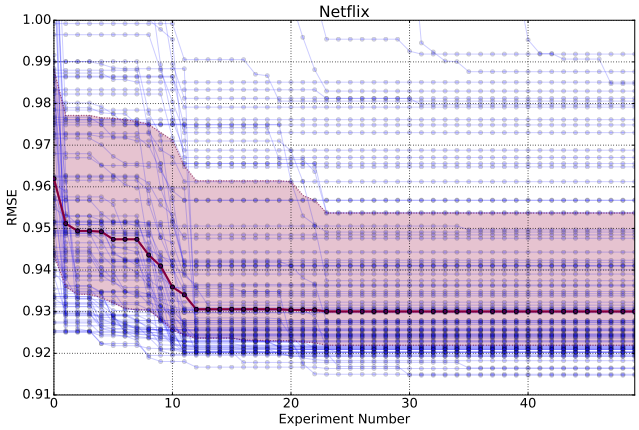

Figure 1: An example of learning curves of the greedy algorithm on the Netflix dataset. To prevent random effects, every algorithm was run 100 times (blue curves). For comparison, we aggregate them into a median learning curve (in red) and indicate the 25th and 75th percentile by red shaded area.

Since every optimization method can request RMSE values for a sequence of 50 parameter settings, the result is a learning curve that shows, how this method reduces the RMSE with every further experiment performed. Example of such learning curves are shown in Fig. 1 (any blue curve). This figure shows the evaluation RMSE score of each repetition (blue) of the greedy algorithm w.r.t. the experiment number on the Netflix dataset. Since comparing 100 learning curves of one method to sets of curves of different method would be impossible, we aggregate all those 100 curves into one median curve and plot its 25th and 75th percentiles using the shaded area. It means that half of all curves lies in the shaded area (cf. the red curve in Fig. 1). This enables a simple comparison of many methods. Consequently, all curves in the next section are the median curves.

5. EXPERIMENTS

To evaluate different hyperparameter optimization methods, we use four real-world datasets from the RS domain. The datasets are MovieLens 1M and 100k[2] [9], a sample of 2000 users from the Netflix dataset[3] and 2000 random users from the Flixster dataset[4]. A description of these datasets is given in Table 1. Since we performed more than 160,000 experiments in our evaluation, we took a random sample of 2000 users from the big datasets (i.e. Netflix and Flixter). Without sampling a comparative study of this size would be not feasible due to a long computation time. For our computations, we used a cluster running the (Neuro)Debian operating system [8].

For our experiments, we define a search space with three dimensions. The first dimension is k from the matrix factorization algorithm. k is the number of latent dimensions and it is an integer number in the range $[10, 200]$. The second

[2]http://www.movielens.org
[3]https://www.netflix.com
[4]https://www.flixster.com

Dataset	Ratings	Users	Items	Sparsity
ML1M	1,000,209	6,040	3,706	95.53%
ML100k	100,000	943	1,682	93.7%
Flixster (2k)	101,106	2,000	8,419	99.4%
Netflix (2k)	427,223	2,000	13,588	98.43%

Table 1: Summary of datasets

Method	Parameters
GA	pop. size = 4; mutation perc. = 0.5
PSO	swarm size = 5; influence by local opt. = 1; influence by global opt. = 3;
Nelder-Mead	reflection coeff.=1; expansion coeff.=2; contraction coeff.=0.5; shrinking coeff.=0.5
SA	cooling schedule=geometric; cooling rate=0.8; iter. for equilibrium=10; init. temp.=100

Table 2: Parameters of hyperoptimization methods

parameter is η, which is a learn rate used by the stochastic gradient descent in the process of factorizing a rating matrix. For η we define a range of $[0.001, 0.1]$. The last parameter is λ. It is a regularization parameter used also by the gradient descent to prevent overly high latent factors in the matrix factorization. Its range is also set to $[0.001, 0.1]$.

Some of the hyperparameter optimization algorithms require discrete parameters. One of them is e.g. the full enumeration. Using this method in a continuous search space is not possible because of the infinite number of parameter values. Therefore, we dicretize the parameters for those methods by determining 20 different, equidistant values from the aforementioned range. For λ this results in the following set of possible values: $\lambda \in \{0.0010, 0.0062, ..., 0.1\}$.

The same discretization procedure is also applied to the remaining parameters. According to this discrete definition of parameters, the entire search space encompasses 8000 possible parameter combinations. Every optimization method used in our experiments is allowed to request computation of maximally 50 parameter combinations.

Since some hyperoptimization methods are also parametric, we specify the used parameters in Tab. 2. Ideally, they should also be optimized, however, it is not feasible to also optimize the parameters of the optimizers.

In Figure 2, we present the median learning curves (cf. Sec. 4) on a sample of 2000 random users from the Netflix dataset. Every learning curve represents a median of cumulative minimum achieved over K experiments, where K is on the horizontal axis. The curves visualize the results of all methods described in Sec. 3 except for the full enumeration. The results of this method are considerably worse than of the rest (full enumeration was capped after 50 experiments same as all other methods). Therefore, plotting its curve would make the plot unreadable.

On the Netflix dataset only the random walk algorithm and the full enumeration worked considerably worse than other methods (lower values are better). Other optimization methods converged to a similar level as random methods. The Nelder-Mead algorithm performed the best most of the time, but only with a slightly better results than random methods or simulated annealing. A high degree of overlapping in the shaded areas suggests that the differences between the algorithms are not substantial.

Fig. 3 represents the learning curves on the ML1M dataset. The random walk and the full enumeration (not plotted due to high error) were outperformed by the random methods.

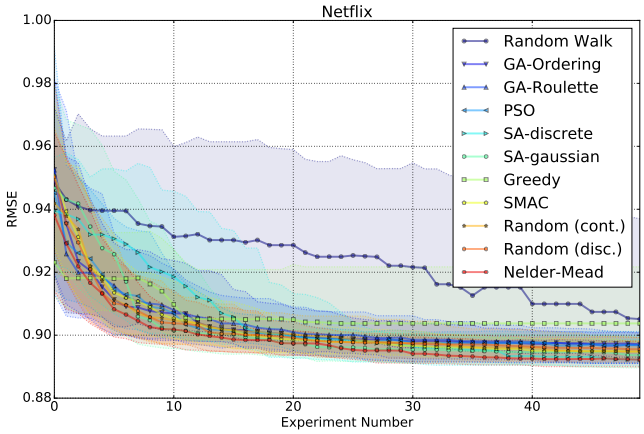

Figure 2: Median learning curves on a random sample of 2000 users from the Netflix dataset (lower results are better).

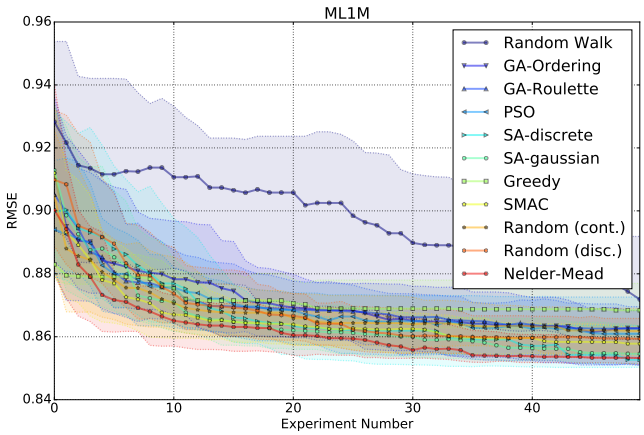

Figure 3: Median learning curves on ML1M

Also on this dataset the Nelder-Mead algorithm dominated other algorithms with only a marginal improvement as compared to random sampling. At the end of the optimization, simulated annealing with Gaussian neighbourhood reached nearly the same result as the Nelder-Mead algorithm.

We observe similar results on a sample from the Flixster dataset (cf. Fig. 4). Here, also the Nelder-Mead algorithm achieves the best result by converging to a similar level as the random algorithms.

Results on the ML100k dataset in Fig. 5 are consistent with previous observations. The Nelder-Mead algorithm performs the best. Random walk and the greedy algorithm are considerably worse.

6. CONCLUSIONS

In this study we compared nine hyperparameter optimization algorithms. To our knowledge, this is the first such study in the domain of recommender systems. We performed more than 160,000 experiments on four real-world datasets.

From our experiments, we conclude that random walk, greedy algorithm and full enumeration (also known as grid search) are certainly not recommendable. Those algorithms were outperformed by the random search on all datasets or,

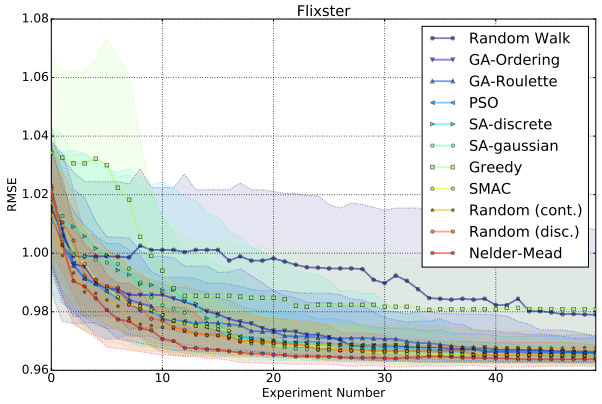

Figure 4: Median learning curves on a random sample of 2000 users from the Flixster dataset.

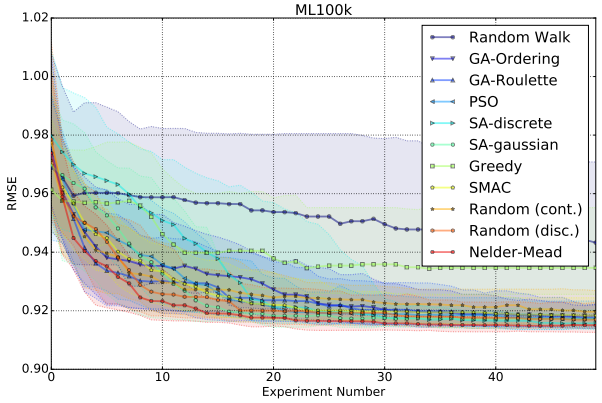

Figure 5: Median learning curves on the ML100k dataset.

in the best case, converged to the level of random search.

SMAC, PSO and genetic algorithms performed similarly to the random search. The best results were achieved by the Nelder-Mead algorithm on all datasets, followed by simulated annealing. Their improvement compared to the random search was, however, only marginal.

Considering many advantages of random search, such as full parallelization, simplicity, constant and nearly negligible computation time, we clearly recommend the random search for optimizing hyperparameters in the domain of recommender systems. Only in application scenarios, where marginal improvements play an important role, the parameters are numerical and paralleization is not necessary, we recommend the application of the Nelder-Mead algorithm or of simulated annealing.

In our future work, we intend to extend this study by more optimization algorithms. Furthermore, we plan to investigate further evaluation metrics, since they form a different response surface than the one used in this study.

Acknowledgements

The authors would like to thank Ananth Murthy, Elson Serrao, Ajay Jason Andrade and Prashanth Siddagangaiah for the implementation of selected algorithms. We also thank the Institute of Psychology II at the University of Magdeburg for making their cluster available for our experiments.

7. REFERENCES

[1] Belarmino Adenso-Díaz and Manuel Laguna. Fine-Tuning of Algorithms Using Fractional Experimental Designs and Local Search. *OR*, 2006.

[2] Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms. LNCS, 2009.

[3] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for Hyper-Parameter Optimization. In *NIPS*, 2011.

[4] James Bergstra and Yoshua Bengio. Random Search for Hyper-Parameter Optimization. *JMLR*, 2012.

[5] Leo Breiman. Random Forests. *Machine Learning*, 45:5–32, 2001.

[6] R C Eberhart and J Kennedy. A new optimizer using particle swarm theory. *International Symposium on Micro Machine and Human Science*, 1995.

[7] David Goldberg, David Nichols, Brian M. Oki, and Douglas Terry. Using Collaborative Filtering to Weave an Information Tapestry. *Commun. ACM*, 1992.

[8] Yaroslav O. Halchenko and Michael Hanke. Open is Not Enough. Let's Take the Next Step: An Integrated, Community-Driven Computing Platform for Neuroscience. *Front. Neuroinform.*, 2012.

[9] F. Maxwell Harper and Joseph A. Konstan. The MovieLens Datasets: History and Context. *TiiS*, 5(4):19, 2016.

[10] J. H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 1993.

[11] Frank Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In *LION*, LNCS, 2011.

[12] Frank Hutter, Thomas Stützle, Kevin Leyton-Brown, and Holger H. Hoos. ParamILS: An Automatic Algorithm Configuration Framework. *CoRR*, 2014.

[13] Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient Global Optimization of Expensive Black-Box Functions. *J. Global Optimization*, 1998.

[14] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 1983.

[15] Y. Koren, R. Bell, and C. Volinsky. Matrix Factorization Techniques for Recommender Systems. *Computer*, 42(8):30–37, August 2009.

[16] Lindawati, Hoong Chuin Lau, and David Lo. Instance-Based Parameter Tuning via Search Trajectory Similarity Clustering. In *LION*, 2011.

[17] J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7, 1965.

[18] Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor, editors. *Recommender Systems Handbook*. Springer, 2011.

[19] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical Bayesian Optimization of Machine Learning Algorithms. In *NIPS*, 2012.

[20] Gábor Takács, István Pilászy, Bottyán Németh, and Domonkos Tikk. Scalable Collaborative Filtering Approaches for Large Recommender Systems. *J. Mach. Learn. Res.*, 10, 2009.

[21] Guangxiang Zeng, Hengshu Zhu, Qi Liu, Ping Luo, Enhong Chen, and Tong Zhang. Matrix Factorization with Scale-Invariant Parameters. In *IJCAI*, 2015.