CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF MASTER'S THESIS

Title:	Recommendation algorithms optimization
Student:	Bc. Jakub Drdák
Supervisor:	Ing. Pavel Kordík, Ph.D.
Study Programme:	Informatics
Study Branch:	Knowledge Engineering
Department:	Department of Theoretical Computer Science
Validity:	Until the end of winter semester 2018/19

Instructions

Survey modern scalable recommendation algorithms. Benchmark selected algorithms on 2-3 datasets and identify parameters suitable for optimization. Design and implement an optimization procedure capable of fine-tuning these parameters according to given evaluation measures. Show that your algorithm is able to optimize recommendation algorithms for new datasets.

References

Will be provided by the supervisor.

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

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague June 11, 2017

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Recommendation Algorithms Optimization

Bc. Jakub Drdák

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

February 16, 2018

Acknowledgements

I would like to thank company Recombee for borrowing hardware for testing and my supervisor Ing. Pavel Kordík, Ph.D. for consultations and help during the whole study.

Declaration

Citation of this thesis

Drdák, Jakub. *Recommendation Algorithms Optimization.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

V posledních letech se vyvinulo velké množství rozličných doporučovacích algoritmů. Jednu věc mají ale všechny společnou. Jejich hyper-parametry se musí pečlivě zvolit, aby dosahovaly dobrých výsledků.

Tato práce se zabývá výběrem takových algoritmů a navrhnutím optimalizační procedury, která bude schopná nalézt vhodné hyper-parametry těchto algoritmů. Výsledky jsou pak ověřeny na reálných datasetech.

Klíčová slova Rekomendační systémy, hluboké učení, hyper-parametrická optimalizace

Abstract

Various recommendation algorithms have been proposed in recent years. However, each of them has one thing in common. It is essential to tune their hyper-parameters in order to achieve good results.

This work has focused on selecting modern and scalable algorithms. The aim has been to design and implement an optimization procedure capable of fine-tuning their hyper-parameters and evaluate the results on real-world datasets.

Keywords Recommendation systems, deep learning, hyper-parameter optimization

Contents

List of Figures

Introduction

Motivation And Objectives

Recommender systems are literally everywhere around us. They recommend to us how to spend free time, which movie to watch, book to read, what to buy, even which job to choose. Recommendation systems are an essential part of many areas. It is a critical tool to promote sales and services for many online websites and mobile applications.

However, not all companies have resources to develop their own recommender system. In many cases, it can be a preferable way to use ready-to-use tools. The company Recommbee offers one of these tools. Recombee provides an intuitive RESTful recommendation API, which is used by various international internet companies for recommendation to their users. This can lead to an interesting problem. Each company has a different business model, collects different data, offers different products and so on. Such data are a content of Recommbee database.

Therefore, the first goal of this thesis is to select modern and scalable recommendation algorithms, which are able to exploit data and offer an accurate recommendation. Further, to design an optimization procedure capable of fine-tuning hyper-parameters selected algorithms and validate results on several real-world datasets.

Problem Definition

The goal is to select modern and scalable recommendation algorithms, design fine-tuning procedure capable to tune hyper-parameters of selected algorithms and this procedure validate on several datasets.

Organisation Of The Thesis

The thesis is structured as follows: Chapter 1 presents a recommender systems overview and survey of selected algorithms together with a brief overview of hyper-parameter optimization techniques. Chapter 2 contains an analysis of chosen algorithms and design of fine-tuning procedure. Following Chapter 3 describes an algorithms implementation and the last Chapter 4 covers experiments.

Chapter 1

Survey

In the survey part we will first look at recommender systems, show their brief overview and describe basic types. Then we will focus on a subset of algorithms, which are suitable for this work. Each selected algorithm will be briefly described as discussed.

Next, we will move focus on hyper-parameter optimization, one of the essential parts of machine learning, although it sometimes suffers from a lack of care. This part will be covered only very briefly, due to an enormously wide range, which is beyond the scope of this work.

1.1 Recommender Systems Overview

Recommender systems are techniques and tools providing a recommendation to users. We often speak about a recommendation of items to a user, where an item is a general term used to denote what the system recommends to users. The goal of Recommender System is providing useful and practical suggestions, often personalized toward user's preferences or taste [1].

1.1.1 Recommender System Classification

We can usually categorize recommendation algorithms into several types based on techniques, that produce a recommendation. According to [2], we will classify Recommender Systems based on technique, that produce recommendation. For clarity, the Figure 1.1 visualise the classification.

1.1.1.1 Personalized Recommendation System

This type of system leverages user's past behavioral and based on it recommend desired items. Further, one can divide personalized systems into the following subcategories.

1.1.1.1.1 Content-based Filtering Content-based system is based on the analysis of the content of the items. It is intended to recommend items with a content similar to items, which the user enjoyed in the past or is looking at in the present. These type of recommenders are often based on creating a user profile, which stores user's preferences, taste, and features of items [1].

1.1.1.1.2 Collaborative Filtering Collaborating filtering approaches can be divided into two types: user-based CF and item-based CF. Both approaches are based on social interactions. The advantage of these types is that they do not need to extract any feature from items. Furthermore, they are able to recommend any items, even items with content, which does not correspond to any of the previous items that the user liked. User-based CF suggests recommendation based on considering users having similar interest. Unlike the user-based collaborative filtering, the item-based CF looks for items similar to items user already rated. The crucial part of the algorithm is how the similarity is computed. From the collaborative point of view, two items are similar if the users agree about ratings [1].

Generally, CF-based models cannot deal with a new user or an item, because they require a history of ratings of the user or the item to calculate the similarities, for the determination of the neighborhood. This issue is called cold start problem [3].

1.1.1.1.3 Knowledge-based This type of system is used in specific domains, where the interaction history is very sparse or does not play a significant role. In this type of system, the algorithm considers the knowledge about the item and its features, user taste (asked explicitly), and various recommendation criteria before providing the recommendation [2].

1.1.1.1.4 Hybrid-based Hybrid recommender systems combine two or more recommendation techniques to obtain better performance and mitigate drawbacks of any individual technique [4].

1.1.1.1.5 Demographic This technique is a recommendation based on the knowledge of demographic data about the user, such as age, gender, employment status, location and so on. The recommendation exploits demographic similarities among users [2].

1.1.1.2 Non-Personalized

Non-Personalized recommender system does not incorporate the personal preferences of the user. All recommendations are identical, regardless on the user [5].

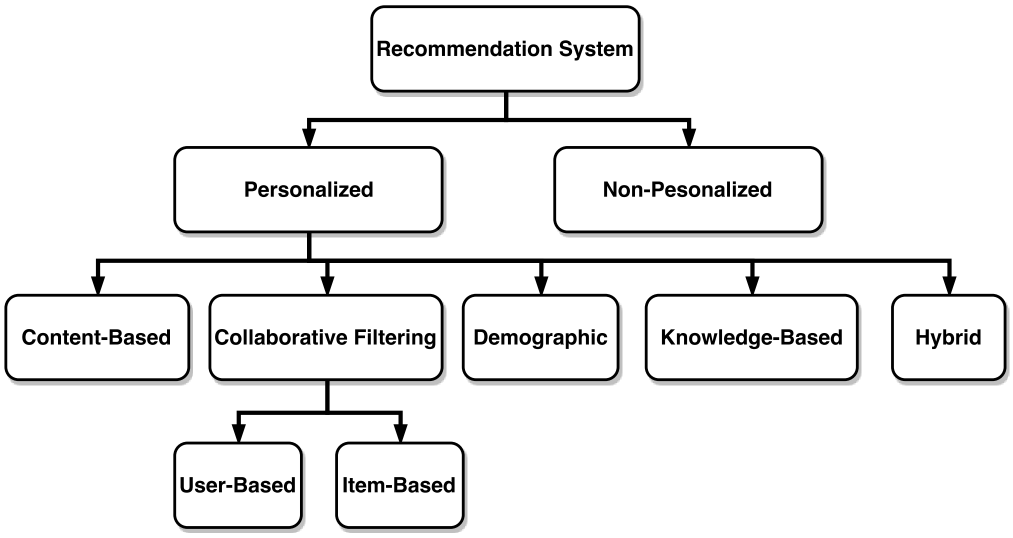

Figure 1.1: Recommender System Classification

1.2 Recommendation Algorithms

This section covers several recommendation algorithms. The first part is dedicated to traditional recommendation algorithms and the second part introduces models that incorporate deep learning techniques.

1.2.1 Traditional Recommendation Algorithms

This section cover two types of algorithms. One is memory-based algorithm, which tries to identifying the neighbors of user or item and the remaining two are model-based algorithm, which belongs to family of latent factors models.

1.2.1.1 Memory-based Collaborative Filtering Techniques

Memory-based CF algorithms use the entire set or a sample of the user-item interactions. It is assumed, that each user is part of a group of people with similar taste. By identifying the neighbors of the user, a prediction of preferences on new items is served. The most common representative are neighborhood-based CF algorithms. This type of CF algorithm uses the following steps [6]:

- calculate the similarity or weight $w_{i,j}$ between two users or two items,
- serve a suggestion for the active user by taking the weighted average of all the ratings of the user or item on a certain item or user.

1.2.1.1.1 Neighborhood-based Algorithm

Similarity computation between items or users is a critical step. In case of item-based CF algorithm, the basic idea is to compute the similarity of all pairs of items. A similarity or weight $w_{i,j}$ between two items is calculated by taking the ratings of the users who have rated both of the items and then applying a similarity measure. The prediction is then computed by taking a weighted average (for example equation 1.1) of the target user's ratings r on these similar items. User-based CF algorithm first calculates the similarity $w_{u,v}$, between users u and v who have both rated the same items [6].

$$p_{u,i} = \frac{\sum_{n \in N} r_{u,n} - w_{i,n}}{\sum_{n \in N} |w_{i,n}|} \tag{1.1}$$

There are many different methods to calculate similarity or weight between users or items. Usual measures are for example correlation-based similarity (Eq. 1.2), cosine-based similarity (Eq. 1.3) and so on [6].

$$w_{u,v} = \frac{\sum_{i \in I}(r_{u,i} - \hat{r}_u)(r_{v,i} - \hat{r}_v)}{\sqrt{\sum_{i \in I}(r_{u,i} - \hat{r}_u)^2} \sqrt{\sum_{i \in I}(r_{v,i} - \hat{r}_v)^2}} \tag{1.2}$$

$$w_{i,j} = \frac{\vec{i} \cdot \vec{j}}{||\vec{i}|| ||\vec{j}||} \tag{1.3}$$

1.2.1.1.2 Scalability

Complexity of the neareset neighbor algorithm grows with both the number of users and the number of items, hence it has limited scalability for large datasets. From view of interactions dynamic, user-based CF suffers more than item-based CF. Unlike similarity between users, items similarity is more or less static, therefore it enables precomputing of item-item similarity [7].

1.2.1.2 Matrix Factorization

Matrix Factorization algorithm belongs to a family of latent factors model. Latent factor model tries to explain the ratings by characterizing both user and item by a vector inferred from the ratings pattern. When user and item do highly correspond, it provides a lead to a recommendation.

Matrix Factorization models map users and items to a joint latent factors f dimensional space. User-item interaction are modeled as inner products in that space. Each user is then represented by vector $\vec{u} \in \mathcal{R}^f$ and each item by vector $\vec{v} \in \mathcal{R}^f$. The dot product $\vec{u}_u^T \vec{v}_i$ captures the interactions between user u and item i [8]. This approximates rating given by user u to item i:

$$r_{u,i} = \vec{u}_u^T \vec{v}_i \tag{1.4}$$

The most challenging part is finding the mapping of each item and user to factor vectors $u_u, v_i \in \mathcal{R}^f$. However, when the mapping is complete, recommender system can easily estimate unobserved ratings by using the equation 1.4.

A common way to learn the factor vectors u_u and v_i is to minimize the regularized squared error on the set of observed and unobserved ratings 1.5.

$$\min_{u^*,v^*} \sum_{(u,i)\in K} c_{u,i}(r_{u,i} - u_u^T v_i)^2 + \lambda(\|u_u\|^2 + \|v_u\|^2) \tag{1.5}$$

Here, K is the set of the all (u, i) pairs, the constant λ controls the extent of regularization and $c_{u,i}$ is confidence of rating [9].

1.2.1.2.1 Learning Procedures Several learning algorithms have been developed for searching the minimum of 1.5. The most common ones are stochastic gradient descent, alternating least squares and coordinate descent.

1.2.1.2.1.1 Stochastic Gradient Descent The algorithm loops through all ratings in the training set, for each training example compute associated error $e_{u,i} = r_{u,i} - u_u^T v_i$. Then update parameters according to Eq. 1.6.

$$\begin{aligned} u_u &= u_u + \gamma \cdot (c_{u,i}e_{u,i} \cdot v_i - \lambda u_u) \\ v_i &= v_i + \gamma \cdot (c_{u,i}e_{u,i} \cdot u_u - \lambda v_i) \end{aligned} \tag{1.6}$$

1.2.1.2.1.2 Alternating Least Square The ALS technique is based on switching between fixed u_u and fixed v_i. When all u are fixed the optimization problem becomes quadratic and can be solved optimally. Then system recomputes v_i by solving a least squares problem (in general e.g. 1.7) and vice versa [8].

$$\theta = (X^T CX)^{-1} X^T CY \tag{1.7}$$

In general, stochastic gradient descent is easier and faster than ALS, however ALS can be easily parallelized. The algorithm computes each u_u and v_i independently of the other users, items factors respectively. Due to this independence, computation can be massively distributed [8].

1.2.1.2.1.3 Coordinate Descent Approaches The basic idea of coordinate descent is similar to ALS. A single variable is updated at a time while keeping others fixed [10].

1.2.1.2.1.4 Extensions In recent years, lots of extensions have been developed, for example adding user, item, and global biases. The prediction of the rating is then decomposed to 4 parts μ, b_i, b_u and $u_u^T v_i$ 1.8, where μ is the overall average rating, and the parameters b_u and b_i indicate the observed deviation of user u and item i, respectively, from the average. Further, adding user and item bias terms tend to capture much of the observed signal, their accurate modeling is vital. Hence, this allows each component to explain only the part of a signal relevant to it [8].

Another example is temporal dynamics, which enriched the static model with the ability to model temporal effects [8]. The terms b_i^t, b_u^t and $u_{u^T}^t v_i^t$ can then vary over time 1.9.

$$\hat{r}_{u,i} = \mu + b_i + b_u + u_u^T v_i \tag{1.8}$$

$$\hat{r}_{u,i}^t = \mu^t + b_i^t + b_u^t + u_{u^T}^t v_i^t \tag{1.9}$$

1.2.1.2.1.5 Scalability The time complexity per iteration of SGD is $O(|K|k)$, ALS $O(|K|k^2 + (m+n)k^3)$ and coordinate descent based algorithm (CCD++) $O(|K|k)$, where K is the set all observed ratings, m is the number of users, n the number of items, and k the number of factors (size of latent vector representation). Despite the fact, that ALS has the worst asymptotic complexity, massive parallelization can mitigate this issue. Further, the SGD suffers from sensitivity on the choice of the learning rate, when compared to CCD++[11].

1.2.1.3 Factorization Machines

The Factorization Machines is another example of factorization models similarly to MF. It combines the advantages of Support Vector Machines with factorization models. FMs model interactions between variables using factorized parameters. Thanks, this property, FMs are capable estimate interaction even in a problem with huge sparsity where other models fail [12]. Equation 1.10 shows factorization machine of degree $d = 2$.

$$\hat{y}(\vec{x}) = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} < v_i, v_j > x_i x_j \tag{1.10}$$

where $\vec{x}$ is vector of features, w_0 is global bias, w_i models the strength of i-th feature, v_i and v_j are latent representations of feature x_i and x_j, $< v_i, v_j >$ is dot product and model interactions between the i-th and j-th feature. The FM can be extended to arbitrary degree of interaction.

1.2.1.3.1 Learning Procedure The author proposed several learning procedures such as stochastic gradient descent, alternating least squares, and Markov Chain Monte Carlo. Even stochastic gradient descent can learned parameters efficiently. The gradient of the FM model is [12]:

$$\frac{\partial}{\partial\theta}\hat{y}(\vec{x}) = \begin{cases} 1, & \text{if } \theta \text{ is } w_0 \\ x_i, & \text{if } \theta \text{ is } w_i \\ x_i \sum_{j=1}^{n} v_{j,f} x_j - v_{i,f} x_i^2, & \text{if } \theta \text{ is } v_{i,f} \end{cases} \tag{1.11}$$

1.2.1.3.2 Scalability The model equation 1.10 is feasible to compute in linear time $O(kn)$, where n is a number of features and k size of latent representation. Thus, the algorithm does not suffer from pairwise interactions [12].

1.2.2 Deep Learning based Recommendation Algorithms

In many fields such as computer vision and speech recognition, deep learning (DL) is tremendously successful. This trend continues the past few decades and the academia and industry have been in a race to apply deep learning to a wider range of application. Recently, deep learning has been appearing in the domain of recommendation systems and brings more opportunities in reinventing the user experiences for better customer satisfaction. Deep learning can efficiently capture the nonlinear and non-trivial user-item relationships and leverage abundant data sources such as contextual, textual and visual information [13].

1.2.2.1 Deep Learning Techniques

This part will briefly introduce deep learning techniques, which are used in following recommender system algorithms.

1.2.2.1.1 Multilayer Perceptron Multilayer Perceptron is a feedforward neural network with multiple (one or more) hidden layers between input layer and output layer. The perceptron can hold arbitrary activation function.

1.2.2.1.2 Autoencoder Autoencoder is an unsupervised model, which is trained to reconstruct its input data in the output layer.

1.2.2.1.3 Recurrent Neural Network Recurrent Neural Network has been designed to be able model sequential data. In RNN are loops and memories to remember previous computations. Variants such as Long Short Term Memory (LSTM)and Gated Recurrent Unit (GRU) have been developed to improve network capabilities.

1.2.2.2 Collaborative Deep Learning

Collaborative Deep Learning is a hierarchical Bayesian model, which jointly models deep representation for the content information and collaborative filtering for the ratings matrix. The model combines stacked denoising autoencoder (SDAE) with probabilistic matrix factorization. SDAE is a deep neural network, which is able to process various side information. PMF acts as the task-specific component. These two parts are tightly coupled and enable CDL to balance the influences of side information (SDAE) and ratings (PMF) [13]. The generative process can be defined as follow:

- For each layer l of the SDAE network,
  - For each column n of weight matrix W_l, draw $W_{l,*n} \sim \mathcal{N}(0, \lambda_w^{-1} I_{D_l})$.
  - Draw the bias vector $b_l \sim \mathcal{N}(0, \lambda_w^{-1} I_{D_l})$
  - For each row i of $X_{l,i*} \sim \mathcal{N}(\sigma(X_{l-1,i*} W_l + b_l), \lambda_s^{-1} I_{D_l})$
- For each layer item i,
  - Draw a clean input $X_{c,i*} \sim \mathcal{N}(X_{L,i*}, \lambda_n^{-1} I_{I_i})$
  - Draw a latent offset vector $\epsilon_i \sim \mathcal{N}(0, \lambda_u^{-1} I_D)$ and set the latent item vector: $V_i = \epsilon_i + X^T_{\frac{L}{2},i*}$
- Draw a latent user vector for each user u, $u_u \sim \mathcal{N}(0, \lambda_u^{-1}, I_D)$
- Draw a rating $r_{u,i}$ for each user-item pair (u,i), $r_{u,i} \sim \mathcal{N}(u_u^T v_i, C_{u,i}^{-1})$

where W_l and b_l arethe weight matrix and biases vector for layer l, X_l represents layer l. $\lambda_w, \lambda_s, \lambda_n, \lambda_v, \lambda_u$, are hyper-parameters, $C_{u,i}$ is a confidence parameter for measuring the confidence to observation, X_c is the clean input, X_0 is corupted input [13].

Several technique to find parameters of model may be applied. The authors expoited an EM-style algorithm for obtaining the maximum a posteriori estimates. In this case, one maximizes join-log-likelihood of U, V, X_l, X_c, W_l, b_l, and R given $\lambda_w, \lambda_s, \lambda_n, \lambda_v, \lambda_u$. Then, equation 1.12 can give intuition behind the model.

$$\begin{aligned}
\mathcal{L} = &-\frac{\lambda_u}{2}\sum_u \|u_u\|^2 - \frac{\lambda_w}{2}\sum_{l \in layers}(\|W_l\|^2 + \|b_l\|^2) \\
&-\frac{\lambda_v}{2}\sum_j \|v_j - X^T_{\frac{L}{2},j*}\|^2 - \frac{\lambda_n}{2}\sum_j \|X_{L,j*} - X_{c,j*}\|^2 \\
&-\frac{\lambda_s}{2}\sum_l \sum_j \|\sigma(X_{l-1,j*}W_l + b_l) - X_{l,j*}\|^2 \\
&-\sum_{u,i}\frac{C_{u,i}}{2}(r_{u,i} - u_u^T v_i)^2.
\end{aligned} \tag{1.12}$$

If λ_s goes to infinity, it simplifies model and the likelihood reduces to 1.13:

$$\begin{aligned}\mathcal{L} = &-\frac{\lambda_u}{2}\sum_u \|u_u\|^2 - \frac{\lambda_w}{2}\sum_{l\in layers}(\|W_l\|^2 + \|b_l\|^2)\\ &-\frac{\lambda_v}{2}\sum_j \|v_j - f_e(X_{s,j*}, W^+)^T\|^2 - \frac{\lambda_n}{2}\sum_j \|f_r(X_{s,j*}, W^+) - X_{c,j}\|^2\\ &-\sum_{(u,i)}\frac{C_{u,i}}{2}(r_{u,i} - u_u^T v_i)^2\end{aligned} \tag{1.13}$$

where W^+ denotes the collection of all layers weights and biases, $f_e(\cdot, W^+)$ is encoder function and $f_r(\cdot, W^+)$ computes encoding and then reconstructs the content vector of item j. The first two terms are regularization, second line terms balance model reconstruction error between item content vector $X_{c,j}$ and item latent vector v_j and the last term incorporate task-specific component. One can approximate the prediction rating as 1.14 for both models.

$$R_{i,j} \approx u_i^T v_j \tag{1.14}$$

1.2.2.2.1 Learning Procedure One of the possible learning procedure is very similar to the idea of ALS. Learning algorithm alternates between fixed W^+ and latent representations U, V of users, items, respectively. Latent representations updates (Equations 1.15) lead to the updates rules similar to 1.2.1.2.1.2 and for W and b, authors use a modified version of backpropagation [13].

$$\begin{aligned}u_u &= (V^T C_u V + \lambda_u I_k)^{-1} V^T C_u R_u\\ v_i &= (U^T C_i U + \lambda_i I_k)^{-1}(U^T C_i R_i + \lambda_v f_e(X_{0,j*}, W^+))\end{aligned} \tag{1.15}$$

1.2.2.2.2 Scalability Authors presented update rules have the computational complexity of updating u_i $(O(k^2 J + k^3))$, where k is the size of latent representation and J is the number of items. Update v_j has complexity $O(k^2 I + k^3 + SK_1)$, where I is the number of users, S is the size of the vocabulary, and K_1 is dimensionality of output in the first layer.

1.2.2.3 Wide & Deep

This general model can be used for both regression and classification problems. The wide part of the model is single layer perceptron, and the deep learning part is multilayer perceptron. The authors suggested that combining these two components enable to capture both memorization and generalization. For catching the direct features from historical data is proposed the memorization component and deep learning component produce more general and abstract representation [13].

Formally, the wide component is a generalized linear model 1.17, where x is the vector of features, w and b are model parameters. The features include various transformed and raw features. The authors noted, that cross-product transformation is one of the most important [14].

$$y = w^T x + b \tag{1.16}$$

The deep component is a feed-forward neural network. Categorical features are first transformated to low-dimensional and dense real-valued vectors often referred as an embeddings. At the start, the embeddings are initialized randomly and during model training are updated to minimize the final loss function. These embeddings are then fed into the hidden layers. Each hidden layer calculates the following computation:

$$a^{(l+1)} = f(W^{(l)} a^{(l)} + b^{(l)}) \tag{1.17}$$

where l is the layer number, f is the activation function and $a^{(l)}, b^{(l)}$ and $W^{(l)}$ are the activations, bias, and weights at l-th layer.

1.2.2.3.1 Learning Procedure The wide component and deep component are coupled using a weighted sum of their outputs and then fed into one common logistic loss function for joint training. Thus, during training both components are trained simultaneously. The authors suggested using backpropagation for training[14].

1.2.2.3.2 Scalability The authors have evaluated the Wide & Deep model on a massive-scale commeracial app store Google Play. At the peak, the model has had to score over 10 million apps per second [14].

1.2.2.4 Session-based GRU4Rec

GRU4Rec is significantly differ from models mentioned before. The task of previous models is simply recommend the most relevant items to user, whereas the goal of GRU4Rec is recommend item based on user session. The authors argue, that this approach is more suitable for many e-commerce recommender systems. Particularly small retailers, most of news and media sites, because they do not typically track the user-id's of the users that visit their sites over a long period of time. Further, cookies and browser fingerprinting are not always reliable enough [15].

To overcome this issue, authors have proposed GRU-based reccurent neural network (RNN). The current state of session is fed to the RNN while the output of is the item of the next event in the session. The state of the session is either the item of the current event or the events in the session so far. The authors have used 1-of-N encoding in the first case, i.e. binary vector equal to the number of items filled with zeros except the coordinate corresponding to the active item and weighted sum of theses representations [15].

The architecture is composed from GRU layer(s) and additionaly feedforward layers can be added between the last and the output layer. The output layer predicted preferences of the items of being the next in the session. When more then one GRU layers are used, the hidden state of the previous layer is the input of the next one. Further, optionally, the input can be connected to the successive GRU layers. The authors has reported, that this option improves performance [15].

1.2.2.4.1 Learning Procedure The authors proposed a session-parallel mini-batches, a sampling method for output and own loss function. Mini-batches have been constructed as follow:

- create ordered sessions,
- use first event of the first X sessions to form input, desired outputs are the seconds events of active session,
- the next mini-batch is formed from the seconds events and so on,
- if any session end, the next available session is put in its place.

The loss function 1.18 belongs to family of ranking losses and measure relative rank of the relevant item according to sampled items. N_s is sample size, σ is a sigmoid function, $\hat{r}$ are predicted item scores.

$$L_s = \frac{1}{N_s}\sum_{j=1}^{N_s} \sigma(\hat{r}_{s,j} - \hat{r}_{s,i}) + \sigma(\hat{r}_{s,j}^2) \tag{1.18}$$

Due to a potentially large number of items, computing scores for each of them would be unusable in practice. Therefore the authors have suggested sampling items in proportion to their popularity. Instead of generating separate samples, they have been used items from previous mini-batches as negative examples [15].

1.2.2.4.2 Scalability The authors have suggested a method for significantly reducing time complexity. However, the authors in [16] argue, that much more simple methods as frequent-pattern-based approaches can be competitive in accuracy and result strongly depends on datasets characteristics. Compare to GRU4Rec these simple models learn much faster and applying the rules is very fast. Despite this fact, the authors expect that continuously improved RNN-based methods will be able to outperform the frequent pattern based algorithms used in their evaluations [16].

1.3 Hyper-parameters Optimization Techniques

In the context of a recommender system, hyper-parameters optimization is essential for adequate performance in real-world applications. For example, on the recommender system related ads recommendation, authors reported results, which thanks online tuning of algorithm showed a significant 4.3 % revenue lift overall traffic [17].

An offline-world fine-tuning is as essential as fine-tuning in a real-world running application. Deploying low-performance models can lead to decreasing revenue, user's satisfaction, a number of users, etc. Therefore is necessary do a careful model preselections.

Most of the recommender algorithms have numerous hyper-parameters, which can be tuned and accordingly significantly change their behavior. To deal with optimization of high number and ill-conditioned hyper-parameters has developed several techniques and different approaches.

1.3.1 Problem Definition

Most of machine learning task can be described as training model M which minimizes some predefined loss functions $L(X^{test}; M)$ on given test data X^{test}. The model M is constructed by a learning algorithm A using a training set X^{train}. The algorithm A can be parametrized by hyper-parameters H itself, e.g. $M = A(X^{train}, H)$. The goal of hyper-parameters search is to find a set of hyper-parameters H of model M that yield to minimize function $L(X^{test}, M)$ [18]. Formally:

$$H^* = \arg\min_{H} L(X^{test}; A(X^{train}; H)) \tag{1.19}$$

1.3.2 Hyper-parameter Optimization Techniques

This section provides a brief overview of optimization techniques that have been reported in recent years in the domain of recommendation systems.

1.3.2.1 Grid Search

Grid search is a method, which calculates all possible hyper-parameters settings. It works only with discrete hyper-parameters since the continuous hyper-parameters space is infinite. Therefore is often necessary discretize continuous hyper-parameters. Recommendations algorithms have often several hyper-parameters and then is grid search highly inefficient, due to the exponential growth of possible combinations of hyper-parameters.

1.3.2.2 Random Search

Random search uniformly samples hyper-parameters within given bounds. Despite this simplicity, the random search has huge advantage in easy parallelization.

1.3.2.3 Greedy Search

Gready search optimize only one hyper-parameter a time, while fixing remaining hyper-parameters. For each hyper-parameter is selected m random samples and then fixated to the best from the random samples. The procedure is then repeated with remaining hyper-parameters.

1.3.2.4 Random Walk

Random Walk is an iterative method. In each step, it computes the performance of each neighbour, based on a selection mechanism select one of the neighbours and makes step towards this neighbour.

1.3.2.5 Simulated Annealing

Simulated Annealing is related to the random walk method. It uses temperature as a control variable that manages probability of selection worse solutions than the current one by selection mechanism. As temperature decrease, the probability of choosing worse solution also decrease.

1.3.2.6 Nelder-Mead

Nelder-Mead method search minimum of function by spanning a simplex in the hyper-parameter space. The simplex has $k+1$ vertices, where k is the number of hyper-parameters. The simplex can move in space of hyper-parameters by leveraging four transformation: reflection, contraction, expansion and shrinking [19].

1.3.2.7 Particle Swarm Optimization

Particle swarm optimization is inspired by behaviours of swarms. A population of particles called swarm move in the space of hyper-parameters and maintains its velocity and its best position. The entire swarm store the global best position. Each particle adjust its position according to personal and global best position.

1.3.2.8 Genetic Algorithm

Genetic algorithm is inspired by the process of natural selection present in evolution. Each individual is part of a population and it is a candidate for a solution. The information about the individual is encoded to an array. The following process of optimization is iterative. In each round, the individuals are combined, possibly mutated and selected based on a fitness function. Only those individuals with high fitness are likely to transfer their information to the next generation.

1.3.2.9 Sequential Model-based Optimization

Model-based optimization methods build a regression model that predicts the performance of a target algorithm. These models iterate between the additional data collection and constructing approximation model, often called surface or surrogate model. The model is fitted to a training set $(\theta_1, y_1), \ldots (\theta_n, y_n)$ of observed performance y_i when target algorithm run with hyper-parameters θ_i. The Gaussian process as surrogate model is common choice [20].

Chapter 2

Solution Analysis

2.1 Algorithms Analysis

The goal of selected algorithms is to be able to operate with various datasets without expert knowledge about the content of these datasets. The datasets contain a huge number of interactions and items related auxiliary information. MF algorithm is one of the most successful algorithms in domain of recommendation systems which for training leverage only user-item interactions [21]. In contrast with MF, CDL algorithm is capable use various kind of auxiliary information and therefore, in some cases, increase recommendation quality. For these reasons, MF and CDL have been chosen.

2.1.1 Matrix Factorization

From wide family of matrix factorization based algorithms [22], the algorithm in [23] has been mainly followed because of stability of learning algorithm and parallelization potencial. Further, it is extended by optional user and item bias terms.

The system learns by minimizing the squared error function:

$$\begin{aligned}\min_{u^*,v^*} \sum_{r_{u,i} \in observed} (1+\alpha)(r_{u,i} - \hat{r}_{u,i})^2 + \sum_{r_{u,i} \in missing} (0 - \hat{r}_{u,i})^2 \\ + \sum_{u} \lambda n_u \|u_u\|^2 + \sum_{i} \lambda n_i \|v_i\|^2\end{aligned} \tag{2.1}$$

where $\hat{r}_{u,i}$ is $b_u + b_i + u_u^T v_i$, b_i and b_u are bias terms for user u respectively item i, $r_{u,i}$ is observed rating given user u to item i, u_u and v_i are user u respectively item i latent vectors, term $1+\alpha$ plays a role of weight, n_u and n_i are numbers of observed ratings for user u respectively item i and λ is regularization parameter.

To deal with a searching minimum of function 2.1 one can use several techniques as we have discussed earlier in section 1.2.1.2. The Weighted Alternating Least Square optimization method has been used in this work. The solution of the Least square problem is obtained by solving a system of linear equations a.k.a normal equation [24]. Following pseudocode describes algorithm without biases [23]:

```
initialize V, U;
for $i$ to $N$ do
    $UtU \leftarrow U^T U$
    foreach $i \in items$ do
        $UtU \leftarrow$ UtU $+\alpha U_{obs}^T U_{obs} + \lambda n_i I$;
        $Utr \leftarrow (1+\alpha) U_{obs} r_{obs}$;
        $V_i \leftarrow solve(UtU, Utr_{obs})$;
    end
    $VtV \leftarrow V^T V$
    foreach $u \in users$ do
        $VtV \leftarrow VtV + \alpha V_{obs}^T V_{obs} + \lambda n_u I$;
        $Vtr \leftarrow (1+\alpha) V_{obs} r_{obs}$;
        $U_u \leftarrow solve(VtV, Vtr_{obs})$;
    end
end
```

Algorithm 1: Weighted ALS Matrix Factorization

I in 1 refers to identity matrix, V_{obs} and U_{obs} to latent factors corresponded to observed user-items interactions.

Matrix factorization with biases (Algorithm 2) has almost identical pseudocode except that it is necessary to subtract biases from ratings, add dummy columns filled with value one and then do carefully indexing during optimizations. Superscript $^{(1)}$ expresses added dummy columns, (:,:) indexing matrices and I^0 resp. I_0 identity matrix with zero in the first resp. the last diagonal element.

```
initialize $V^{(1)}$, $U^{(1)}$;
for $i$ to $N$ do
    $UtU \leftarrow U^{(1)}(:,:-1)^T U^{(1)}(:,:-1)$
    foreach $i \in items$ do
        $U_{obs} \leftarrow U^{(1)}_{obs}(:-1)$
        $b_u \leftarrow U^{(1)}_{obs}(-1)$
        $UtU \leftarrow$ UtU $+ \alpha U^T_{obs} U_{obs} + \lambda n_i I^0$;
        $Utr \leftarrow (1+\alpha) U_{obs}(r_{obs} - b_u)$;
        $V_i(:-1) \leftarrow solve(UtU, Utr_{obs})$;
    end
    $VtV \leftarrow V^{(1)}(:,1:)^T V^{(1)}(:,1:)$
    foreach $u \in users$ do
        $V_{obs} \leftarrow V^{(1)}_{obs}(1:)$
        $b_v \leftarrow V^{(1)}_{obs}(0)$
        $VtV \leftarrow VtV + \alpha V^T_{obs} V_{obs} + \lambda n_u I_0$;
        $Vtr \leftarrow (1+\alpha) V_{obs}(r_{obs} - b_v)$;
        $U_u(1:) \leftarrow solve(VtV, Vtr_{obs})$;
    end
end
```

Algorithm 2: Weighted ALS Matrix Factorization with biases

2.1.2 Collaborative Deep Learning

Simplified version of CDL [25] with minor change has been used in this work. That change incorporates adjusting each user's latent vector regularization to λ_u multiply by his number of ratings.

The simplified version was presented as part of the KDD 2016 [26]. Hence, that version is possible to separate into two components, i.e., autoencoder and Matrix Factorization and one can recycle the implementation of the matrix factorization 2.1.1 with just minor adjustments.

The system learns by minimizing the following function [27]:

$$
\begin{aligned}
\min_{u^*,v^*,W^*,b^*} \; & \sum_{r_{u,i} \in observed} (1+\alpha)(r_{u,i} - \hat{r}_{u,i})^2 + \sum_{r_{u,i} \in missing} (0 - \hat{r}_{u,i})^2 \\
& + \lambda_u n_u \sum_u \|u_u\|^2 \\
& + \lambda_w \sum_{l \in layers} (\|W_l\|^2 + \|b_l\|^2) \\
& + \lambda_v \sum_j \|v_j - f_e(X_{s,j}, W^+)^T\|^2 \\
& + \lambda_n \sum_j \|f_r(X_{s,j}, W^+) - X_{c,j}\|^2
\end{aligned}
\tag{2.2}
$$

where W and b are autoencoder's weights and biases, $X_{s,j}$ is vector of item j content, $X_{c,j}$ is vector of reconstructed item j content, v_j is the item j latent representation comming from the Matrix Factorization component, f_e is the encoder function and f_r is function returned the reconstructed content vector.

The first line of equation refers to matrix factorization component, the second line is user's regularization term, the third line is autoencoder's weights and biases, the fourth line forces item's latent vectors and item's encoded representation to be similar in Frobenius norm and the last line do the same with original and reconstructed item's content.

The pseudocode 3 describes learning procedure of simplified CDL. One can see, that the autoencoder component and the matrix factorization component are not trained jointly, but alternating between each other. While parameters one of the components have been training, parameters second component are fixed.

```
1  initialize V, U;
2  preprocess content of items;
3  for $i$ to $N$ do
4      foreach $(X_{batch}, V_{batch}) \in batches$ do
5          $trainAutoEncoder(X_{batch}, V_{batch})$;
6      end
7      $theta \leftarrow AutoEncoder(X)$;
8      $UtU \leftarrow U^T U$;
9      foreach $i \in items$ do
10         $UtU \leftarrow$ UtU $+\alpha U_{obs}^T U_{obs} + \lambda_v I$;
11         $Utr \leftarrow (1+\alpha) U_{obs} r_{obs} + \lambda_v theta_i$;
12         $V_i \leftarrow solve(UtU, Utr_{obs})$;
13     end
14     $VtV \leftarrow V^T V$;
15     foreach $u \in users$ do
16         $VtV \leftarrow VtV + \alpha V_{obs}^T V_{obs} + \lambda_u I$;
17         $Vtr \leftarrow (1+\alpha) V_{obs} r_{obs}$;
18         $U_u \leftarrow solve(VtV, Vtr_{obs})$;
19     end
20 end
```

Algorithm 3: Collaborative Deep Learning

2.1.3 User Recommendation Calculation

Conventional approach to user recommendation computation is for each item to calculate score as dot product (equation 2.3) between latent user representation and latent item's representation and then sort them in descending order. On the top of the list are the most relevant items to the user.

$$score(u, v) = u \cdot v = \sum_{i=1}^{n} u_i v_i = u_1 v_1 + u_2 v_2 + \cdots + u_n v_n \tag{2.3}$$

Second approach tested in this work is compute the score as a cosine similarity between user u and item v 2.4.

$$score(u, v) = cos(u, v) = \frac{u \cdot v}{\|u\| \|v\|} = \frac{\sum_{i=1}^{n} u_i v_i}{\sqrt{\sum_{i=1}^{n} u_i^2 \sum_{i=1}^{n} v_i^2}} \tag{2.4}$$

2.2 Optimization Procedure

Probably the most relevant source to this thesis is the work [21]. Authors benchmarked various algorithms on four real-world datasets. The algorithm chosen for hyper-parameter optimization was the matrix factorization with the objective to optimize the RMSE (root mean square error). As the conclusion, the authors suggested that in scenarios, where marginal improvement plays the critical role the Nelder-Mead algorithm should be used. The algorithm achieved the best performance on all datasets, followed by simulated annealing. Although Nelder-Mead algorithm achieved the best performance, improvement compared to random search was neglectful. Therefore after considering many advantages of random search, such as full parallelization, simplicity, constant and nearly negligible computation time, they recommend use it for hyper-parameter optimization in the domain of recommender systems.

Authors in [28] demonstrated on state-of-the-art ANN models for dialog act classification, that optimizing hyper-parameters using Gaussian Processes (GP) further improves the results and reduces the computational time by a factor of 4 compared to a random search. Further they compared various GP settings and impact of the number of initial random hyper-parameter combinations.

Based on these results, Gaussian Processes and Random Search methods have been chosen as algorithms suitable for fine-tuning recommendation algorithms.

2.2.1 Evaluation Measures

As in [27] we have adopted recall as a measure of recommender algorithm accuracy. We sort the predicted ratings of the candidate items and recommend the top K items to the target user. Equation 2.5 show recall calculation. The final result reported is the average recall over all users.

$$recall@K \leftarrow \frac{\#L_{rec}}{\min(K, \#L)} \tag{2.5}$$

where $\#L_{rec}$ is number of items that the user likes among the top K, $\#L$ is total number of items that the user likes.

As an additional measure, catalog coverage has ben chosen. Catalog coverage can be express as the percentage of the available items which effectively are ever recommended to a user[29]. It is given by equation 2.6:

$$coverage@K \leftarrow \frac{\left| \bigcup_{u \in test\ users} I_u \right|}{|I|} \tag{2.6}$$

where I_u is a set of recommended items to user during evaluation phase and I is set of all items in catalog.

2.2.2 Optimization Procedure Design

At first, it is necessary to identify hyper-parameters suitable for fine tuning. This is done by initial Random Search (RS). Based on results of the initial random search, part of hyper-parameters let be fixed and remaining hyper-parameters are searched by grid search to expose simple and obvious dependencies between hyper-parameters and algorithm performance. The performance of an algorithm is measured by recall desribed in 2.2.1.

After detecting obvious dependencies between hyper-parameters and algorithm performance, a selected subset of hyper-parameters is fixed or set to an appropriate range and remaining hyper-parameters are tuned according to GP or RS. Algorithm 4 is an overview of the process.

```
input : $D$ (dataset), $A$ (recommendation algorithm),$O$ (either GP or
        RS), $RS$ (Random Search), $H$ (space of $A$ hyper-parameters)
1  $res \leftarrow \{\}$;
2  for $i$ to $N$ do
3      $\theta \leftarrow$ selecthyper-parameters($RS$, $A$, $H$);
4      $model \leftarrow$ train($A$, $\theta$ , $D_{train}$);
5      $res \leftarrow$ eval($model$,$D_{test}$);
6  end
7  $H \leftarrow$ adjust($H$, eval($res$));
8  res $\leftarrow \{\}$;
9  for $i$ to $M$ do
10     $\theta \leftarrow$ selecthyper-parameters($O$, $A$, $H$);
11     $model \leftarrow$ train($A$, $\theta$ , $D_{train}$);
12     $res \leftarrow$ eval($model$, $D_{test}$);
13 end
14 eval($res$);
```

Algorithm 4: Evaluation procedure

Further, due to strong relationship between MF and CDL, second approach has been designed. CDL and MF have broad intersection of hyper-parameters. Therefore, one can try to leverage the knowledge gained from MF optimization and set CDL hyper-parameters to the same values as associated hyper-parameters of the optimal MF model. Algorithm 5 shows an overview of the process.

```
input : $D$ (dataset), $A$ (CDL algorithm), $O$ (either GP or RS), $H_{MF}$
        (MF optimal hyper-parameters), $H_{CDL}$ (space of CDL
        hyper-parameters)
1 $H_{CDL} \leftarrow$ adjust($H_{CDL}$, $H_{MF}$);
2 res $\leftarrow$ {};
3 for $i$ to $M$ do
4     $\theta \leftarrow$ selecthyper-parameters($O$, $A$, $H_{CDL}$);
5     $model \leftarrow$ train($A$, $\theta$ , $D_{train}$);
6     $res \leftarrow$ eval($model$, $D_{test}$);
7 end
8 eval($res$);
```

Algorithm 5: Evaluation procedure

Chapter 3

Realization

Within the scope of this thesis, a dataset preprocessing pipeline, a recommender algorithm evaluation pipeline, a hyper-parameter fine-tuning pipeline and two algorithms have been implemented. During implementation, we have focused on scalability and a potential possibility to deploy particular algorithm as a component of the real-world hybrid recommender system.

3.1 Programming Language

The whole system is written in Python focusing on easy extensibility and editability. The main reason for choosing Python as a programming language is that it enables fast prototyping of an idea [30]. Also, Python contains an enormous amount of machine learning libraries.

3.2 Library Usage

Following libraries has been used in this work.

- NumPy [1] – NumPy is a fundamental package for scientific computing with Python.
- PyTorch [2] – PyTorch is a popular deep learning framework with pythonic syntax.
- Spark [3] – Apache Spark is a powerful open source processing engine built around speed, ease of use, and sophisticated analytics [4].

[1]http://www.numpy.org/
[2]http://pytorch.org/
[3]http://spark.apache.org/docs/2.1.0/api/python/pyspark.html
[4]https://databricks.com/spark/about

- Scikit-optimize [5] – Scikit-optimize (skopt) is a simple and efficient library to minimize (very) expensive and noisy black-box functions.
- Scikit-learn [6] – Scikit-learn provides simple and efficient tools for data mining and data analysis.
- Matplotlib [7] – Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.
- Pandas [8] – Pandas is an open source library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.
- Majka [9] – Majka is a free morphological analyzer. Majka includes databases for Czech, Slovak, Polish, Swedish, German, French, Italian, English, Portuguese and others.
- TensorFlow [10] – TensorFlow is mainly designed for deep neural network models.
- Jupyter [11] – Jupyter Notebook is an open-source web application that allows create and share documents that contain live code, equations, visualizations and narrative text.

3.3 Implementation

The diagram 3.1 provides an overview of the whole system, each component is then described in following sections.

3.3.1 Data Access

nonpublic datasets (described in 4.1) were provided in the PostgreSQL database. The Spark Python API (PySpark) [12] was used to connect to the database and basic filtering operations. The main advantage of this approach is that PySpark allows multicore data processing. After data loading and basic filtering, PySpark data structures have been transformated to Pandas data structure for easier manipulation.

[5]https://scikit-optimize.github.io/
[6]http://scikit-learn.org/stable/
[7]https://matplotlib.org/
[8]https://pandas.pydata.org/
[9]https://nlp.fi.muni.cz/czech-morphology-analyser/
[10]https://www.tensorflow.org/get˙started/
[11]http://jupyter.org/
[12]http://spark.apache.org/docs/2.1.0/api/python/pyspark.html

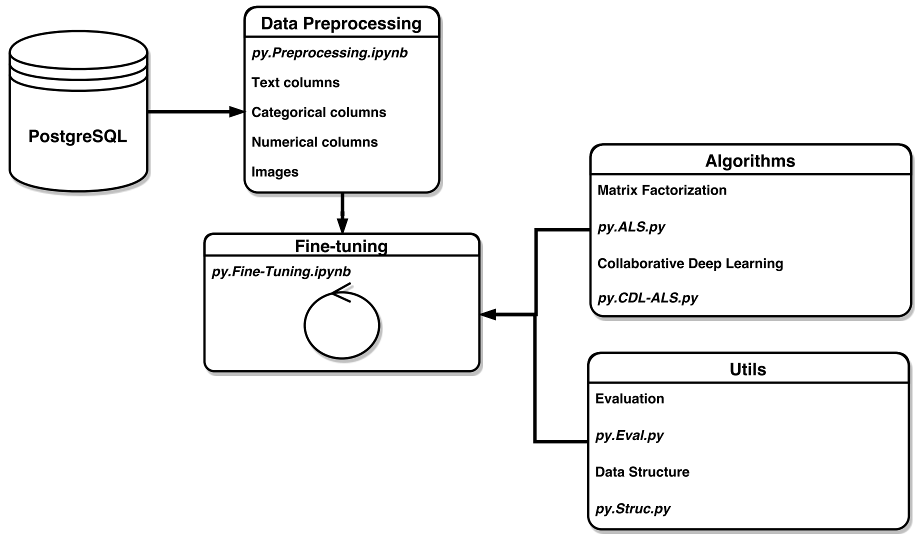

Figure 3.1: System Overview

3.3.2 Data Preprocessing

Despite the fact that this work has focused on minimizing the need to incorporate data analysis and the expertise of datasets owners, some pre-processing of datasets has been required. At the end of preprocessing serialized data has been stored in disk.

3.3.2.1 Numeric Data Preprocessing

Numerical columns have been transformed according to the formula 3.1, where c indicated column, v_i^c original value in row of column c, $\hat{\mu}^c$ is mean and $\hat{\sigma}^c$ is standard deviation of values in particular column and $\hat{v_i^c}$ transformated value v_i^c.

$$\hat{v_i^c} \leftarrow \frac{v_i^c - \hat{\mu}^c}{\hat{\sigma}^c} \tag{3.1}$$

3.3.2.2 Categorical Data Preprocessing

Rows with categorical data (e.g. tags, categories) have been simply split by appropriate separator.

3.3.2.3 Text Preprocessing

In cases, when columns contain text in natural language, for instance, news article description or title, a lemmatizer has been used. In the scope of this work, datasets suitable for this type of processing have been only in the Czech language. The data processing steps for each row have been as follows:

- removing punctuation,
- removing stopwords (the lists of Czech stopwords [13] and [14]),
- collecting the first lemma suggested by Majka,
- chaining collected lemmas.

3.3.2.4 Image Preprocessing

For image preprocessing has been used script available in GitHub [15]. The script is a wrapper for feature extraction in TensorFlow. It offers well known pre-trained models on ImageNet [16]. From offered models, the Inception v4 has been selected cause reported high performance compared to other models. Further, the script allows choosing the extraction layer that returns the resulting embeddings. The Logits layer has been chosen.

Because the datasets do not contain images but only links to images, the first step has been downloading images to disk and then using the script for feature extraction mentioned above. To summarize, whole procedure has been looked as follows:

- collecting all links to images,
- using the linux command wget to download images to disk,
- using the script to extract features from images.

After image feature extraction, each one has been described as vector of size 1001 with real numbers.

3.3.2.5 Leveraging Preprocessed Data

After data has been preprocessed, categorical columns and lemmatized text data have been chained and transformed to a vector with parameterizable size by function HashingVectorizer [17].

HashingVectorizer function turns a collection of text documents into a matrix holding token occurrence counts (or binary occurrence information). It allows normalizing token frequencies, either l1 norm or projected on the euclidean unit sphere. The HashingVectorizer implementation leverages the hashing trick to find the token string name to feature integer index mapping.

Hashed data has been then chained with numerical columns. This approach is universal and it is possible to replicate on an arbitrary dataset.

[13]https://github.com/stopwords-iso/stopwords-cs

[14]https://github.com/crodas/TextRank/blob/master/lib/TextRank/Stopword/czech-stopwords.txt

[15]https://github.com/tomrunia/TF˙FeatureExtraction

[16]https://github.com/tensorflow/models/tree/master/research/slim#pre-trained-models

[17]http://scikit-learn.org/stable/modules/generated/sklearn.feature˙extraction.text.HashingVectorizer.html

3.3.3 Algorithms

The following sections describe implementation details of chosen algorithms and bring some insight to implementation scalability. Simply put, the implementation of Collaborative Deep Learning uses the implementation of MF. Therefore MF results also apply to the MF component of CDL.

3.3.3.1 Matrix Factorization

As noted in the section 2.1.1, Weighted Alternative Least Squares approach with solving a normal equation has been used. This approach has one immediately visible advantage and one disadvantage. Because calculation of a user latent vector is independent to other users (same for items), calculations are easy to parallelize. This property is leveraged, for example, by the MLlib library (Apache Spark) [18]. On the other hand, solving a system of linear equations is asymptotically cubic for a solver used in this work[19].

The reason, why the official implementation has not been used, is that original version [20] is implemented in C++ and MATLAB, therefore, it would be hard and expensive to incorporate it into the current hybrid recommender system. Further, due to a sequential implementation of simplified version [21], this version is not scalable and it is not feasible to use it for huge datasets.

3.3.3.1.1 Computation Parallelization Due to the global interpreter locker (GIL) [22], it is not feasible to use a thread-level parallelization for computations speed up. Therefore, Python package multiprocessing has been used [23] for a process-level parallelization. It allows distributing calculations of user and item latent representations to multiple processes.

A new artifical dataset has been generated for measurements. The dataset contains approximately 1 million users, 100 thousand items and 10 millions interactions. Each measured parametrization has been repeated three times and reported best achieved wall time. For measurement, standard IPython function timeit [24] has been used. The calculations were run on a machine with 6 physical cores and 6 logical cores. More details are in appendix A.1.2.

Fig. 3.2 shows speed up with respect to the number of processes. On the x-axis is number of processes used to compute latent representations and on the y-axis is duration in minutes. One can see, that duration decreases according to a number of physical cores. After the number of physical cores is exceeded, the time consumed by computation stays approximately the same.

[18]https://spark.apache.org/docs/2.2.1/mllib-collaborative-filtering.html
[19]http://www.netlib.org/lapack/lug/node71.html
[20]https://github.com/js05212/CDL
[21]https://github.com/js05212/MXNet-for-CDL
[22]https://wiki.python.org/moin/GlobalInterpreterLock
[23]https://docs.python.org/3.6/library/multiprocessing.html
[24]http://ipython.readthedocs.io/en/stable/interactive/magics.html

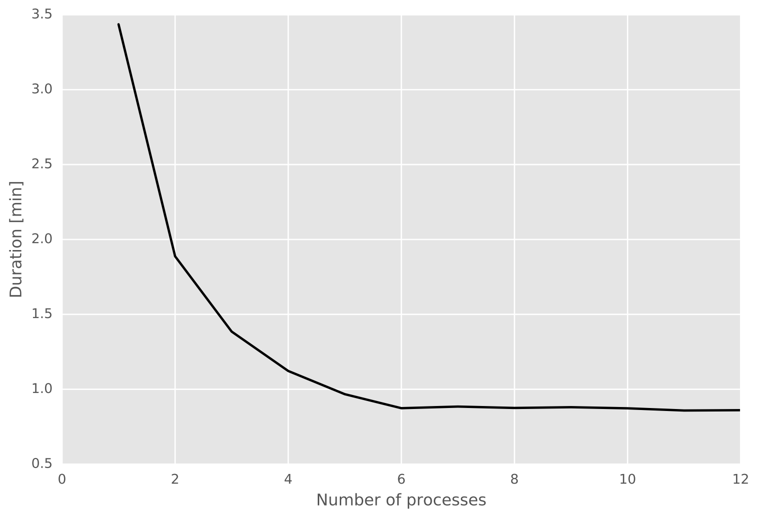

Figure 3.2: Scalability: number of processes

3.3.3.1.2 Data Structures To leverage process-level parallelization, it has been necessary to choose an appropriate data structure for storing latent representations of users and items. For this purpose has been chosen a data structure multiprocessing.Array, which is offered by the same package as process-level parallelization. This structure allows to share memory between processes.

Furthermore, it has been shown it is very important to choose the right data structure for quick access to user and item data. Therefore, a new data structure has been implemented, which is fit to the MF implementation in this work. It is allows by fast indexing in Python list data structure access to data. One disadvantage is that for users and items, the structures have to be created separately, so it claims twice as much memory, but on present hardware, this is not restrictive.

3.3.3.1.3 Latent Representation Calculation For linear algebraic operations necessary for computation latent representations, the package NumPy has been used. To get maximal performance, disabling multithreading support in NumPy package has been necessary.

In the Figure 3.3 one can see a relation between the size of latent representation and the duration of calculation. As before, each measurement has been repeated three times and reported the best-achieved wall time. One can see exponentially increasing duration with respect to the size of latent representation.

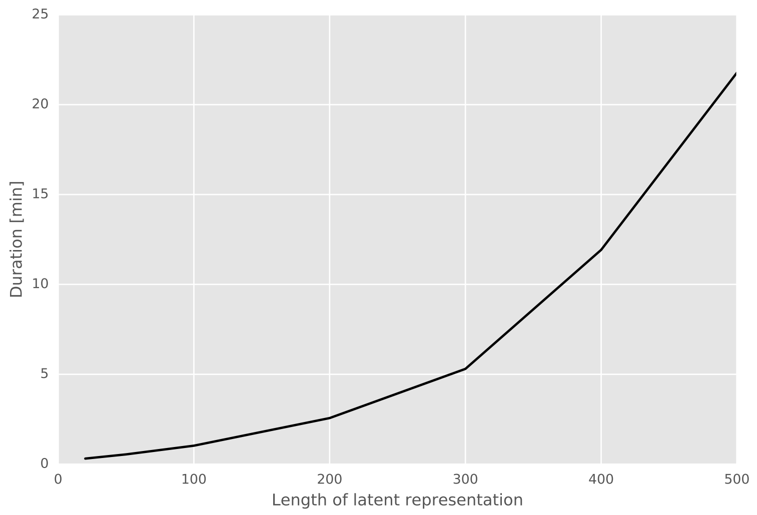

Figure 3.3: Scalability: size of latent representation

3.3.3.2 Deep Collaborative Filtering

As noted before, one component of CDL is the adjusted MF implementation. This modification does not change the complexity of MF implementation. Therefore in the following part, we will focus more on an Autoencoder component.

3.3.3.2.1 Matrix Factorization and Autoencoder linking A whole model has been designed using the framework PyTorch. The framework behaves as a common Python package. Hence, connection to MF component has been comfortable. Effectively, both models are in the same Python script and share same variables.

3.3.3.2.2 Autoencoder Architecture The Autoencoder architecture follows the CDL author's suggestions in his article [27] and in his code accessible on GitHub [25]. Generally, it consists of a various number of layers, with ReLu or originally sigmoid activation functions and trained by stochastic gradient descent with momentum or by ADAM optimizer.

In contrast with original article, an input and output layer is fed by a vectorized preprocessed auxiliary information as described in the previous part 3.3.2.5, not by bag of words.

3.3.3.2.3 Scalability Comparison The implementation suggested in this work contains process-level parallelization of computation MF component

[25]https://github.com/js05212/MXNet-for-CDL/blob/master/collaborative-dl.ipynb

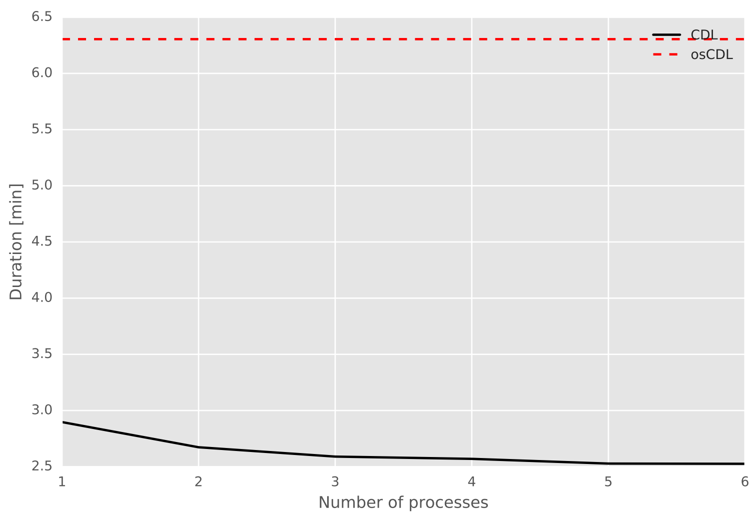

Figure 3.4: Scalability: original simplified CDL (loosely dashed) and proposed CDL (solid) implemenation

compared to the proposed implementation of the author of CDL. The Figure 3.4 shows comparison of these two implementations. For the comparison, dataset CiteULike has been used, and models have beet set to the same hyper-parameters. The x-axis shows number of processes and the y-axis shows duration.

3.3.4 Algorithm Performance Evaluation

Due to the significant number of models that have to be evaluated during hyper-parameter optimization, it is necessary to have a fast evaluation. Because the chosen measures allow to evaluate each user separately, similarly to 3.3.3.1.1, process-level parallelization has been used for reduction of evaluation time.

3.3.5 Fine-tuning pipeline

Ready-to-use optimizers from the package scikit-optimizer have been used in a fine-tuning pipeline. Based on the analysis in section 1.3, a Bayesian optimization using Gaussian Processes (function gp_minimize) and a random search by uniform sampling (function dummy_minimize) have been selected.

The entire fine-tuning environment has been implemented in Jupyter Notebook. The code in Jupyter Notebooks is separated into cells, each cell can be run repeatedly and in any order, while variables are shared among cells, moreover it enable to render charts directly in the IDE, all these features make

Jupyter Notebook an ideal tool for comfortable hyper-parameters tuning, responding to results and reporting them.

The whole fine-tuning procedure can be described by following steps:

- loading data from disk,
- setting searching space of hyper-parameters,
- running optimizer,
- storing results to disk,
- loading results,
- results analysis.

Chapter 4

Experiments

Experiments are conducted on one public dataset and two nonpublic datasets from different domains. The goal of the experiments is to compare two hyperoptimization technique and also compare two algorithms quantitatively.

4.1 Datasets

Within the scope of this work, three datasets from a different real-world domains for experiments were chosen. The first CiteULike has been used in [27] for evaluation CDL performance and it is available from [26]. CiteULike allows users to create their own collections of articles. There are abstract, title, and tags for each article. It contains 5551 users, 16980 items and 49960 interactions. The remaining three datasets are not public and have been provided by the company Recombee [27].

One of nonpublic datasets, henceforth Fashion, contains data about user-item interactions as detail views, purchases, bookmarks and items description as a price, categories, a brand, a title, a link to an image. Fashion dataset contains ca. 100 000 items, 8 million users, 30 millions of items detail views and about 2 million purchases.

The last dataset comes from news webpage, henceforth News. Besides user-item interactions it contains article features such as a title, tags, and authors of articles. It contains about 450 000 articles, 100 million users and 600 million interactions.

4.2 Evaluation Scheme

Due to the size of nonpublic datasets and vast amount of experiments during fine-tuning, each dataset has been subsampled to approximately 25 000 users.

[26]http://www.wanghao.in/CDL.htm
[27]https://www.recombee.com/

Further, users in each dataset have been randomly divided into two non-overlapping subsets (90 % training, 10 % testing), see 2.2.2. The training set consists of 90 % of users with all their interactions. The testing set consist of the remaining 10 % users. Items that are not in the training set are removed from the testing set. For each user in the testing set, we hide an item in his history, the remaining items are used to calculate user's latent representation. The latent representation is used to calculate personalized recommendations. We repeat this procedure N times. Furthermore we record list of top K recommended items and if the hidden item appears in it. The number of successful trials divided by $min(K\ ,\ \#user's\ interactions - 1)$ (see 2.5) is value of user's recall. N recorded lists are used to calculate coverage according to 2.6. K and N have been set to 10 during evaluation.

4.3 Experimental Settings

At first, the original simplified CDL implementation (osCDL) has been evaluated and compared to CDL implementation proposed in this thesis (nCDL). For this purpose, the dataset CiteUlike has been used. Other evaluations have been done on datasets CiteULike, Fashion and News with proposed implementation 2.1.2. The Fashion dataset has been evaluated in two variants of auxiliary information. First, the text features have been leveraged and later image embeddings. The optimization algorithms have been set to the default settigns except number of calls and for GP the acquisition function has been set to EIs. In this setting, GP takes into account the function compute time and expected improvement. According to 2.2.2, suitable hyper-parameters for fine-tuning have been identified for MF and CDL. This has been done based on first optimization procedure by random search algorithm. The MF suitable hyper-parameters are:

- size of latent representation,
- regularization,
- weight of observed ratings,
- activation of user and item bias term,
- score computation method.

An ideal number of iterations has oscillated around number 15. Because the time complexity grows cubicly with the size of latent representation, the size has been fixed to 80 for further experiments. The Figure 4.3 shows relation between size of latent representation and recall-coverage curves.

The CDL hyper-parameters suitable for fine-tuning are:

- λ_v and λ_n controlling reconstruction errors,

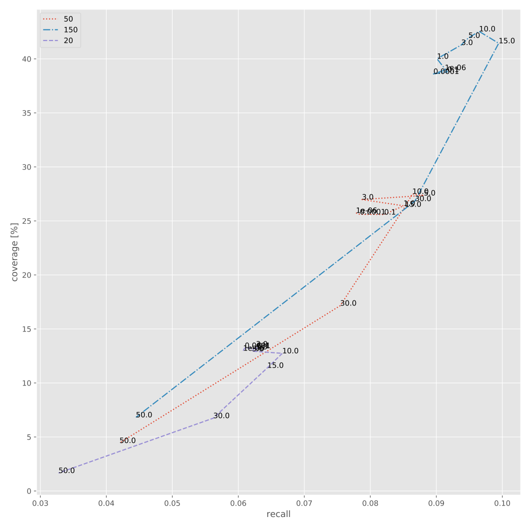

Figure 4.1: Coverage-Recall curve

- dropout,
- learning rate.

The hyper-parameters related to MF has not been listed for brevity. Size of mini-batches and input content vector have been derived from first random search such as number of iterations, which has been set to 60.

4.4 Quantitative Comparison

This part covers the quantitative comparison. First we compare original CDL implementation and proposed implementation. Then, the ability of optimization algorithms find optimal or suboptimal hyper-parameter settings of target algorithm. The last part is devoted to comparison MF and CDL accuracy, coverage and time complexity of training.

4.4.1 Implementation Accuracy Comparison

Figure 4.2 shows comparation osCDL, nCDL, osCDL's MF component and nCDL's MF component accuracy. For clarity, nMF component has been modified to mimic original MF component. Hyper-parameters CDLs has been set according to [28]. MF components have started with the same random seed. As one can see, MF components have exactly the same results, however nCDL with the same hyper-parameters as osCDL is worse. Despite doing my best, I

[28]https://github.com/js05212/MXNet-for-CDL/blob/master/collaborative-dl.ipynb

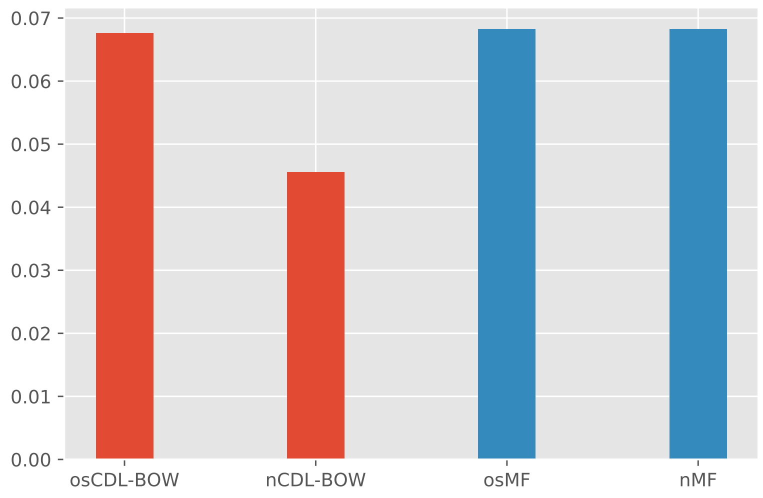

Figure 4.2: Accuracy: comparison between osCDL and nCDL

could not get closer to implementation osCDL. The osCDL is based on mxnet framework [29], whereas nCDL on PyTorch. These two frameworks do not have exact intersect of features and I believe, as results of MF components suggest, that difference between the performance of both models is caused by the fact, that nCDL autocoencoder does not exactly imitate osCDL Autoencoder implementation, mainly optimization procedure. However, as further results show, with different hyper-parameters, nCDL is able to beat nMF. Further, interestingly, the osMF models have beaten the osCDL models on almost same dataset (except excluded users for testing set) and measure, where the authors presented significant better performance then other models (MF was not included in this evaluation).

4.4.2 Optimization Algorithm Convergence and Algorithms Accuracy

According to results mentioned in 2.2 both RS and GP are able to find similarly performance suboptimal hyperparameters after similar numbers of optimization steps. Also in almost all datasets MF algorithm shows better accuracy and beats more complex model, which has a capability of leverage auxiliary information. Further, CDL needs approximately five times more iterations than MF to achieve similar results.

[29]https://mxnet.apache.org/

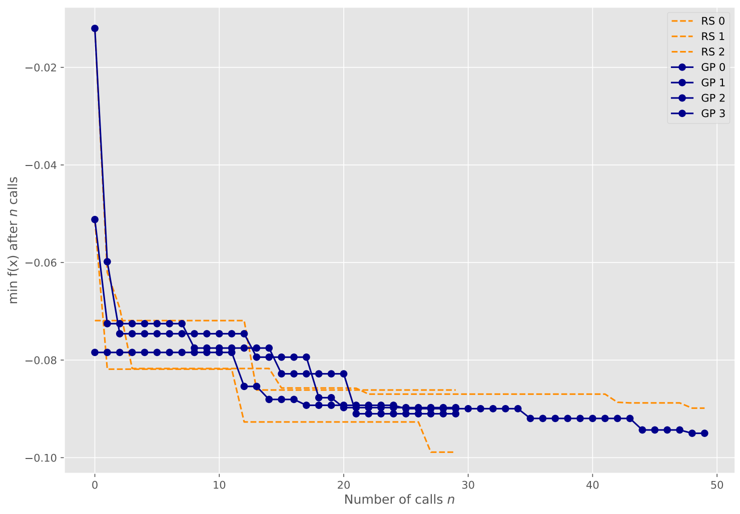

Figure 4.3: CiteULike MF convergence: RS vs. GP

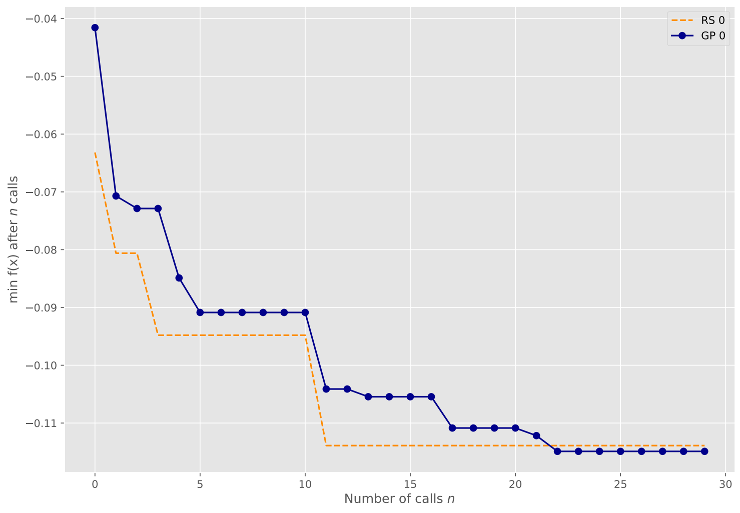

Figure 4.4: Fashion MF convergence: RS vs. GP

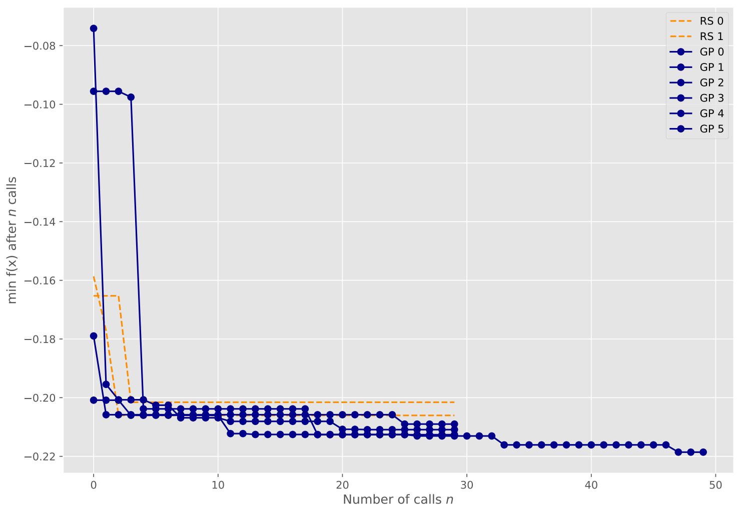

Figure 4.5: News MF convergence: RS vs. GP

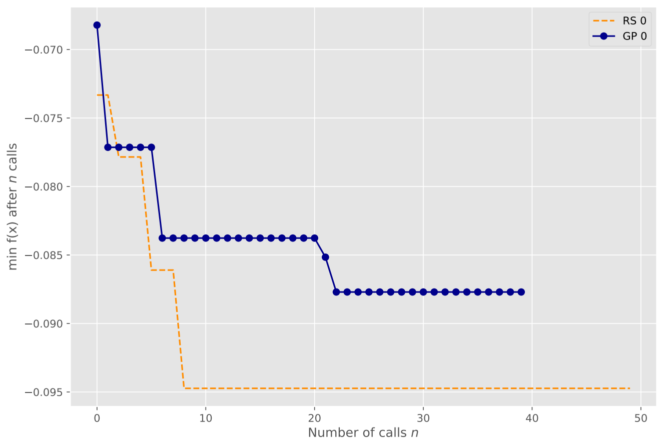

Figure 4.6: CiteULike CDL convergence: RS vs. GP

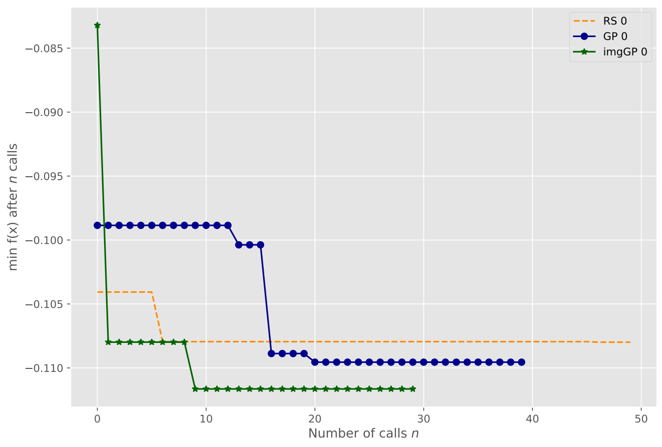

Figure 4.7: Fashion CDL convergence: RS vs. GP

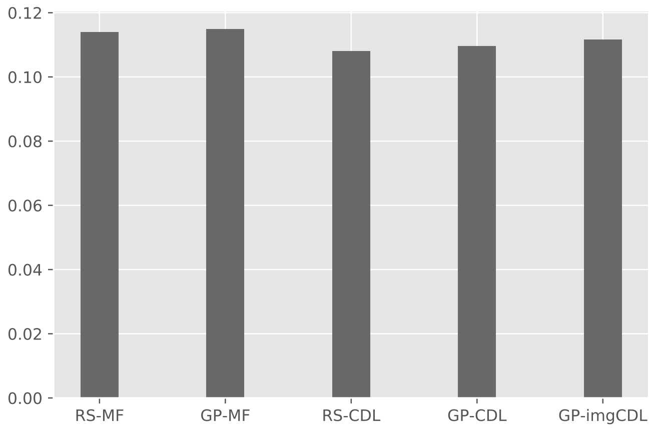

Figure 4.8: Fashion Recall

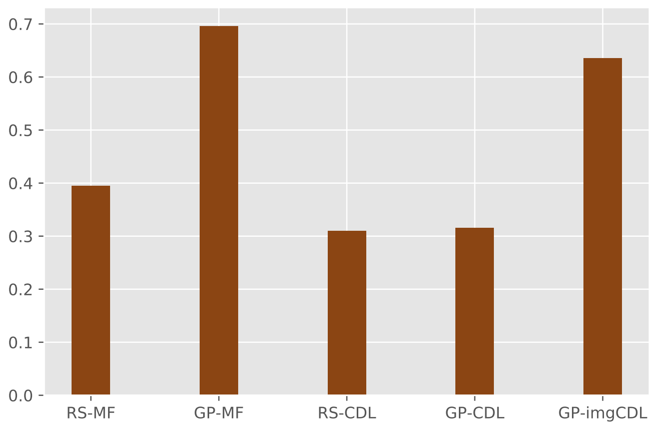

Figure 4.9: Fashion Coverage

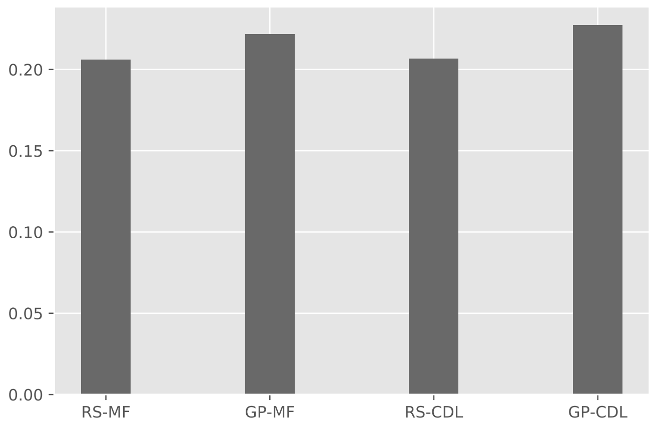

Figure 4.10: News Recall

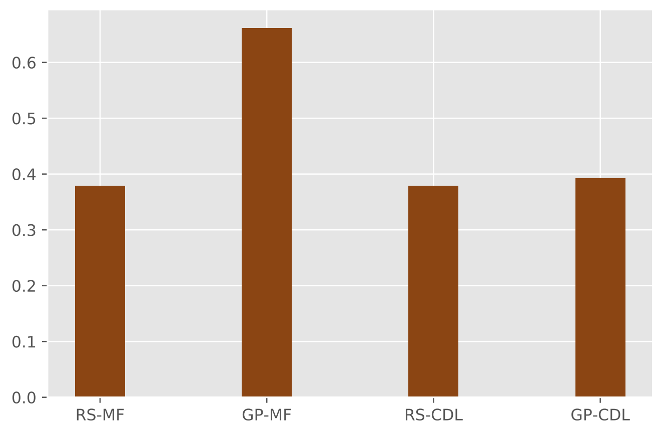

Figure 4.11: News Coverage

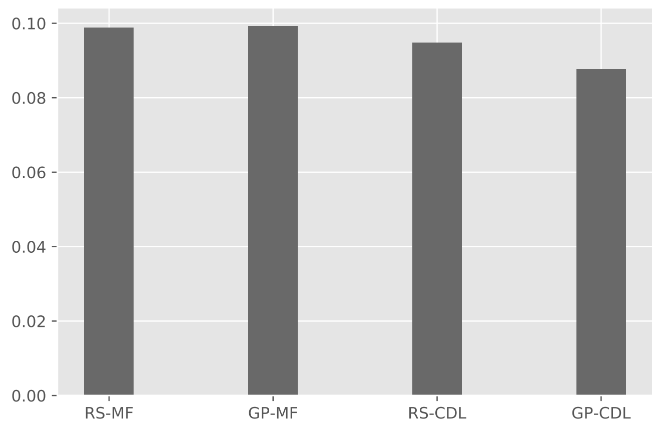

Figure 4.12: CiteULike Recall

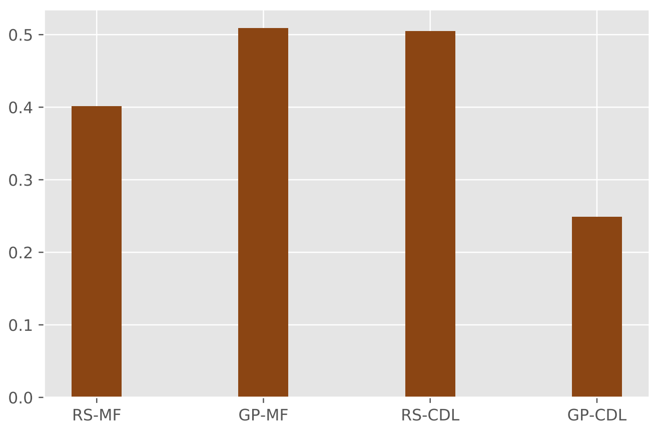

Figure 4.13: CiteULike Coverage

Conclusion

Within the scope of this thesis we covered several scalable traditional and deep learning based recommendation algorithms. We gave a brief overview of recommendation and hyper-parameter optimization techniques.

Further, two algorithms have been implemented, namely Matrix Factorization and Collaborative Deep Learning. Both implementations incorporate process-level parallelization and are thus well scalable.

For evaluation and fine-tuning of hyperparameters, the optimization pipeline has been proposed and implemented. The random search and gaussian process have been leveraged for the purpose of hyper-parameter tuning.

Three datasets have been used for evaluation of the algorithms and the whole pipeline. One public dataset and two unpublic datasets. The experiments show, that fine-tuning pipeline is capable tune hyper-parameters of the algorithms,while performance of both optimizers is very similar. In the end, surprising results the comparison of two recommendation algorithms has shown. More complex model Collaborative Deep Learning with an incorporation of auxiliary information has not been able to significantly beat simple Matrix Factorization.

Bibliography

[1] Ricci, F.; Rokach, L.; et al. *Recommender Systems Handbook*. New York, NY, USA: Springer-Verlag New York, Inc., first edition, 2010, ISBN 0387858199, 9780387858197.

[2] Nagarnaik, P.; Thomas, A. Survey on recommendation system methods. In *2015 2nd International Conference on Electronics and Communication Systems (ICECS)*, Feb 2015, pp. 1603–1608, doi:10.1109/ECS.2015.7124857.

[3] Son, L. H. Dealing with the new user cold-start problem in recommender systems: A comparative review. *Information Systems*, volume 58, 2016: pp. 87 – 104, ISSN 0306-4379, doi:https://doi.org/10.1016/j.is.2014.10.001. Available from: `http://www.sciencedirect.com/science/article/pii/S0306437914001525`

[4] Kodakateri Pudhiyaveetil, A.; Gauch, S.; et al. Conceptual recommender system for CiteSeerX. 01 2009.

[5] Poriya, A.; Bhagat, T.; et al. Article: Non-Personalized Recommender Systems and User-based Collaborative Recommender Systems. *International Journal of Applied Information Systems*, volume 6, no. 9, March 2014: pp. 22–27, published by Foundation of Computer Science, New York, USA.

[6] Su, X.; Khoshgoftaar, T. M. A Survey of Collaborative Filtering Techniques. *Adv. in Artif. Intell.*, volume 2009, Jan. 2009: pp. 4:2–4:2, ISSN 1687-7470, doi:10.1155/2009/421425. Available from: `http://dx.doi.org/10.1155/2009/421425`

[7] Yao, G.; Cai, L. User-Based and Item-Based Collaborative Filtering Recommendation Algorithms Design. 2015.

[8] Koren, Y.; Bell, R.; et al. Matrix Factorization Techniques for Recommender Systems. *Computer*, volume 42, no. 8, Aug 2009: pp. 30–37, ISSN 0018-9162, doi:10.1109/MC.2009.263.

[9] Hu, Y.; Koren, Y.; et al. Collaborative Filtering for Implicit Feedback Datasets. In *2008 Eighth IEEE International Conference on Data Mining*, Dec 2008, ISSN 1550-4786, pp. 263–272, doi:10.1109/ICDM.2008.22.

[10] Yu, H. F.; Hsieh, C. J.; et al. Scalable Coordinate Descent Approaches to Parallel Matrix Factorization for Recommender Systems. In *2012 IEEE 12th International Conference on Data Mining*, Dec 2012, ISSN 1550-4786, pp. 765–774, doi:10.1109/ICDM.2012.168.

[11] Yu, H.-F.; Hsieh, C.-J.; et al. Parallel matrix factorization for recommender systems. *Knowledge and Information Systems*, volume 41, no. 3, Dec 2014: pp. 793–819, ISSN 0219-3116, doi:10.1007/s10115-013-0682-2. Available from: `https://doi.org/10.1007/s10115-013-0682-2`

[12] Rendle, S. Factorization Machines with libFM. *ACM Trans. Intell. Syst. Technol.*, volume 3, no. 3, May 2012: pp. 57:1–57:22, ISSN 2157-6904.

[13] Zhang, S.; Yao, L.; et al. Deep Learning based Recommender System: A Survey and New Perspectives. *CoRR*, volume abs/1707.07435, 2017.

[14] Cheng, H.-T.; Koc, L.; et al. Wide Deep Learning for Recommender Systems. *arXiv:1606.07792*, 2016. Available from: `http://arxiv.org/abs/1606.07792`

[15] Hidasi, B.; Karatzoglou, A.; et al. Session-based Recommendations with Recurrent Neural Networks. *CoRR*, volume abs/1511.06939, 2015, `1511.06939`. Available from: `http://arxiv.org/abs/1511.06939`

[16] Kamehkhosh, I.; Jannach, D.; et al. A Comparison of Frequent Pattern Techniques and a Deep Learning Method for Session-Based Recommendation. In *RecTemp@RecSys*, 2017.

[17] Aharon, M.; Kagian, A.; et al. Adaptive Online Hyper-Parameters Tuning for Ad Event-Prediction Models. 2017: pp. 672–679, doi:10.1145/3041021.3054184. Available from: `https://doi.org/10.1145/3041021.3054184`

[18] Claesen, M.; Moor, B. D. Hyperparameter Search in Machine Learning. *CoRR*, volume abs/1502.02127, 2015, `1502.02127`. Available from: `http://arxiv.org/abs/1502.02127`

[19] Price, C.; Coope, I.; et al. A Convergent Variant of the Nelder–Mead Algorithm. *Journal of Optimization Theory and Applications*,

volume 113, no. 1, Apr 2002: pp. 5–19, ISSN 1573-2878, doi: 10.1023/A:1014849028575. Available from: `https://doi.org/10.1023/A:1014849028575`

[20] Hutter, F.; Hoos, H. H.; et al. Sequential Model-based Optimization for General Algorithm Configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, LION'05, Berlin, Heidelberg: Springer-Verlag, 2011, ISBN 978-3-642-25565-6, pp. 507–523, doi:10.1007/978-3-642-25566-3_40. Available from: `http://dx.doi.org/10.1007/978-3-642-25566-3_40`

[21] Matuszyk, P.; Castillo, R. T.; et al. A Comparative Study on Hyperparameter Optimization for Recommender Systems. 2016. Available from: `http://socialcomputing.know-center.tugraz.at/rs-bda/`

[22] Singh, A. P.; Gordon, G. J. A Unified View of Matrix Factorization Models. In *ECML/PKDD*, 2008.

[23] Hu, Y.; Koren, Y.; et al. Collaborative Filtering for Implicit Feedback Datasets. 2008: pp. 263–272, doi:10.1109/ICDM.2008.22. Available from: `http://dx.doi.org/10.1109/ICDM.2008.22`

[24] *Normal Equations*. New York, NY: Springer New York, 2008, ISBN 978-0-387-32833-1, 380–382 pp., doi:10.1007/978-0-387-32833-1_286. Available from: `https://doi.org/10.1007/978-0-387-32833-1_286`

[25] Wang, H. MXNet-for-CDL. `https://github.com/js05212/MXNet-for-CDL`, 2017.

[26] Mu Li, T. C. MXNet, 2016, kDD2016. Available from: `http://www.kdd.org/kdd2016/tutorials/view/mxnet`

[27] Wang, H.; Wang, N.; et al. Collaborative Deep Learning for Recommender Systems. 2015: pp. 1235–1244, doi:10.1145/2783258.2783273. Available from: `http://doi.acm.org/10.1145/2783258.2783273`

[28] Dernoncourt, F.; Lee, J. Y. Optimizing Neural Network Hyperparameters with Gaussian Processes for Dialog Act Classification. *CoRR*, volume abs/1609.08703, 2016, 1609.08703. Available from: `http://arxiv.org/abs/1609.08703`

[29] Shani, G.; Gunawardana, A. Evaluating Recommender Systems. Technical report, November 2009. Available from: `https://www.microsoft.com/en-us/research/publication/evaluating-recommender-systems/`

[30] Lanaro, G. *Python High Performance Programming*. Packt Publishing, 2013, ISBN 1783288450, 9781783288458.

Appendix **A**

Appendix

A.1 Measurements

A.1.1 Machine

- 6 physical + 6 logical core Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50GHz
- 252 GB RAM
- OS Linux dev02 4.9.0-4-amd64 #1 SMP Debian 4.9.51-1 (2017-09-28) x86_64 GNU/Linux
- Python 3.6
- NumPy 1.13.0

A.1.2 Measurement of relationship between number of processes and time

A.1.2.1 Matrix Factorization parametrization

- size of latent representation: 50
- alpha: 100
- regularization: 0.03
- number of iterations: 2
- bias: no
- normed: no

A.1.3 Measurement of relationship between size of latent representation and time

A.1.3.1 Matrix Factorization parametrization

- alpha: 100
- regularization: 0.03
- number of iterations: 2
- bias: no
- normed: no
- number of iteration: 1

A.1.3.2 Performance Comparison

- alpha: 1
- beta: 0.01
- optimizer: sgd
- learning rate: 0.1
- decay: 0
- momentum: 0.9
- dropout: 0.2
- size of lantent representation: 50
- size of auxiliary information embedding: 8 000
- number of iterations: 17
- regularization users: 1
- regularization items: 10

Appendix **B**

Acronyms

CDL Collaborative Deep Learning

MF Matrix Factorization

GRU Gated Recurrent Unit

LSTM Long short-term memory

CF Collaborative filtering

ALS Alternating Least Squares

SGD Stochastic Gradient Descent

FM Factorization Machines

Appendix **C**

Contents of enclosed CD

```
│ readme.txt ....................... the file with CD contents description
├── src ....................................the directory of source codes
│   └── thesis ..............the directory of LaTeX source codes of the thesis
└── text .......................................the thesis text directory
    ├── DP_Drdák_Jakub_2018.pdf ............the thesis text in PDF format
    └── DP_Drdák_Jakub_2018.ps ...............the thesis text in PS format
```