FH Aachen

University of Applied Sciences

Master's Thesis

Engineering a Hybrid Reciprocal Recommender System Specialized in Human-to-Human Implicit Feedback

Kai Dinghofer

January 28, 2021

1st Examiner	Prof. Dr. rer. nat. Heinrich Faßbender
2nd Examiner	Torben Hensgens, M. Eng.

Contents

List of Figures

Acronyms

RS Recommender System. v, 1–3, 5–7, 13, 15, 18, 20–31, 34–40, 50–53, 63, 65, 66, 69, 73, 74, 81, 87, 102, 103, 105–108

MF Matrix Factorization. v, 10, 14–18, 103

FM Factorization Machine. v, 17, 18, 109

RRS Reciprocal Recommender System. v, vi, 1–3, 29–40, 43, 45, 47, 49–51, 61, 64, 66, 73, 75, 77, 78, 94, 101–108, 110

RE Reciprocal Environment. v, 33, 42, 43, 94, 104, 105

DL Deep Learning. 2, 63

CBF Content-Based Filtering. 6–10, 14, 15, 20, 23, 26, 52, 78, 82, 108

CF Collaborative Filtering. 6, 7, 9, 10, 14, 15, 17, 19, 20, 23, 26, 52, 63, 64, 69, 73, 78, 79, 103

ML Machine Learning. 11, 13, 14, 16, 21, 22, 28, 47, 59, 71, 107

SGD Stoachastic Gradient Descent. 14, 71

UX User Experience. 22

AI Artificial Intelligence. 24, 106

EU European Union. 24

SNS Social Network Site. 34, 35, 37, 54, 75, 78, 82, 84, 99, 105, 107

EoI Expression of Interest. 34, 42, 53, 59

API Application Programming Interface. 47, 50, 73, 81, 84, 88, 89, 99, 104, 105

NLP Natural Language Processing. 50, 93

UI User Interface. 51

UML	Unified Modelling Language. 52
ISE	Information Systems Engineering. 55
CSV	Comma-Separated Values. 55, 98
CG	Candidate Generator. 61–63
LRU	Least Recently Used. 62
RCF	Reciprocal Collaborative Filtering. 64, 78, 80, 103, 108, 109
AUC	Area Under the ROC-Curve. 68, 69
GQL	GraphQL. 84, 85, 93, 99, 105
DSL	Domain-Specific Language. 84
REST	Representational State Transfer. 85, 87
BFS	Breadth-First Search. 87–90, 93, 94, 99, 105
GPE	Geo-Political Entity. 91
UMAP	Uniform Manifold Approximation and Projection. 98
tSNE	t-distributed Stochastic Neighbor Embedding. 98
PCA	Principal Component Analysis. 98

1 Introduction

1.1 Motivation

In times of technology and information overload, user attention is a rare resource. To alleviate the problem, so-called Recommender Systems (RS) are working in the background: Whenever we open up Netflix, shop on Amazon or search with Google, a system that recommends new *items* based on personal interests attempts to offer a tailored experience that fits in the bounds of our time-limited focus. Optimally, the provided recommendations let us discover innovative options that would otherwise perish in the huge amount of information that we need to filter (un)consciously. On the other side of the coin, this potential of RSs could be exploited solely as a means of maximizing profits, with the true quality of recommendations for the users being only of a second-class nature. Balancing two-sided interests for recommendations is key to long-term success, as described in the following.

As humans, we not only want to find a new movie, fashion product or search result. As social beings, we want to find the best people to connect to. Human-to-human recommendations can be applied to many different areas, for instance learning and studying (finding a suitable study partner), in the job industry (applicant and recruiter) or in more obvious social network scenarios (finding friends or dating). When it comes to typical RSs, it is important to follow the user's interests, but not the item's. To me, recommendations where both participants know that they contribute to success is what makes them an exciting technology. Hence, I specifically want to examine the state of the art for Reciprocal Recommender Systems (RRS): They heavily depend on mutual interests, the inherent reason of being a social being.

After first experiments, it became clear that this field of RSs is still in an early stage. To this end, I am very motivated to present *Chaos*, a framework for RRSs that hopefully enables researchers to discover new possibilities and interpretations in this field.

1.2 Objectives

A recurring and severe problem of research on RSs is that the results are often not *reproducible* in even “slightly different scenarios”, commonly called the *reproducibility crisis* of RSs [Bee+16]. The reproducibility of experiments is a fundamental aspect of open science and key to reliability and efficient advancement [Mun+17]. On one side, the limited reproducibility leads to comprehension problems: If the results cannot be reproduced, it is impossible to evaluate the validity of the study which is the necessary condition to further improve it. On the other side, novel RS approaches (for instance based on DL) are often claimed to outperform traditional algorithms, but are in the reality not consistently better [Dac+19], which leads to wrong expectations towards specific algorithms.

For the subarea of user-to-user RRSs, the problem is even further accelerated: There are only a few scientific papers and the field is still emerging [Agg+16, p. 444] [Pal+20]. Consequentially, because reproducibility on research of non-reciprocal RSs is already limited, the problem is worse for RRSs. Additionally, many studies about RRSs only use proprietary datasets for the evaluation of proposed algorithms [Ake+11] [Piz+10b] [Xia+16] [Pal+20] [NP19a]. This is especially a problem when the algorithm is fine-tuned on this specific dataset for maximized performance but cannot further generalize beyond that. Regarding the implementation side of RRSs, there are far less developer resources (libraries and frameworks) available for RRSs: For novice RS developers and researchers, getting started with RRSs is therefore challenging. It is often unclear *if* and *how* an established RS algorithm can be applied to reciprocal environments.

The following three key objectives are set to tackle the aforementioned problems:

1. Highlight the importance, characteristics and requirements of RRSs by comparing them with their parent, traditional RSs.
2. Implement a multi-purpose framework for reciprocal recommendations to aid and accelerate the research on RRSs.
3. Provide reproducible and open-source results.

Finally, the following is formulated as a research question to support and accompany the realization of the above objectives:

Can a user-to-item RS be utilized to generate reciprocal user-to-user recommendations?

1.3 Conventions

Throughout this work, important terms will be written *italic* on first occurrence. Acronyms (such as RS) always link to their full form in the glossary.

To highlight, **bold** formatting is used, but only rarely to not defeat the purpose of importance. Code examples and references to code, like keywords, are written in a `Monospace` font.

Additional illustrative examples are put into coloured boxes to support and deepen the understanding and to separate them from the text flow. Figures are numerated based on parent section, to make it easier to locate them.

1.4 Overview

We start with chapter 2 which is essentially a literature review of traditional user-to-item RSs that introduces important algorithms to build our foundation of common knowledge, therewith assisting humans who are not familiar with RSs. Moreover, we show common challenges when implementing a state-of-the-art RS. Readers with the necessary knowledge for RSs and the named key terms in the section headlines can decide to skip this chapter or alternatively only read the necessary ones.

In chapter 3, with the acquired knowledge, we explain what it means for a RS to be *reciprocal*. Thus, the differences and most relevant properties are highlighted. Then, we are able to derive requirements for a generic framework by comparing the challenges from the previous chapter and putting them into a reciprocal context.

In chapter 4, we introduce *Chaos*, a multi-purpose RRS framework implementation that is able to output reciprocal recommendations by using human-to-human interactions as input. We explain the main components of the framework and its most relevant concepts and software-architectural decisions. In addition, we provide example scenarios that can be reproduced by the interested reader, thus making the thesis partly interactive.

In the last chapter 5, we analyze strengths and limitations of this work and the implemented solution: Finally, we examine the contributions made by this work and propose future improvements and research possibilities in the outlook.

2 Recommender Systems

We start by introducing the basic terminology of RSs (section 2.1), their most popular types (section 2.2) and their building blocks – some of the most relevant algorithms which are applied in this thesis (section 2.2.2 and section 2.2.3).

In the last section 2.3, we analyze challenges of modern RSs that typically need to be tackled.

2.1 Terminology

The following basic terms are introduced beforehand:

- **Users**: The users of a RS are the human entity who receive recommendations.
- **Items**: Consequently, items are the entity to be recommended to the user.
- **Features**: Specific attributes of users that are (fully or partially) used as inputs to make new item recommendations to the user.
- **Interactions**: Interactions are made by users and either indicate a form of *explicit* feedback (e.g. star ratings as shown in fig. 2.1.1, like-/dislike button) or *implicit* feedback (e.g. viewing an item, using the share button) for an item. The latter does not clearly distinct between good and bad interactions, but usually many interactions to one and the same item are an expression of interest and liking, thus the feedback is implicit rather than explicit.

Bob, how would you rate our app?
★☆☆☆☆ I strongly dislike it...
★★★☆☆ I think it's okay.
★★★★★ I love it!

Figure 2.1.1: Example star ratings (explicit feedback through interaction)

2.2 Types

Figure 2.2.1 provides an overview of the three most important RS types which are discussed in the next sections. Content-Based Filtering (CBF) considers content features of users and/or items to infer recommendations, Collaborative Filtering (CF) uses collaborative information of multiple users and lastly Hybrid RSs incorporate both approaches in a suitable way. These are the core of most modern RSs and therefore fundamental to this thesis.

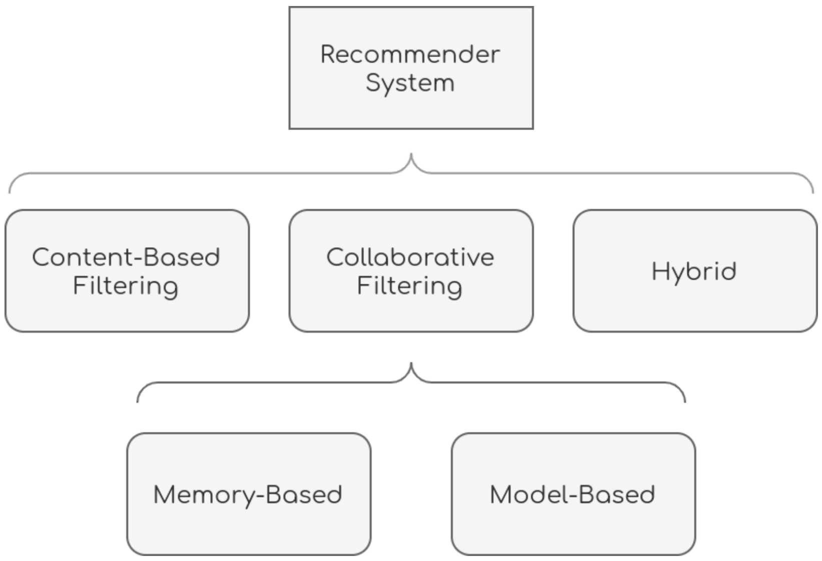


Figure 2.2.1: Type hierarchy of different RS (sub)divisions

In CF, one typically differentiates between *Model-Based* and *Memory-Based* methods. The latter were among the first studied algorithms in the collaborative filtering domain. Here, either user-oriented or item-oriented CF is used. Considering the objective to predict an unobserved rating for user U to an item I, they work in different but complementary ways: User-Based CF first calculates similar users to U and then averages the known ratings of the similar users. Item-based CF first calculates similar items to I and then approximates the rating by calculating the weighted average of already rated items by U [Agg+16]. Broadly speaking, these two Memory-Based methods are often too simple to represent complex real-world scenarios on their own. We therefore concentrate on Model-Based CF that usually need a model to be trained (explanation follows).

Both, CBF as well as CF often make use of an *Embedding Space* to capture semantic similarity. The more similar the items or users are, the "closer" they are positioned to each other, with the *Embeddings* being denoted by vectors. Figure 2.2.2 shows such a 2 dimensional embedding space E), with $u \in U$ being the user and $v_n \in V$ representing the items. There are multiple arithmetic methods to measure the *closeness* (or negated: the *distance*). The function for similarity can be chosen dependent on the specific use-case, e.g. *cosine*, *dot product* and *Euclidean distance*. For instance, the *dot product* is more suited to capture the popularity of an item, whereas cosine is scale-invariant [Goo18]. In fig. 2.2.2, all of the named metrics would calculate v_3 as the closest item for user u.

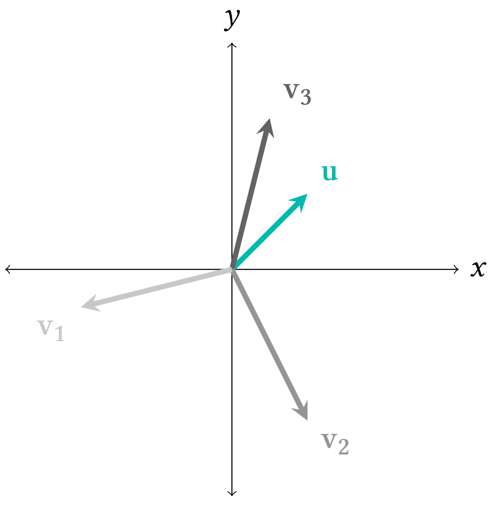


Figure 2.2.2: Simple 2D embedding space *ES*

Example 2.2.1: Embeddings for Fashion

One can think of the items v_n in fig. 2.2.2 being fashion products to be recommended to user u. The x axis captures the sustainability, e.g. v_1 represents a "fast-fashion" article produced under bad circumstances, opposing to v_2 and v_3 which implement some goals of sustainable development[a]. The y axis captures "trendiness", e.g. v_1 and v_2 being "oldschool", opposing to v_3 which strongly follows a recent trend. User u is positioned based on his preference. In this case, the user liked sustainable and trendy fashion products in the past.

The semantic of the embeddings is manually crafted here, and for many RS algorithms the meaning/embedding is actually *learned* through training a model, but here we wanted to show that a wide spectrum of information (features) can be used and embedded.

[a] see `https://sdgs.un.org/goals` for the UN sustainable development goals.

In the next section, the idea of embeddings becomes even more concrete and less abstract.

2.2.1 Content-Based Filtering

While there is a broad spectrum of *Content-Based Filtering* algorithms available, we focus on understanding the general approach and one exemplary similarity measure.

Consider the following scenario [see Goo18, CBF Basics]: Alice (user) loves smartphone applications and would like to install a new application (item). Recently, she has interacted with 2 applications (one from the publisher "Science R Us" and one from "Healthcare Solutions"). She also selected apps from the category "Education" when she has been asked for her preferences by her favorite app store.

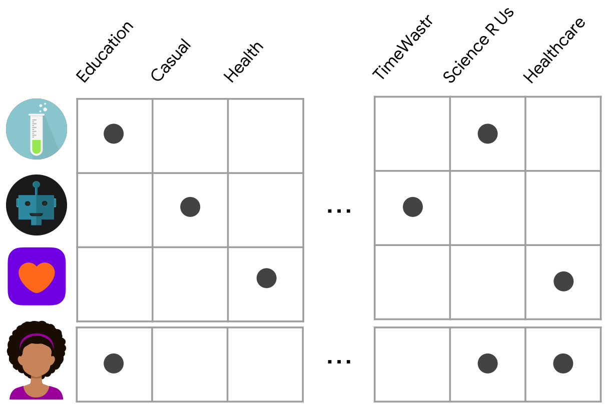


Figure 2.2.3: Embeddings in matrix notation

Source: [Goo18]

The upper part of fig. 2.2.3 shows that each application has a row vector v that represents its features. Alice's preferences are denoted in the lower part vector u of fig. 2.2.3. Note that the preferences are represented on a per-feature basis rather than a per-app basis; in this example, the matrix does *not* contain any explicit information about the actual apps Alice has interacted with. For CBF, we do not need to consider any other user and and we solely focus our view on Alice. Both vectors, u and v, are filled with binary values for simplicity.

When searching for a new app recommendation, we can now calculate the similarity based on the content information we have, e.g. by using the dot product as our similarity function $s(u, v)$ which accepts two vectors as an input (the higher the result for an app, the more similar it is to Alice's preferences):

$$s(u, v) := u \cdot v = \sum_{c=1}^{d} u_c v_c \tag{2.2.1}$$

In eq. (2.2.1), d is the *dimensionality* of our embedding space $ES = \mathbb{R}^d$. In the above example fig. 2.2.3, an explicit set of 6 features (3 categorical information or *tags* plus 3 different app publisher) is shown, but there can be far more, as indicated by the three dots. Enlarging the number can improve accuracy, but features should be engineered carefully and require domain knowledge [Goo18] (more details follow in section 2.3.2).

With a similarity *score* of 2, Alice will probably like the educational app from "Science R Us" most (first row in the app matrix). Finding similar items to make predictions is intuitive here; the more preferred features are in the app, the more the user will like it.

Finally, CBF is especially suited for *new items* that enter the system since the representation consisting of the items' feature vector is known right at the start. But CBF is *not* effective in making predictions for new users [pp. 15 Agg+16] (more details will follow in section 2.3.1).

2.2.2 Collaborative Filtering

While pure CBF utilizes similarities between items only, Collaborative Filtering (CF) follows a different approach and is in fact *collaborative* because it allows *transfer learning* between *users* to occur. For example (see section 2.2.1): Alice likes the app from "Science R Us". If Bob, another user, is *similar* to Alice, he will probably like the app, too.

First, we denote the user-item interaction matrix[1] as a $n \times m$ matrix R, with n users and m items, for instance 2 users and 3 items. In this first example, R contains *explicit feedback* data in form of star-ratings (see fig. 2.1.1) from 1 to 5:

$$R = \begin{bmatrix} 5 & 0 & 4 \\ 0 & 0 & 3 \end{bmatrix} \tag{2.2.2}$$

For instance, the first user u_1 rated item v_1 with 5 stars and v_3 with 4 stars, but did not rate the second item v_2, as we denote *unobserved* ratings with 0. Note that we do *not* capture

[1]The matrix is denoted with R because this is a common notation. While some sources accordingly use the term *rating matrix*, in this work, we consistently use the term *interaction matrix*.

interactions on a per-feature basis like in section 2.2.1. Instead, we store the items with which the users have interacted explicitly, while the item's content features itself are *latent*. We do not need to know about or incorporate any user- or item features.

This has an important advantage: It means that we do not need to carefully engineer features about the items (remember that this requires a deep understanding of the item domain, see section 2.2.1). The negative side: With CBF from section 2.2.1, we could actually recommend completely new items that no user has *ever* interacted with, because we only need the item's metadata to measure similarity. With CF, this is an impossible task, since its conceptual base always is an interaction matrix that will not have an entry for this item initially. The question is now how we can build a model which is able to learn the latent embeddings on its own simply by feeding the interaction matrix as our only input.

2.2.2.1 Matrix Factorization - Intro

One popular embedding model is Matrix Factorization (MF) which is based on the idea that R can be approximated by decomposing it into two embedding matrices; the user embedding matrix U and the item embedding matrix V. Thus, the interaction matrix R can be factorized as follows [Agg+16] [KBV09]:

$$\underset{m\times n}{R} \approx \underset{n\times d}{U} \cdot \underset{m\times d}{V}^{T} \tag{2.2.3}$$

In fact, the reason for using the *dot product* of U and the transposed V^T (also called dot product UV^T) gets more intuitive when we internalize that the model needs to capture the *relationship* between users and items from a high-level view. This relationship is also illustrated in fig. 2.2.4.

Since the dimensionality d of the latent embeddings is often chosen to be much smaller than n or m of the interaction matrix itself, MF models (also called *latent-factor models*) are typically memory-efficient, as it enables a more compact representation of R. UV^T is only an approximation which is due to the fact that $(n + m)d < n \times m$. It means that there are less entries in the embedding matrices UV than in the interaction matrix R, making it impossible to *reconstruct* exact values in every case [Goo18].

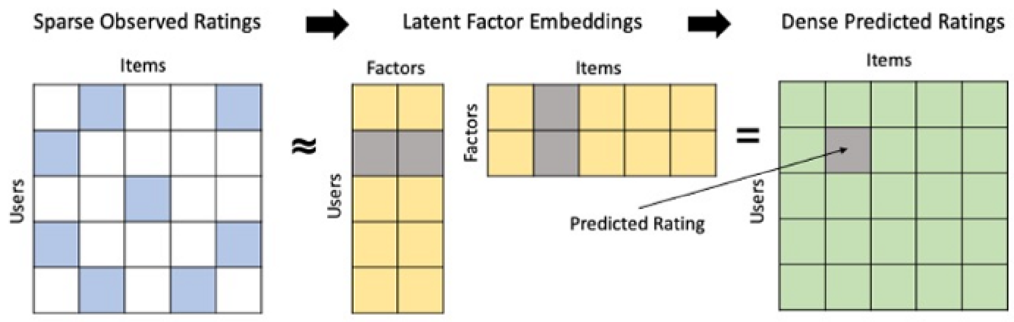


Figure 2.2.4: Matrix Factorization Process

Source: [Lun20]

2.2.2.2 Matrix Factorization - Loss and Objective

Whenever we want to find out if an approximation algorithm works well (in this context, in approximating the interaction matrix), it is mandatory to measure *loss*. In the ML world, loss is the key concept to penalize for inaccurate predictions. Reducing loss is how ML models make (better) predictions.

A small analogy: *Learning* occurs by doing mistakes that need to be observed. If an error cannot be properly identified, no learning can occur. After all, this not only applies to humans, but also to *machines*: In the ML field, learning is accomplished by adjusting weights of a function after quantifying their respective loss, optimally until a defined convergence condition is fulfilled.[2]

UV^T is an attempt to *reconstruct* the matrix R, thus we want to identify the error. This error is called the *reconstruction error* of UV^T, so let us introduce a residual matrix E [Agg+16, p. 95][3]:

$$E = R - UV^T \tag{2.2.4}$$

The residual matrix E is the difference between our real data (the interaction matrix R) and the approximation (the dot product of the embeddings UV^T). From another point of view UV^T becomes exact (=) instead of approximated (≈) by simply adding the residual matrix E:

[2]see also Goo20, for a comprehensive introduction to Machine Learning.

[3]The residual matrix is less commonly used in other sources, but in this case, it helps with introducing the concept of loss on a higher level (the matrices').

$$R = UV^T + E \tag{2.2.5}$$

This conceptually highlights how we can calculate error on a matrix level. Now, let us go the path from top-to-down, from matrix → vector → scalar-level, by expressing how to calculate a prediction for a user-item pair (i, j): Let $\hat{r}(i, j)$ be the rating prediction function for a user i to item j, which can be expressed as the dot product of the respective ith and jth row vectors of the matrices U and V, $u_i \in U$ and $v_j \in V$ [see Agg+16, pp. 94 – 97]:

$$\hat{r}(i, j) := u_i \cdot v_j = \sum_{c=1}^{d} u_{ic}\, v_{jc} = \hat{r}_{ij} \approx r_{ij} \quad \text{with } \hat{r}_{ij} \in UV^T \text{ and } r_{ij} \in R \tag{2.2.6}$$

Note that this relation is represented by the grey colored cell in fig. 2.2.4, with $\hat{r}(2, 2) = \hat{r}_{22}$. Given the definitions of our actual rating r_{ij} and the prediction $\hat{r}_{ij}$ from the previous eq. (2.2.6), one possibility to calculate the error for (i, j) is given by the following loss function:

$$e(i, j) := |r_{ij} - \hat{r}_{ij}| \tag{2.2.7}$$

This corresponds to the absolute value of entry (i, j) in $e_{ij} \in E$ from eq. (2.2.4). It is also known as L_1 loss. A more suitable and commonly used loss function is the L_2 loss (squared loss), which is defined as follows [Goo20]:

$$e(i, j) := |r_{ij} - \hat{r}_{ij}|^2 \tag{2.2.8}$$

Due to the quadratic loss, outliers (i.e. bad predictions) are much more penalized than by the linear L_1 loss in eq. (2.2.7).

Example 2.2.2: L_2 as Function Composition

Formally, loss can be defined as:

$$f(x) = \text{actual}(x) - \text{prediction}(x)$$

Therefore, L_2 loss can be composed by $q \circ L_1$, with q being the quadratic function $q(x) = x^2$, as it is a quadratic form of L_1 loss:

$$L_2(x) = q(L_1(x)) = L_1(x)^2 = |\text{actual}(x) - \text{prediction}(x)|^2$$

From a practical point of view, composition leads to abstraction and aids to make ML more understandable for humans. This concept is especially useful within the following practical chapter 3 of this thesis.

By combining the above-described mathematical building blocks, we can express a typical objective function for an RS model to learn the best values for the embedding matrices U and V over the set of observed ratings S with $r_{ij} > 0$ (in case of the initial R) in eq. (2.2.2) [HKV08][KBV09]:

$$\min_{u_*, v_*} \underbrace{\sum_{(i,j) \in S}}_{\text{Sum over obs.}} \underbrace{(r_{i,j} - u_i \cdot v_j)^2}_{e(i,j)} + \underbrace{\lambda(\|u_i\|^2 + \|v_j\|^2)}_{\text{Regularization}} \tag{2.2.9}$$

Minimizing the loss of reconstructing the interaction matrix only over the observed values as formulated above comes at a cost: Without any *regularization*, the model would likely fit very well to the supplied training interaction matrix R, but overall it would generalize poorly for new (predicted) user-item interactions. Therefore, as formulated in eq. (2.2.9), the additional cost to compensate is a regularization term, which is added to the error term $e(i, j)$. The choice of λ in this term is dependent on the dataset and as a *hyperparameter*, it needs to be fine-tuned. In general, this is a commonly used approach to prevent overfitting and it allows the model (some) additional space to generalize. The model representation above refers to an *explicit feedback model*, which for example has been used similarly during the 2009 public *Netflix Prize* competition's best RS model for movie recommendations [KBV09]. For *implicit feedback models*, there are further considerations to be made which are handled in section 2.2.3.2.

Minimizing the objective function in terms of the embeddings as in eq. (2.2.9) is typically done using Stoachastic Gradient Descent (SGD), which is a common choice for ML models and enables gradually updating u_* and v_* for each so-called *epoch* until convergence is reached. Describing this central ML technique is out of this thesis' scope and further details can be obtained from [Agg+16, p. 100] and [Goo20] which algorithmically and illustratively explain the process.

2.2.2.3 Summary

The key advantage of CF over CBF is to enable predictions on how likely a user wants to interact with an item by utilizing existing interaction data from other users. This is in direct contrast to CBF (as described in section 2.2.1), because with CF we are able to make predictions for users with only few interactions, but not for *new items* that no one has ever interacted with, where no collaborative information is available.

Moreover, as opposed to CBF, Collaborative Filtering has the advantage of producing more diverse and less obvious recommendations by utilizing the other users' (neighbours) feedback, too, instead of just considering a single user's preferences in isolation [Agg+16, p. 140]. Hence, MF algorithms became very popular after their success has been established during the *Netflix Prize* challenge [KBV09].

Example 2.2.3: Importance of Diversity and Novelty

Consider the following scenario: Bob wants to watch a new movie on his favorite streaming platform. Recently, he watched *Star Wars I* and *Star Wars II*, but also *Matrix I*. He liked both movies. A CBF-based approach might recommend other movies with similar attributes (e.g. "Science Fiction" and "Action"), they could even recommend *Star Wars III* and *Matrix II*. To Bob, these recommendations are quite obvious. Do they really help Bob with his choice or do they bore him and simply show what he already knows? By contrast, in a CF-based approach, there are multiple users involved in the recommendation process. One of them is Alice, who recently interacted with *Star Wars I*, too, but also rated *The Shawshank Redemption* with 5 stars. The system learns that Alice is similar to Bob and therefore, it recommends *The Shawshank Redemption* to Bob. This is the more diverse recommendation. With a bit of luck, Bob will like it – even though the movie itself is quite different! In other words: The CF system helps Bob to expand his horizon, even though the system does not know about the fact that the movie is out of his "comfort zone".

2.2.3 Hybrid: LightFM

There are multiple different approaches to combine CBF and CF. In this work, we focus on the implementation of *LightFM* introduced in [Kul15], a hybrid content-collaborative model that is used in productive systems. In the real-world, interaction data is often very sparse. For instance, there are millions of products in a large online-shop, but a user only interacts with a fraction of them. This sparsity makes it very difficult for a model to generalize well [Goo18].

Therefore, the main motivation behind the project is to tackle the new user/item-problem within such enterprise use-cases. This is also called the *cold-start* problem, which we faced before (though it has not been named explicitly), at the end of section 2.2.1 as well as section 2.2.2. It can either occur when a new user or a new item enters the system and is a general challenge of RSs that is described more differentiated in the following section 2.3.1.

LightFM outperforms both pure CBF and pure CF approaches by combining the best of both worlds: It uses Matrix Factorization with embeddings for CF as described in the previous section 2.2.2, but instead of using plain user/item vectors as latent factors, metadata (i.e. *content features*) is *encoded* within these embedding vectors [Kul15].

2.2.3.1 Embedding Metadata Features

The *embedding of metadata* is the essence of *LightFM* as it enables to feed metadata into a classic MF approach. In the following step-wise example, let's assume that the user Alice from section 2.2.1 is defined by the following metadata:

Table 2.2.1: Alice's Metadata Features

Feature	Value	Example Encoding
Gender	Female	$(0\ 1)$
Age	Young Adulthood, 18 - 24	$(0\ 1\ 0\ 0\ 0)$
Preference	Education	$(1\ 0\ 0)$

Table 2.2.1 shows the feature type, value and the actual vector representation which is the model's input. For the "Representation" column, we performed some *Feature Engineering*: The features are not fed into the model directly as raw values (e.g. the string "female"), but they are rather preprocessed. In our case, we applied a process called *one-hot encoding*, which

is suited for representing categorical data [Goo20].[4] The first row's tuple of all possible values is $(male\ female)$ and therefore $(0\ 1)$ translates to $male = 0 = false$ and $female = 1 = true$ (for simplified explanations, we focus on the binary gender case in this example). For the second column, we also used a technique called *bucketing* by not describing the age of the user directly but rather dividing it into a total of 5 categories so that we can once again one-hot encode it. In Alice's case, "Young Adulthood" is set to 1, which is the group after "Childhood". The decision which buckets to use is entirely up to the ML Engineer and highly domain-specific. Analogously, the three preference categories $(Education\ Casual\ Health)$ are one-hot encoded.

In section 2.2.2.1, the Matrix U describes all users' latent factors. One row within this matrix represents *one* user's latent factors, which will be denoted as vector x_u in the following. In the bare MF model, no content information has been encoded into this vector. The model needs to learn the semantics on its own by reconstructing the interaction matrix by UV^T. We want to change that and embed *content* information, too: Formally, each user u has an own set of metadata features f_u from the complete user feature set F^U, i.e. $f_u \subset F^U$ describes one specific user's metadata [Kul15]. First, each of these features has its own latent embedding vector e_j^U of a specified dimensionality d specifically assigned to user u. Second, we assume that adjusting the vector u by the sum of these latent vectors describes the latent user in total, represented as q_u [Ros16][5]:

$$q_u = x_u + \sum_{j \in f_u} e_j^U \tag{2.2.10}$$

In eq. (2.2.10), we clearly distinguished the latent user representation x_u from the metadata features e_j^U of the user. In fact, in a *LightFM* model, *everything* is treated as features (or metadata) to simplify the model, even the user/item representations themselves and thus the equation becomes [Kul15]:

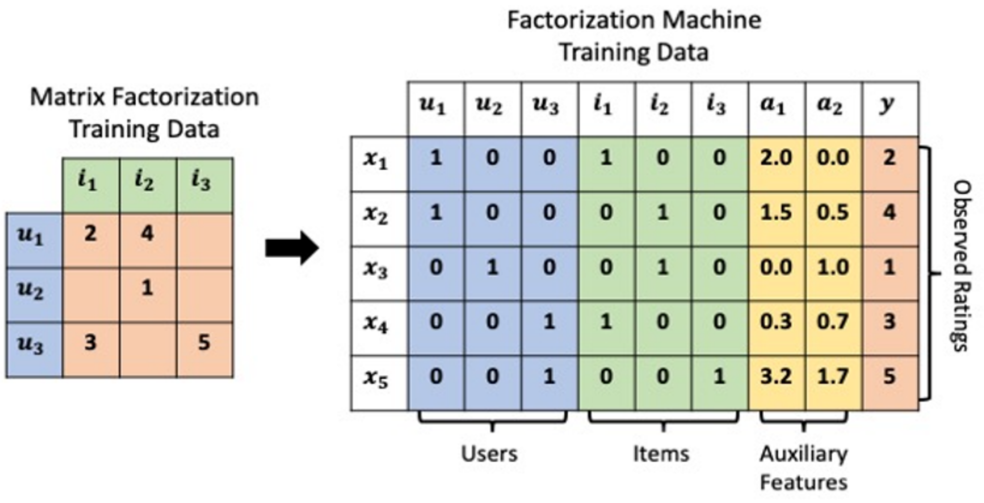
$$q_u = \sum_{j \in f_u} e_j^U \tag{2.2.11}$$

For items, the same approach is used to represent its metadata features $f_i \subset F^I$) [Kul15]:

[4]ML algorithms often require preprocessing and typically work with numerical data only.
[5]notation adapted from Kul15.

$$p_i = \sum_{j \in f_i} e_j^I \tag{2.2.12}$$

In the example table 2.2.1, Alice would be described by the sum of the *latent* representations of `(Female, Young Adulthood, Education)`. Often, it can be problematic that the tuple is not unique to Alice, as it can describe any user with the same three attributes similarly. To distinguish between users/items and to offer more personalized recommendations, the following approach is used: An *indicator variable* is introduced which uniquely identifies each user and each item. This identifier is one-hot encoded and fed into the model as a feature on a per-user/per-item basis. The indicator is the feature that makes *LightFM* reduce to a classic MF model (traditional CF) in cases where no content information is given or available. However, in cases where indicator features are missing and only content information is provided, the model does *not* fallback to a classic content-based model. This is because *LightFM* always works by decomposing the interaction matrix into collaboratively available factors. Therefore, *LightFM* is always based on CF, with or without indicator variables and with or without content features. [Kul15].


Matrix Factorization
Training Data

	i_1	i_2	i_3
u_1	2	4	
u_2		1	
u_3	3		5

Factorization Machine
Training Data

	u_1	u_2	u_3	i_1	i_2	i_3	a_1	a_2	y
x_1	1	0	0	1	0	0	2.0	0.0	2
x_2	1	0	0	0	1	0	1.5	0.5	4
x_3	0	1	0	0	1	0	0.0	1.0	1
x_4	0	0	1	1	0	0	0.3	0.7	3
x_5	0	0	1	0	0	1	3.2	1.7	5

Users
Items
Auxiliary Features
Observed Ratings

Figure 2.2.5: Matrix Factorization and Factorization Machine training data

Source: [Lun20]

By approximating the preferences as a sum of latent vectors, *LightFM* can be seen as a specialization of a Factorization Machine (FM). In such a machine model, interactions between n

variables can be estimated [Lun20] (see fig. 2.2.5) and it is not restricted to the more specific form of user-item feature variables in case of the *LightFM* model. Because most RSs do not need an n-way interaction model, *LightFM*'s restriction aids the simplicity [Kul15] which is a good trade-off for the versatility of a fully-featured FM.

2.2.3.2 Implicit Feedback Specialization

In eq. (2.2.9), we built our model based on the explicit feedback matrix R from eq. (2.2.2). However, in production systems, implicit feedback is often more widely available [HKV08] and considering observed explicit ratings only ignores the fact that missing entries potentially carry useful information [Ste10] [Kul18] [Lun20]. The MF model in eq. (2.2.9) did not particularly handle unobserved entries other than through a naive regularization term. In the definition, we added this term to generalize better and not overly fit the embeddings to the trained (observed) data only, i.e. we penalized the model's ability to adjust "too well" to the training data. We therewith indirectly assumed that the unobserved entries do *not* correlate with the users' preferences, which rarely holds true. For instance, if Alice likes a learning-app she is probably more invested to rate the app with 5 out of 5 stars. Often, users leave ratings because they *strongly* like (or dislike) a particular item [Ste10]. Users *choose* to do so. In reverse, this means that *if* ratings are not *given* at random, ratings can not be *missing* at random. Therefore, even though the exact mechanics behind unobserved entries are unknown, they should rarely be excluded from the model's training. In fact, unobserved entries can carry useful information just as their counterpart [Ste10] [Kul18].

With implicit feedback, we assume that observed entries indicate a degree of confidence that a user likes a particular item, without the need of an explicit negative feedback mechanism. This dramatically improves the wide availability of interaction data over explicit data, where a typical user only rates a few items at most [Ste10]. For instance, if Alice opens a learning-app ten times a day, she probably likes the app, even without her explicitly stating it. Furthermore, it means that she is probably less interested in other applications that she only opens once in a month, as every user has a limited time-attention span. By incorporating the observed *and* unobserved entries in the model directly, *LightFM* is a suitable model for implicit feedback data [Kul15] [Kul15, see eq. 2] and as such, the objective is not to predict a rating, but rather to learn how to *rank* based on the *confidence* of the interaction data [Ros16], which arguably increases if a user often interacts with a particular item.

2.2.3.3 Summary

Ultimately, combining the latent vectors makes it possible to jointly learn similarities between users, items and their metadata features, which leads to improved predictions in both, warm- and cold-start as well as high- and low-sparsity scenarios [Kul15]. Simultaneously, the model allows room for more comprehensive interpretations in the latent feature space than classic CF models where no metadata is embedded and semantic information is unlabeled (no input feature vectors are given). For instance, if Alice is the dominating user of educational apps, the embedded feature vectors of "Education" and the representation of herself (see eq. (2.2.11)) will be close together (user-feature to user). Likewise, if applications from the categories "Education" are used by many "FH Aachen" students, the model will learn that they are similar (feature to feature). This relationship is illustrated in fig. 2.2.6, as a concretization of fig. 2.2.2 from the preliminary section 2.2.

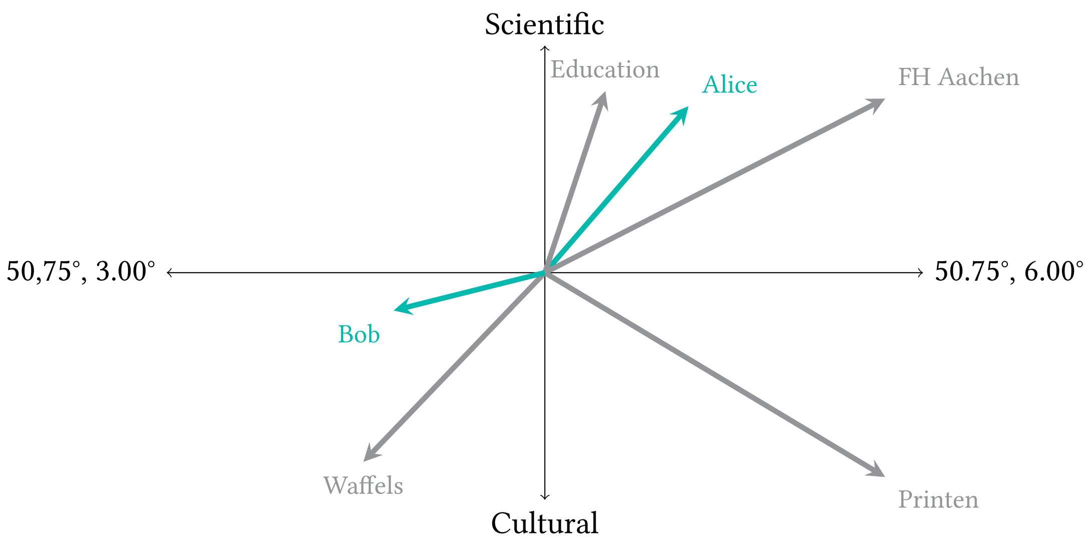


Figure 2.2.6: Users/Features represented in a 2D embedding space

2.3 Challenges

In the following, four of the most important challenges for the implementation of RSs are introduced. The focus is on *Hybrid* RSs, but also pure CBF and CF approaches are considered where appropriate to deepen the understanding. The challenges are ordered in a way that each challenge leads to another (the arrow reads as "requires"):

Tackling Cold-Start $\Longrightarrow$ Feature Engineering $\Longrightarrow$ Fairness $\Longrightarrow$ Explainability

2.3.1 Tackling Cold-Start

As already briefly introduced in section 2.2.3, the cold-start problem poses a serious challenge to RSs. It occurs whenever sufficient feedback through interactions is missing (or only few interactions exist) and therefore no accurate or reliable recommendations can be made [Bob+11]. One typically differentiates between two types of cold-start [Bob+11, p. 2]:

1. ***New item** cold-start*: A new item enters the system. When using a CBF-based or hybrid RSs, the existing attributes (or features) of the item can be used: If the user has interacted with an item with similar attributes in the past, the new item might be similarly liked by the user. Therefore, CBF-based RSs take advantage of the existing per-attribute preferences to make recommendations even for completely new items with no interaction history [Agg+16]. To the contrary, in pure CF, no metadata about the item is involved. Therefore, it becomes hard to make recommendations for items that no user has ever interacted with (see remark at the end of section 2.2.2), because this also means that there is no collaborative information available.

2. ***New user** cold-start*: A new user enters the system. Here, CBF-based approaches are ineffective because at the very initial state, a user did not interact with any item. Therefore, no item preferences are known at this time. CF is also equally ineffective to handle new users, since the user is not known during the model's training phase or because of missing interactions, no inference is possible [Goo18]. Instead, this type of cold-start is manageable by using a hybrid RS as introduced in section 2.2.3: If the user enters only a few information about herself/himself (user features), the system is able to use the collaborative information (by relying on the metadata of similar users). With *LightFM*, this works without the user being present during training, because the metadata features explain part of the user/item structure (see section 2.2.3).

The most drastic form of the cold-start problem is the complete lack of interactions (also called *new community problem*) [Bob+11] which takes place at the initial state of the recommender system itself. To collect initial information, users can e.g. be encouraged to interact with (random) items that are suitable based on domain-knowledge [see also Agg+16, pp. 16, knowledge-based RS]. In the end, all ML-based models need training data to learn with, which is why the cold-start problem is still one of the biggest challenges of RSs. Additionally, the training features needs to be carefully prepared, leading us to the next challenge.

2.3.2 Feature Engineering

Particularly important for a RS's ML model to perform well is the *collection and selection*, *cleaning and preparation* of data to use as inputs for training the model. This process is commonly called *Feature Engineering* [Ren19]. We engineered features "on-the-fly" in section 2.2.3, table 2.2.1 (bucketing and one-hot encoding). However, real-world feature engineering is a complex and significantly time-consuming task [Ren19], with surveys indicating that data scientists spent up to 80% of their time on it [Pre16].

For a hybrid RS, feature engineering itself is not only a regular process but also a challenge. Content features for items and users are needed and their quality are the foundation of good (especially cold-start) recommendations. The basic steps of feature engineering are as follows [Goo20] [Ren19]:

1. **Collect and select features**: Collect which data is/will be available to the recommender system. Typically, a person with the necessary domain knowledge needs to decide which features are important to capture, e.g.: Is the user's age important for the predictive power of a model? Does it make sense to use the item's country?
2. **Clean the data**: Without cleaning the data, the model potentially learns based on a false premise. Therefore, invalid/incomplete/unavailable information needs to be filtered out or handled accordingly.
3. **Prepare the data for inputting to the model**: Data can only rarely be used as model input without a conversion. For example, the raw value of the user's age does not necessarily make the model more expressive. Especially in smaller datasets, bucketing into semantic age groups is more suitable (as done in section 2.2.3) for the model to generalize well. In general, the feature values need to be numeric so that they can be

used as a feature vector input [Goo20]. Non-numeric data such as the item's country needs to be converted (for example via one-hot encoding). There are many other data preparation techniques which are out of this thesis' scope. They are described in a practical approach in [Ren19] which for instance describes how to handle outliers or how to extract dates.

Utilizing user and item features is typically an important step to mitigate the cold-start problem as described in the previous section 2.3.1. To help with the *user cold-start* problem, users can be asked to enter personal information in an on-boarding process during registration, where it is mandatory to enter profile information such as age, bio or favorite activities. The RS engineer has to find a compromise between the predictive power of the information and the amount of entered data: Comprehensive information can be collected and converted to useful features and the model can potentially get more expressive to deliver novel recommendations [Agg+16, p. 233]. On the other hand, if the system asks the user to enter a lot of information for the registration to be completed, the user could easily get frustrated and even quit the application, thus resulting in loss of a potential customer. Moreover, each feature adds an additional layer of complexity to the model and practically to every step of RS engineering.

For this reason, feature engineering is a deep- and wide-ranging challenge for RSs, since finding the optimal balance between feature-usability and amount of information is non-trivial and requires a broad spectrum of experiments as well as a deep technical and domain-specific knowledge, let aside the psychological perspective of mitigating the user's potential negative mindset (such as privacy concerns) for instance by a transparent data protection policy and/or a motivational User Experience (UX) design. The choice and preparation of features implies ethical consequences, which are described in the next section.

2.3.3 Fairness

Whenever a RS makes a recommendation, it should be as fair as possible. Even more general, an ML model should be completely objective. This arises the question of what the definition of fairness is in the context of ML and how to identify it. One possible interpretation is that fairness is the minimization (or ideally elimination) of *bias*. However, in the world of ML, bias is a broadly used term that can have many different meanings [see HDB20, for a selection]. In the following, we therefore focus on *popularity bias* as an example for a specific kind of

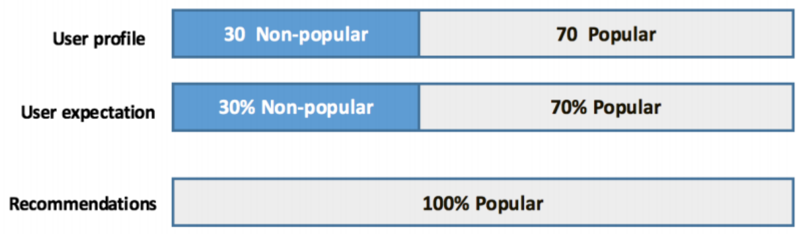


Figure 2.3.1: Inconsistent recommendations through popularity bias for a diverse user

Source: [Abd+19]

bias that happens during the RS learning stage and is in fact manifested in many RS models themselves, which means that it usually is *not directly* caused by humans. As such, it can be seen as *learning bias* (also called *inductive bias*).[6]

Popularity bias has been extensively analyzed in [Abd+19]. This bias leads to items that are less popular being less recommended, while popular items preserve their position and are recommended all over again, see fig. 2.3.1. For users (as well as other stakeholders), this can be unfair and problematic on two counts that are highly correlating [Abd+19, pp. 1 ff.]:

1. Expectation vs. Recommendation: Users who might have interest in less popular items ("niche group") or a diversified opinion regarding popularity ("diverse group") can never be fully satisfied and only receive popular recommendations.
2. Lack of Diversification: The RS's environment will be dominated by popular items with certain properties. In a exemplary marketplace, (newcomer) sellers will try to imitate the popular products and their creative limits are bounded to the market leaders. Users also do not have the possibility to find new innovations easily due to the pre-dominance of popular items which are recommended (see also example 2.2.3 where we provided a look upon diversity for CBF and CF). From a socioeconomic perspective, merchants who sell popular products (market-dominating brands) will always be preferred so recommendations of smaller sellers with niche products do never have the chance to enlarge their customer base even though their products might be much more innovative or personalized to the user [Abd+19].

Overall, while the challenges of fighting unfairness are often not completely solvable, establishing an *awareness* that specific types of (*human*) biases exist makes it possible to identify

[6]Terminology proposals and information regarding the ambiguity of *bias* can be read in [HDB20].

them and react accordingly while developing a RS [HDB20] [Goo20]. To support the implementation of fairness in an organization, one could for instance introduce a dedicated role, i.e. an independent person who supervises and exposes potential ethical flaws and biases within the system. Striving for an independent supervisor is important because the person who is responsible for the technical realization of system might be unconsciously biased, which could depend on various factors (e.g. only concentrating on optimal model evaluation results and ignoring ethical flaws of the model).[7]

The next challenge of explainability is closely coupled to fairness, since users will ask for explanations when they e.g. suppose that the system is biased in any direction by receiving unfair recommendations [GF17].

2.3.4 Explainability

Explainable AI is an emerging field [KRK18] that is also manifested in regulatory bodies' protection guidelines such as the EU's "Right to Explanation" [GF17]. Humans need explanations for recommendations to *trust* their RS environment [Agg+16, 232 ff.] [KRK18] [GJG14]. Providing explanations also means that the user is more likely to accept the recommendation [KRK18].

Moreover, as analyzed in the human-computer study [GJG14], explanations can serve a variety of goals besides trust, most popularly among them: *Satisfaction* (the user is satisfied with the recommendation and enjoys using the system) *Transparency* (the user and others can retrieve insights about the RS algorithms to better understand how they work), *Efficiency* (the user needs less time to perform a task when reading the explanation), *Effectiveness* (the user makes better decisions when completing a task based on the explanation), *Persuasiveness* (the user's behavior can be changed by proposing new solutions) [GJG14].

Establishing trust is – as a multi-faceted goal – very difficult to measure and estimate [GJG14, 378 f.], but a main driver behind long-term users which should be considered during the whole RS development process. Interestingly, trust and usefulness are not complementary to each other, because the user probably trusts *obvious* recommendations (see example 2.2.3), but nonetheless they do not enable the user to discover a broad, diverse spectrum of items [Agg+16, p. 233].

[7] see also [HDB20] and [Goo20, chapter "ML Engineering - Fairness"]

Transparency is divided in *perceived* (subjective) and *real* (objective) transparency. As the former concentrates on making the transparency understandable to the user (sometimes not being in compliance with the underlying RS algorithms), the latter describes the internals of the RS accurately, but is more difficult to understand (RSs are complex, highly technical systems) [GJG14]. Therefore, we concentrate on the *perceived transparency*, though it is important to provide truthful explanations for ethical reasons, but also as users otherwise lose their *trust* in the system.

The following has been found out regarding the dependency of the goals in [GJG14]: The goals *transparency* and *effectiveness* increase (long-term) user *satisfaction*. Making the user happy is considered is a significant contributor to trust [GJG14], thus creating a positive feedback loop which enables content long-term users that are considerably easier to *persuade* to make behavioral changes (e.g. changing buying behavior).

When it comes to choosing the most suitable explanation type for specified goals, there are are multiple dimensions to consider, some of them have been extracted and are described in the following [GJG14]:

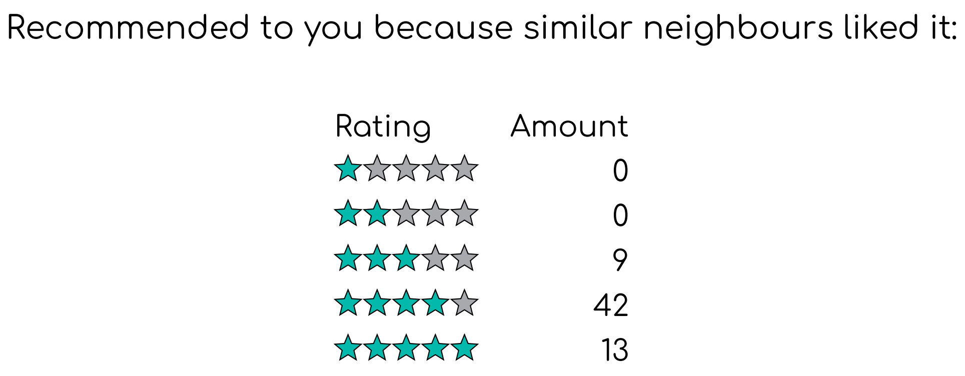
1. **Privacy Level** - *personalized* vs. *public*: Recommendations can either be justified by personalized explanations based on private user-information such as "This book is recommended because you liked *A Song of Ice and Fire*" or based on publicly available information such as "This book is recommended because it has an average rating of 4.2 stars by 7300 users"

2. **Information Type** - *content* vs. *collaborative*: The explanation interface either presents content information (e.g. *tag cloud* and *related (liked) items*) or collaborative information (e.g. *average rating*, *similar user ratings*). A special type is the use of performance data about the recommendation itself, such the recommendation *confidence* which is the probability the model has for the item to be recommended, i.e. how likely it will match the user's taste.[8]

3. **Presentation Type**: There are many ways to display explanations, some example classes include *text-driven* (text-explanation or word-cloud), *diagram-driven* or *graphic-driven* (e.g. pie chart) and *tabular*.

[8]see table 2 GJG14, for more explanation interfaces.

Consequentially, transparent explanations depend on the used algorithm that provided the recommendation. CBF makes it easier to use *content-based* information than CF as these are already provided by the model itself (see section 2.2.1). In pure CF, this type of information needs to be extracted outside of the RS, e.g. by collecting information about the recommended items' content data from another service. On the other hand, CBF is limited when it comes to collaborative information such as the "neighbourhood" of similar users/items. This is where the strengths of CF are, as shown in fig. 2.3.2 where a *personalized* explanation can be provided by including the similar users' ratings (the "neighbourhood" for a specific item). Similarly as before, a *hybrid RS* comes with the advantage that two sources of information can be utilized (and correlated) for the explanation, often without the need of retrieving additional data such as tags from another external service (i.e. a service that is outside of the RS environment). For instance, when using *LightFM*, we can estimate the similarity of two items by calculating the distance between their features' factors [Kul15, p. 6] (e.g. by using cosine similarity). A recommendation which is explainable by feature/tag similarity is e.g. "We recommend you *It* written by 'Stephen King' because you recently watched a 'Horror' movie" in which the RS learned that 'Horror' and 'Stephen King' are closely related to each other.


Recommended to you because similar neighbours liked it:

Rating	Amount
★☆☆☆☆	0
★★☆☆☆	0
★★★☆☆	9
★★★★☆	42
★★★★★	13

Figure 2.3.2: Example interface for `neighborsrating`, adapted from [GJG14]

As there are many possibilities to design an explanation interface, fulfilling one goal may result in a trade-off between others [GJG14]. For instance, fig. 2.3.2, does not necessarily show a suitable solution for user satisfaction, as it has been found that this interface (namely `neighborsrating`) does not satisfy the user, although it offers a good level of transparency [GJG14, p. 376]. This is can be linked to a) the user failing to understand the meaning of similar users [GJG14] or b) the relatively complex presentation which might confuse the user by showing "too much information". To alleviate b), a relatively simple solution is to exchange the presentation type to a diagram, thus making the data more easy to understand (and likely more visually appealing). This observation is also backed by [GJG14, p. 376], which shows a rise in the user's satisfaction just by using a bar chart[9] instead.[10]

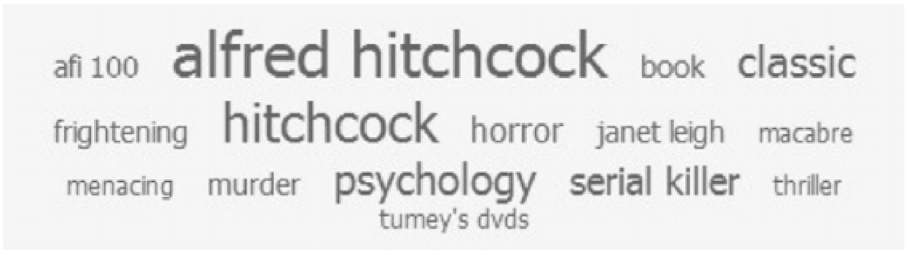


Figure 2.3.3: Example tag cloud for a movie recommendation explanation

Source: [GJG14]

Using a well-known *tag cloud*[11] explanation (see fig. 2.3.3) is considered to be highly effective in transparently showing how internals of the RS came to the conclusion that the recommendation is suitable, even though users needed the most time to process this type (worst measured efficiency), they were most satisfied with the personalized variant of it [GJG14].

Summarizing, choosing the right explanation interface depends on many factors (such as available private/public data, information type and presentation type), which should be carefully weighted dependent on the set goals. Ultimately, to help engineers with setting up a proper explanation type, [GJG14] proposes guidelines which can be summarized to the following three points:

1. Use domain-specific content to increase *effectiveness.*
2. Use familiar explanation types to improve the user understanding, which leads to higher *transparency* further contributing to long-term *user satisfaction.*
3. Prefer *transparency and effectiveness* over *efficiency.*

[9]Diagram which uses one bar per star rating on the x-axis and the amount of users on the y-axis.
[10]further reading available on pp. 374 ff. GJG14.
[11]Visualization where keywords are arranged alphabetically, with the font size indicating relevancy.

2.3.5 Summary

In this section, we examined a chain of challenges that we consider to have an enormous impact on the design of recent modern RSs [Fal19]: Tackling the *cold-start* problem (section 2.3.1) is necessary to recommend straight away from the start if the user or item is completely new to the system. The challenge of proper *feature engineering* (section 2.3.2) is a direct consequence of that; hybrid solutions that make the cold-start problem (more) manageable often require an enormous amount of work of this essential ML technique. This is followed by the considerations on fairness (section 2.3.3) that are practically relevant in every engineering step, but especially when *decisions* on specific features (e.g. during feature engineering) and/or training data are made. In the end, with *explanations* (section 2.3.4), we not only discovered one of the recent directions of ML and RSs, but also a technique that makes recommendations much more transparent to the user, thus ensuring the important goal of long-term user satisfaction.

3 Reciprocal Recommender Systems

In this chapter we first introduce Reciprocal Recommender System (RRS) and provide a formal definition that needs to be fulfilled by any RS to make *reciprocal* predictions (section 3.1). Subsequently, we describe the specifics of RRSs. This is followed by a comparative overview between RRSs and traditional RSs in section 3.2.

Afterwards, we are able to contextualize the previously described challenges in section 2.3 for RSs in section 3.3 for RRSs. Furthermore, we use these to have a guideline that can be used to derive the requirements and the objective for a multi-purpose RRS framework (section 3.3.2).

3.1 Introduction

After reading chapter chapter 2, one could assume that, to make effective user-to-user recommendations, it would be sufficient to trivially exchange *items* with *users* in our definitions. This holds true for recommendations where the other user has conceptually no influence in the outcome of an interaction. For instance, following a user on *Twitter* does not require any *consent* [Pal+20, pp. 2-5]. To the contrary, the "follow interaction" on *Instagram* is more sophisticated:

- Following a user with *public* profile: No consent required, allows the user to follow straight away. The interaction is always successful.
- Following a user with *private* profile: The user with the private profile needs to accept the follow request for the interaction to be successful.

In fact, whenever the other user's consent is required in a system, the *reciprocity* becomes important. As it now becomes clear, the root of *reciprocal* recommendation and one approach to differentiate between RSs and RRSs is once again: interactions. Furthermore, RRSs are as an extension thereof *inherently* more complex than traditional RSs [Pal+20], as they require both parties to accept in order to be successful.

The concept of a Reciprocal Recommender System has been introduced by Pizzato et al. in the year 2010. At this time, only very few sources highlighted the importance of reciprocity. The research in this field was only sparse [Piz+10a], with many sources focusing on user-to-item recommendations only – even though the need for reciprocal recommendations has always been existent, since the beginning of RSs and human-to-human online interaction [Pal+20]. The few sources that also covered aspects of reciprocity were only very basic [Piz+10a], e.g. outlining that implications such as "A likes $B \implies B$ likes A" generally do not hold true, being more on a psychological rather than technical level.

The field of RRSs is still emerging. Very recently in mid 2020, a snapshot paper has been published by Palomares et al. in [Pal+20] which provides a bird's-eye view over the state-of-the-art RRS landscape from an algorithmic and scientific perspective. This is useful throughout the whole engineering process of an RRS, for instance to efficiently identify the broad spectrum of opportunities and their success probabilities beforehand – in one view.

3.1.1 Definition

Pizzato et al. first defined the RRS formally and established a common wording to describe a RS of this specific type. Interestingly, in the first definition, the term "item" is used to describe the human counterpart which is to be recommended to the other human user [Piz+10a]. To be as clear as possible, we define the user whose intent is to receive recommendations as the *subject*, and the recommendation as the *object* (put into grammatical context - "subject receives object recommendation"). This is a well-established convention [Pal+20], though as pointed out in the common literature, one can view the problem symmetrically, because all users are potential recommendation objects and recommendation subjects [Agg+16, p. 443].

Initially, we introduce a user-to-user RS that works in a non-reciprocal way [Pal+20, p. 5] [Piz+10a, p. 5]. Note how this also represents a traditional user-to-item RS, since we only consider user u's preferences towards the objects v and w (given by the preference function p), with R representing the list of recommendations:

$$RS(u) := \{v \,:\, p(u, v) > p(u, w)\ \forall v \in R,\ \forall w \notin R\} \tag{3.1.1}$$

The outcome is a set of optimal objects to be recommended $v \in R$ that fit to the preferences $p(u)$ unidirectionally.

For reciprocal recommendations, we also need to consider the preferences of the object v, which can be achieved by combining both RSs [Pal+20, p. 5]:

$$RRS(u) := \{v \, : \, v \in RS(u) \text{ and } u \in RS(v)\} \tag{3.1.2}$$

The *RRS* in eq. (3.1.2) considers the preferences of both users – therewith, the subject u can receive a recommendation that is reciprocal, i.e. also in the interest of the recommended object. More extensively, a typical *RRS* often resolves to the following [cf. Pal+20]:

$$RRS(u) := \{v \, : \, rp(p(u,v), p(v,u)) > rp(p(u,w), p(w,u)) \; \forall v \in R, \; \forall w \notin R\} \tag{3.1.3}$$

Both users' preferences are aggregated by the reciprocal preference function rp as denoted in eq. (3.1.3).[1] This key concept is illustrated in fig. 3.1.1, which directly leads us to the next section, describing the process of preference aggregation in more detail.

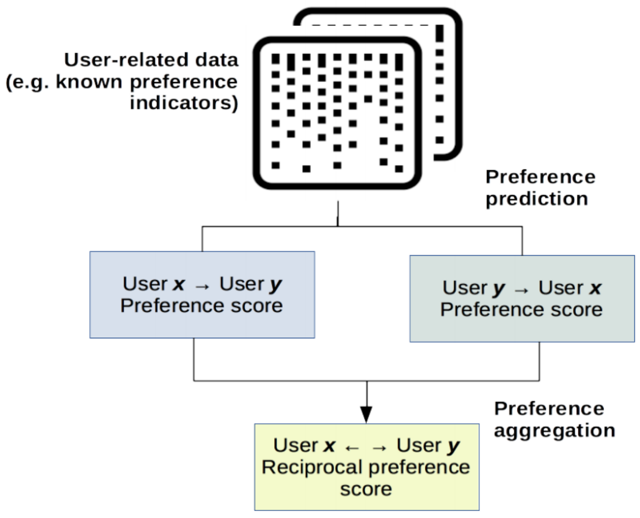

User-related data (e.g. known preference indicators)
Preference prediction
User x → User y Preference score
User y → User x Preference score
Preference aggregation
User x ← → User y Reciprocal preference score

Figure 3.1.1: Conceptual view on RRSs

Source: [Pal+20]

[1]Sometimes also referred to as ϕ in literature [Pal+20, p. 5]

3.1.2 Specifics

In this section, we highlight the important specific properties in the field of RRSs.

3.1.2.1 Preference Aggregation

Whenever we calculate two preference scores as formulated in eq. (3.1.2), it becomes necessary to fuse the recommendations in order to get the final reciprocal ranked result of $RRS(u)$, which can e.g. be an ordered list for the top k reciprocal recommendations for u. Therefore, we specify that the preference function p outputs a numeric score s in the range of 0.0 (min. affinity) to 1.0 (max. affinity) describing the likelihood that a user u will like another user v.

There are different techniques to accomplish the aggregation of the preferences. Historically, for RRSs, one of the first ideas was to use a simple sum of weighted scores [Piz+10a, p. 6], with s denoting the preference scores of the users:

$$rp_S(s_u, s_v) := w_u s_u + w_v s_v \tag{3.1.4}$$

The weights are used as an option for customizability. Setting both w_u and w_v to a constant (such as 1) will grant true reciprocity that handles the subject's and the object's preferences equally. Hence, using unequal weights will increase imbalance and lessen reciprocity. For instance, setting w_u higher than w_v will prioritize the subject's preferences. This is useful when the subject wants to retrieve recommendations that are "more personalized" [Piz+10a, p. 6], even though it might lessen the chance of sustainable success with the object (successful reciprocal recommendations require consent).

Nevertheless, linear functions such as in eq. (3.1.4) are often too simple to properly capture mutual interests, because ignoring the discrepancy between the preference scores often is a fundamental disadvantage, e.g. if s_u is significantly lower/higher than s_v. In reciprocal settings, it is more appropriate to satisfy both users' preferences *sufficiently* [Pal+20]. Additionally, it is often wished that the function always inputs and outputs a value in a specified range, e.g. $[0, 1] \times [0, 1] \to [0, 1]$.

For this reason, other aggregation functions have been studied, suggesting that the *harmonic mean* often is the most suitable choice over the other Pythagorean means (arithmetic and geometric) [Pal+20] [NP19a]. The harmonic mean is adapted as follows for rp:

$$rp_H(s_u, s_v) := \frac{2}{s_u^{-1} + s_v^{-1}} = \frac{2}{1/s_u + 1/s_v} = \frac{2\, s_u\, s_v}{s_u + s_v} \tag{3.1.5}$$

The harmonic mean as formulated in eq. (3.1.5) proved to be useful because of its property to output a result closer to the minimum of its input values. This enforces that the smaller value has a greater influence over the score [NP19a], i.e. it penalizes the discrepancy between the users' preferences. The mutual interest must be manifested in the aggregation function, because the success of recommendations depend on it (see table 3.2.1).

3.1.2.2 Single-class vs. Two-class

In some sources, RRSs are separated into two types that are found in different reciprocal applications (also called Reciprocal Environment (RE)), *single-class RRSs* and *two-class RRSs* [Pal+20, p. 7]. Table 3.1.1 distinguishes between the two: In the former, users are not separated at all (commonly found in social network REs or on skill-sharing platforms) and in the latter, users are divided into two sets (commonly used in recruiting REs).

Table 3.1.1: Classes of RRSs

Type	Description	Environment
Single-class	Users are in one homogeneous set	Dating, Social Networks, Skill-Sharing (e.g. Learning/Research)
Two-class	Users are divided into two disjoint sets	Heterosexual dating, Recruiting

More formally, in a single-class $RRS(u)$ as defined in eq. (3.1.2) with $u \in U$ *any* user $v \in U \setminus \{u\}$ can potentially be recommended. Opposing, in a two-class RRS, the user set U is divided into two disjoint sets U_1 and U_2 ($U_1 \cap U_2 = \emptyset$) [cf. Pal+20, p. 6 ff.].

Table 3.1.1 shows that "dating" is present in both types, but a two-class RRS is more restrictive in this environment, because it only enables strictly heterosexual recommendations (e.g. male and female set). To the contrary, in a single-class RRS, users are all in one set and non-binary dating becomes possible, too – without per se excluding heterosexual/binary dating.

Concluding, the choice between single- and two-class RRS is highly dependent on the specific use-case and the RE.

3.1.2.3 Symmetric vs. Asymmetric Interaction

For RRSs, we define two different types of interactions (cf. section 2.1):

Table 3.1.2: Types of Interactions for RRSs

Type	Example	Example SNS
Symmetric	Friend/Follow request, Ask for date	Facebook, OkCupid, Instagram
Asymmetric	View profile, Like, Subscribe/Follow	Twitter, YouTube, Instagram

As shown by the examples in table 3.1.2, for symmetric interactions, consent is needed and the interaction takes place on both sides ("blocking" request and response). In asymmetric interactions, the user can perform an interaction without any consent of the other party. This is analogous to the concept of asymmetric (e.g. *Twitter*) and symmetric Social Network Sites (SNSs) (e.g. *Facebook*) described in [Pal+20, p. 19]. Consequentially, a SNS which solely consists of asymmetric interactions (*pure asymmetric SNS*) does not rely on reciprocity. In such a network, the user can simply interact (such as like, view and follow) anyone, without needing their prior consent. In the reality though, as already shown by the *Instagram* example in section 3.1, a pure asymmetric or symmetric SNS does rarely exist: As soon as a user wants to contact another user (e.g. by sending a chat message), she/he expects a response (communication must be reciprocal). Therefore, *if* the system's objective is to maximize the likelihood of chat message responses, it is advisable to use a reciprocal RS. By following this logic, RRSs are a more fitting choice for SNS as long as they also have at least one *symmetric* interaction to consider, or alternatively, if users have a strong tendency of wanting to be "followed back".

It has been found that inferring user preferences from interactions is more effective than explicitly asking the user for preferences about their counterpart [Ake+11, p. 1]. Preferences change over time. Thus, asking users to explicitly enter them is more likely to result in outdated information, impairing the precision of recommendations. Therefore, both asymmetric and symmetric interactions can be used as implicit feedback for user preferences.

Regarding feedback towards liking other users, symmetric interactions are more explicit for both sides (the sender and the receiver of the interactions). For instance, if Bob sends a friend request to Alice and she accepts it, which can be seen as a clear sign of sympathy and therewith positive reciprocal feedback signal. Therefore, in the literature, symmetric interactions are commonly also called Expression of Interest (EoI) [Piz+10a] [Pal+20] [Ake+11].

3.2 Summary: Comparison to traditional RS

Summarizing, table 3.2.1 highlights some of the most notable differences of RRSs as opposed to traditional RSs [cf. Piz+10b, p. 2] [cf. Pal+20, p. 7] [see also Agg+16, p. 443].

Table 3.2.1: Comparison: RSs vs. RRSs

Property	Traditional RS	Reciprocal RS
Entities	Users are the subjects and items are the objects.	Users are the subjects as well as the objects.
Interactions	Users are *proactive* (senders of interactions).	Users are either *proactive* (senders of interactions) or *reactive* (receivers of interactions).
Success	The recommendation is successful if the user responds positively to it.	The recommendation is successful if *both* users respond positively to it (consent).
Availability	Persistent: Usually multiple items are available to be recommended to multiple users.	Temporary: In some scenarios users become unavailable to others after a specific event and are not open to be recommended.
Distribution	Imbalanced distribution of recommendations does not have direct social consequences.	Balancing recommendations is particularly important and all users should be treated equally.

In the following, the different properties of table 3.2.1 are explained in more detail, one paragraph per explanation/row.

The *entities* within a RS describe the participants of the system and their respective roles. As shown in section 3.1.1, a Reciprocal RS requires a human (*object*) which will be recommended to the human (*subject*). To the contrary, in traditional RSs, an item is recommended to a human subject. However, it has to be noted that there are also Non-Reciprocal RSs which have the purpose of connecting two users to each other, but which do not require reciprocity (see eq. (3.1.1) alone). For instance, as introduced in section 3.1, this could be applicable for the asymmetric SNS Twitter, where users can follow other users without any consent [Pal+20].

For *interactivity*, users in a RRS can generally be divided into *senders* of interactions (who act *proactively*) and *receivers* of interactions. Moreover, for symmetric interactions, a receiver of an interaction has the possibility to react accordingly (e.g. “accept” or “deny” friend request), as also explained in section 3.1.2.3. In contrast, RSs only have *proactive* users who engage with items to feed the system's input [Piz+10b, 2 ff.].

In RRSs, a recommendation is *successful* if both parties respond positively to it. Otherwise, the reciprocity of the recommendation is not given (e.g. single-sided liking). Especially, in RRSs, for success, the user knows that his/her counterpart needs to agree. For this reason, the aggregation of preferences as described in section 3.1.2.1 becomes inevitable. Oppositely, in a traditional RS, success is solely determined by the acceptance criteria of the user who receives the recommendation [Piz+10b, p. 2].

In a traditional RS, multiple items are *persistently available* to be recommended to multiple users (*many-to-many* cardinality) – typically without any constraints, even after there were multiple successful interactions with the recommendations. However, in a Reciprocal RS, recommendation candidates can possibly become unavailable after a successful recommendation has taken place (e.g. in dating) [Pal+20].

Lastly, an important difference for the *distribution* of recommendations within RRSs is that users have a limited attention span and therefore can not be recommended indefinitely (see "Availability" characteristic above) [Piz+10b]. On the other side, it should be avoided that a user does not receive any recommendation at all, as it leads to high frustration and users quitting the participation (see section 2.3.1). Therefore, balancing the recommendations equally is important. For traditional RSs, this is usually not a major problem with direct social consequences, as the objects are not humans.[2] Moreover, the items are typically either inexhaustible resources (e.g. applications) or provided on a supply and demand basis, which means that they can be reproduced (e.g. fashion products).

For both types of RSs, non-reciprocal and reciprocal, "rich get richer and poor get poorer" effects should be avoided as they lead to least diversified recommendations (see section 2.3.3).

[2]Note that humans are nevertheless indirectly involved, i.e. as the owner of objects (e.g. retailers of products).

3.3 Analysis of Requirements

In the following, we analyze the requirements for engineering a **Single-Class RRS** that is suitable to be used in SNS scenarios such as *online dating*, *e-learning* and *skill-sharing*.

In this thesis, we focus on *human-to-human* recommendations and do not want to conceptually draw borders between users (e.g. through separation by gender) for our recommendations. Thus, we focus on a homogeneous *single-class* RRS, which we consider as the more generalized class of a RRS. Interestingly, the single-class type received much less attention in the past years, which is (likely) explainable by the binary predominance of heterosexual dating and recruiting use-cases [Pal+20, p. 6 f.]. It is notable that the single-class approach does not restrict the system to consider must-have criteria; filtering techniques can be used to only consider humans that are relevant (compatible) to each other, i.e. fulfilling specific user preference requirements such as specific gender, age range or specific interests.

3.3.1 Challenges and Difficulty

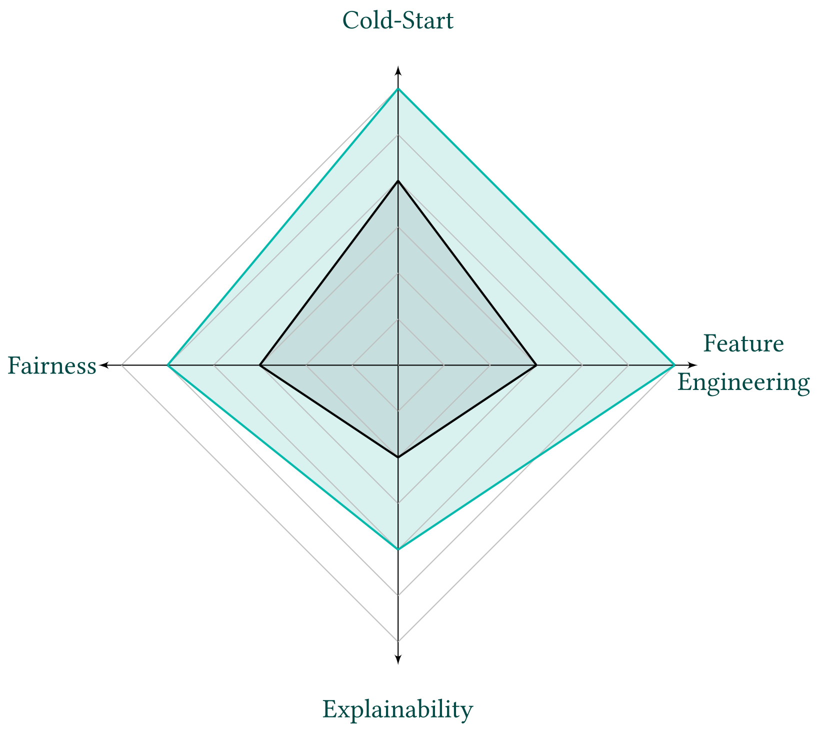


Figure 3.3.1: Comparison of the complexity with RRSs in mint and traditional RSs in black

With Tackling Cold-Start, Feature Engineering, Fairness and Explainability, four general challenges that we consider especially relevant for RSs have been introduced. These apply to RRSs, too, but due to the specific properties (as outlined in section 3.2), accomplishing the goals becomes more complex: By reweighing them accordingly in the context of RRSs, we are able to better estimate their implementation effort. For illustration purposes, the spider chart fig. 3.3.1 provides an overview for each goal's approximated difficulty. The diagram shows the traditional RS baseline in black and the RRS in mint.

In summary, for RRSs, all of the four key goals require a remarkable amount of effort as compared to RSs due to the always prevalent complexity-increasing social component. This also implies that a human-to-human RRS should be tested more carefully than a user-to-item RS, because the consequences of potential failure in these social aspects are inherently more severe for their users. Thus, we suggest that system environments which heavily depend on reciprocal recommendations first introduce a closed beta version (or similar) for highly monitored internal testing.

3.3.1.1 Cold-Start

The *new user cold-start*[3] problem has a substantively higher impact on RRSs. This is due to multiple facets in the reciprocal setting:

- Availability (cf. table 3.2.1): Depending on the application, users might disappear completely from the system after success (e.g. in the dating domain), thus resulting in less available users for recommending. Additionally, it might prevent the system from using the collaborative data of them in cold-start settings (e.g. upon account deletion) [Agg+16, p. 443].
- Success (cf. table 3.2.1): Because one successful recommendation requires two positive user interactions, the difficulty of finding recommendation candidates for cold-start users increases; it is not enough that the new user likes the recommendation, but the object also need to like the new user. This further shrinks the potential set of users to be recommended.
- Social: Users who are new to the system are isolated nodes. It is especially important to be able to offer reliable (preferably reciprocal) recommendations right from the start to

[3]In RRSs, there are no items and therefore the *new item cold-start* is non-existent

prevent disappointment and social exclusion. For traditional RSs, not receiving an *item* recommendation does usually not involve a direct risk of negative social consequences.

3.3.1.2 Feature Engineering

RRSs rely more heavily on (user) content data than RSs due to the accelerated new-user cold start problem [Agg+16, p. 444]. Moreover, there is only one type of information to be used as input, that is *user metadata* (and no *item metadata*). The user input is more prone to inaccurate or invalid information and therefore the features require a more precise selection, cleaning and excessive preparation than the content information of traditional RSs items which is typically entered by experts in the specific domain.

Another view on the importance of proper data cleaning: It is known that attacks by malicious actors against the RS itself exist. Attackers can place specifically designed features or several artificial influential user profiles into the system.[4] As RRSs built on user interaction that can go beyond the controlled virtual environment of the application, such as real-life meetings with foreigners in dating scenarios, users can be brought into dangerous situations. Therefore, filtering out malicious profiles during, but also over and above, the process of Feature Engineering for RRSs is very important for the users' safety and requires more effort due to the complexity of the system itself and its possible broad attack surface that requires expertise.[5]

Concluding, RRSs naturally require more user data that needs to be considered and processed [Agg+16], accordingly the goal of suitable Feature Engineering is substantively more difficult to achieve.

3.3.1.3 Fairness

For RRSs, the Fairness goal has an additional social impact, that makes it a much more controversial and multi-faceted topic than for user-item RSs. In section 2.3.3, we examined the common popularity bias in the context of RSs.

If we consider the following: In a RRS, *if* a user is an extremely popular choice for many reciprocal recommendations, this implies several characteristics:

[4]further reading available in [Agg+16, pp. 385 - 398]
[5]see also SFR+06, for more security considerations on RSs.

- Users have a limited attention span (see table 3.2.1, “Availability”). Over-representing her/him in the list of reciprocal recommendations will likely result in rejection or the user ignoring the majority of asymmetric interactions.
- Other less popular users are likely to be under-represented and can neither receive sufficient reciprocal recommendations as subjects nor be recommended to other users (“niche group”) that normally would in fact be interested (cf. section 2.3.3, “Expectation vs. Recommendation”). This is also an obstacle that makes handling cold-start more difficult.
- Recommendations are less diverse and users can not profit from novel recommendations that are beyond their typical social environment.

Especially because the recommendations have the potential to play an important role in reducing racial and social prejudice [Mcm19] [Lew13] and because of the large social impact of fairness [GF17, 53 f.], it is consequentially ranked higher for RRSs than user-item RSs which usually do not built upon social interactions.

3.3.1.4 Explainability

As shown in section 2.3.4 before, providing meaningful high-quality explanations is by itself a difficult task, as it requires the detailed study of user behavior and understanding. Nevertheless, the baseline in black for RSs in fig. 3.3.1 shows the least estimated importance, because there are relatively safe choices for explanation interfaces (see section 2.3.4). Additionally, failure to provide high-quality explanations does not prevent the user from receiving meaningful recommendations, thus they have more of a complementary (optional) character than the other goals.

As opposed to the traditional variant, for RRSs, we identified three additional restrictions and challenges to accomplish the goal:

1. Personalized explanations are much more suitable in RRSs than explanations that rely on public information. These explanations would justify a recommendation in reciprocal environments too superficially. For instance, “Alice is recommended to you because she is very popular among many other users of this network” would a) fail to transparently explain the recommendation and b) not suit the goal of fairness, i.e. it would explain that the system suffers from popularity bias (see previous section 3.3.1.3).

2. Because of the first restriction, we need to use *private*-personalized information to justify a recommendation. As reciprocal recommendations require both users to take part in the prediction, both users' preferences should be considered within the explanation, for instance to support the transparency. The problem with also explaining why the object user likes the subject is that the general preference information of the respective other party should be kept private, as it might also implicitly reveal past interactions with other users. On the other hand, silencing the object's preferences makes the explanation less transparent and only one-sided, similar to non-reciprocal recommendations.

The first-of-its-kind study [KRK18] from July 2018 analyzed *when* and *how* to provide reciprocal explanations. The following generic construct[6] used to explain reciprocal `recommendations` to a user `u` [KRK18]:

```python
1 def explain(u, recommendations) -> list:
2     explanations = []
3     for v in recommendations:
4         ex_uv = rrs.explain(u, v) # Preferences of u towards v
5         ex_vu = rrs.explain(v, u) # Preferences of v towards u
6         explanations.append((r, ex_uv, ex_vu)) # Aggregate explanations
7     return explanations
```

Listing 3.3.1: Construct for reciprocal explanations

Depending on the implementation of the `explain` method this would be sufficient to get a tuple of preference attributes `ex_uv` of subject user `u` for `v` (line 4) and a tuple of preference attributes `ex_vu` of object user `v` for `u` (line 5) which is then combined for the recommendation `r` and appended to a result list (line 6). Notice how this reflects the aggregation of preferences as in eq. (3.1.4), only that now preference attributes are fused together to form an explanation for each recommendation instead of building a preference score.

[6] cf. KRK18, "Algorithm 1" has been transformed to a more concise Python-like pseudo-code.

Example 3.3.1: Privacy vs. Reciprocal Explanation

In a dating RE, the following data[a] is known about user Alice:

```
alice = {'gender': 'female', 'education': 'B.Sc.', 'economic': 'wealthy'}
```

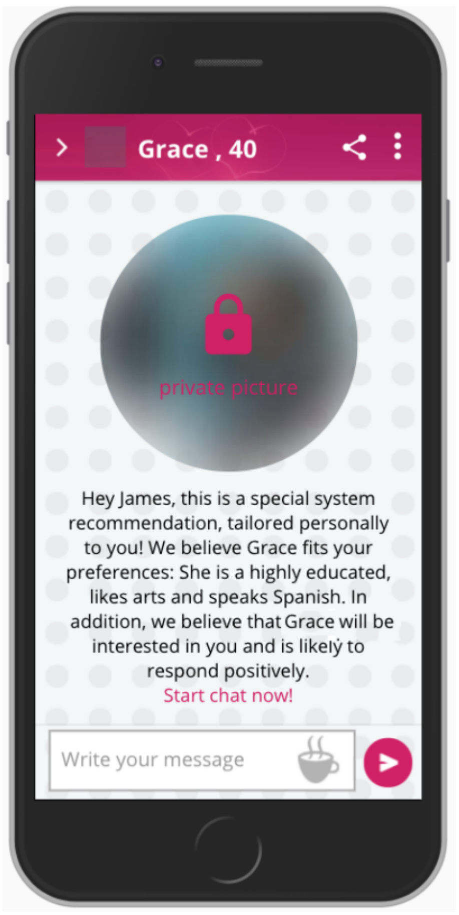
If we keep the reciprocal explanation method in listing 3.3.1 as is for Alice and do not further filter the preference attributes, the tuple `ex_vu` is problematic: Suppose user `v` is Bob, and one of the attributes in `ex_vu` is 'wealthy'; because he recently chatted with a lot of people with the 'wealthy' economic status. Giving Alice the explanation that Bob has been recommended to her because he likes 'wealthy' people could be wrong or misleading and a violation to Bob's privacy. Bob definitely does not want his computed preferences to be exposed, *especially not* to Alice, because it could negatively impact her opinion about him for further interactions (or EoIs).

[a] Some attributes are extracted from KRK18, fig. 1, to reflect the current situation in dating apps.

In [KRK18], two different implementations of the `explain` method have first been tested in a simulated environment with 121 participants and later in a real-world dating RE with 287 participants. The first named “transparent explanation method” works by explaining the top k attributes that were most popular among a user's chat partners. The second named “correlation-based explanation method” works by putting these into *correlation* to the total of profiles the user has sighted (also including the ones that the user did not further interact with beyond viewing them).

For instance, James decided to chat with 13 users, of whom 10 had an 'academic' background and 7 liked 'arts'. For the top $k = 2$, the first algorithm would return `('academic', 'arts')`. The second algorithm would return `('arts', 'academic')`, which is due to the following fact: James viewed a total of 37 profiles, 20 were academic, but only 8 liked arts. Therefore, he only contacted half of the academic persons, but every 'arts'-person except one. This makes the correlation between the messaged people who liked arts and the total representation of viewed people stronger than for academic people.

Overall, the *correlation-based explanation method* was superior, showing significantly higher values for *perceived transparency*, *satisfaction* and the *perceived usefulness* in a survey for the participants, which is why this method has been chosen for reciprocal recommendations [KRK18].



Hey James, this is a special system recommendation, tailored personally to you! We believe Grace fits your preferences: She is a highly educated, likes arts and speaks Spanish. In addition, we believe that Grace will be interested in you and is likely to respond positively.
Start chat now!

Figure 3.3.2: Reciprocal explanation in a dating RE with removed partner preference attributes.

SOURCE: [KRK18]

In the real-world dating RE, the research team of [KRK18] was not allowed to reveal the preference attributes of the recommendation object to the subject due to privacy concerns, which is due to restriction 2 from the beginning of this section. From a data protection aspect, this is ideal, but less beneficial when it comes to a neutral study environment[7]; as shown by fig. 3.3.2, the team had to limit the reciprocal part to a generic sentence that the other user is likely to respond positively instead of showing *why* the user is interested, i.e. which specific attributes the other user likes. Nevertheless, even with this minimal reciprocal part, the explanation outperformed the pure one-sided explanation that is typically to be found in non-REs. It especially had an encouraging effect on users who sent less messages than the median. Therewith, this user group showed a significant raise in the acceptance rate of a recommendation, as it possibly helped with alleviating possible fears of rejection [KRK18, p. 6] and therewith even increasing the *trust* in the system.

3.3.2 Requirements

By carefully examining the results of section 3.3.1, we derive the following *functional* requirements for our multi-purpose RRS framework, considering **researchers** and **developers** as primary *users* of the framework:

[7] An influencing side-effect has been introduced: The study compares reciprocal vs. non-reciprocal explanations, however, especially the reciprocal part is impaired by removing the *other* user's preferred attributes.

1. **Tackle Cold-Start**

   Recommendation subjects that do not have an interaction record should be able to receive suitable recommendation objects.

2. **Automatize Feature Engineering Steps**

   The user should be supported with commonly used feature engineering procedures (even though most of feature engineering is domain-specific). Testing with new features must be made easy.

3. **Algorithmic Choice**

   The framework must offer a built-in variety of recommendation algorithms.

4. **Evaluation**

   By providing common evaluation functionality, users are able to analyze performance and should be encouraged to experiment with different parameters.

5. **Data Source Independence**

   The framework must be independent from any data source, i.e. users should not be forced to a single data storage format.

6. **Fairness**

   Provide functionality to estimate a users popularity and mitigate the popularity bias.

7. **Explainability**

   Provide methods to simplify possible explanations about recommendations to assist so that users are able to create explanations that follow the three guidelines mentioned in section 2.3.4. To further improve the understanding of data and algorithms, the framework should contain built-in visualization capabilities.

8. **Implicit Feedback Specialization**

   Implicit feedback data is a potentially widely available and powerful source of user feedback [Kul18] and the framework should prioritize it over explicit feedback data (see also section 2.2.3.2 for the motivation).

Furthermore, the following four requirements are on the *non-functional* side of requirements:

1. **Reproducibility**

   Provide *reproducible* and *comprehensive* examples to help users to further research the field of Reciprocal Recommender Systems.

2. **Standardization**

   Establish a well-known model to encapsulate users and their interactions usable by researchers to fluently describe a RRS.

3. **Modular Extensibility**

   It should be made easy to extend and enrich the framework in the future (for instance with new RRS algorithms).

4. **Convention over Configuration**

   Default values should fit a broad spectrum of use-cases.

4 Chaos

In this chapter, we explore the solution that has been specifically implemented for this work: *Chaos*, a novel framework for RRSs, specialized in implicit feedback. We start by introducing a typical workflow highlighting use-cases and then outline the technology it builds upon in section 4.1.2. These are chosen to be in compliance with the requirements in section 3.3.2.

In section 4.2, we discuss each core component and the involved architectural and programmatic design in detail. Thereafter, in section 4.3, we describe the implemented recommendation approaches which we consider as the algorithmic core of Chaos.

Finally, in the last section 4.4, Chaos' versatility is demonstrated by using it in combination with GitHub's publicly available API to present an innovative solution for personalized code collaborator recommendations.

4.1 Overview

4.1.1 Workflow

A typical workflow that must be supported by the framework is *offline training*, which is used to train an ML-based model initially [CAS16]. Offline training essentially translates to "learning on a batch of training data that is known beforehand", i.e. there is an ambiguity to the more commonly used offline/online states in network terminology. To disambiguate, some sources use the term *static training*. *Dynamic training* refers to a continuously updated model that is guaranteed to be on par with the latest *online* training data. [CAS16]

In case of Chaos, the process in fig. 4.1.1 reflects most of the framework's functionality and core components, thus providing a good *high-level* overview from a user's perspective. Section 4.1.1 briefly explains each task of the workflow with a reference to the upcoming section(s).

Table 4.1.1: Offline Training workflow description for fig. 4.1.1

Task	Description	Reference
Select features/interactions	The Researcher needs to decide which features and interactions are included to fetch a batch of (training) data.	3.3.1.2
Source origin data	Original data needs to be sourced (for instance from a database or file) and transformed to the data model.	4.2.1, 4.2.2
Process Data Model	The Data Model is processed by cleaning, extracting and processing features.	4.2.3
Translate Data Model	The Data Model is translated to a language that the predictor can understand.	4.2.4.1
Train Predictor	The Predictor is trained on training data so that it is later able to generate recommendations for users.	4.2.4.3
Evaluate Predictor	After training, the Predictor is evaluated by the Evaluator by generating recommendations against known positive interactions with different metrics.	4.2.5
Analyze Report	The researcher analyzes the evaluation results. If the performance is sufficient, the model can be deployed. Otherwise, see below.	4.2.5.2
Change parameters	If performance is insufficient, the model parameters (either features, interactions or hyperparameters) need to be tuned or modified.	4.2.5.2

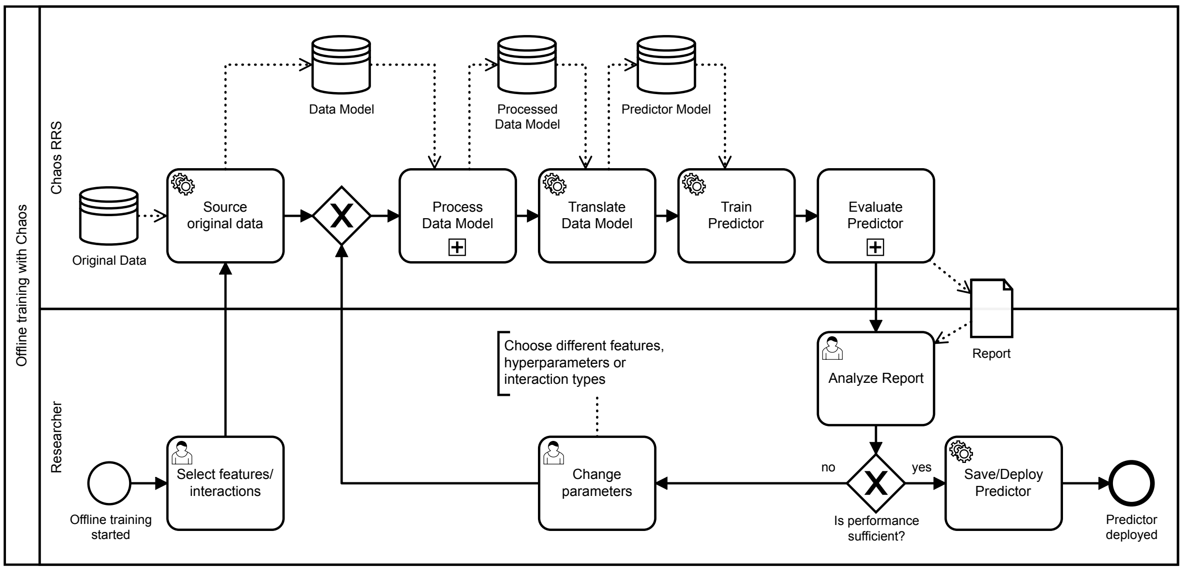


Figure 4.1.1: Offline Training with Chaos before deployment

Please note that the termination criterion either depends on a pre-defined goal or if no further improvement is possible (considering limiting/limited resources). In general, there are multiple metrics to be used, some of the most important for RRSs are outlined in section 4.2.5.

4.1.2 Technology

In the following, the used technology is described. The framework is fully written in *Python* and uses *Conda* as a cross-platform package/environment manager to offer a reproducible researching environment. To provide an overview, the used packages are divided into four categories; Data Model, Feature Engineering, Prediction and Visualization, see fig. 4.1.2.

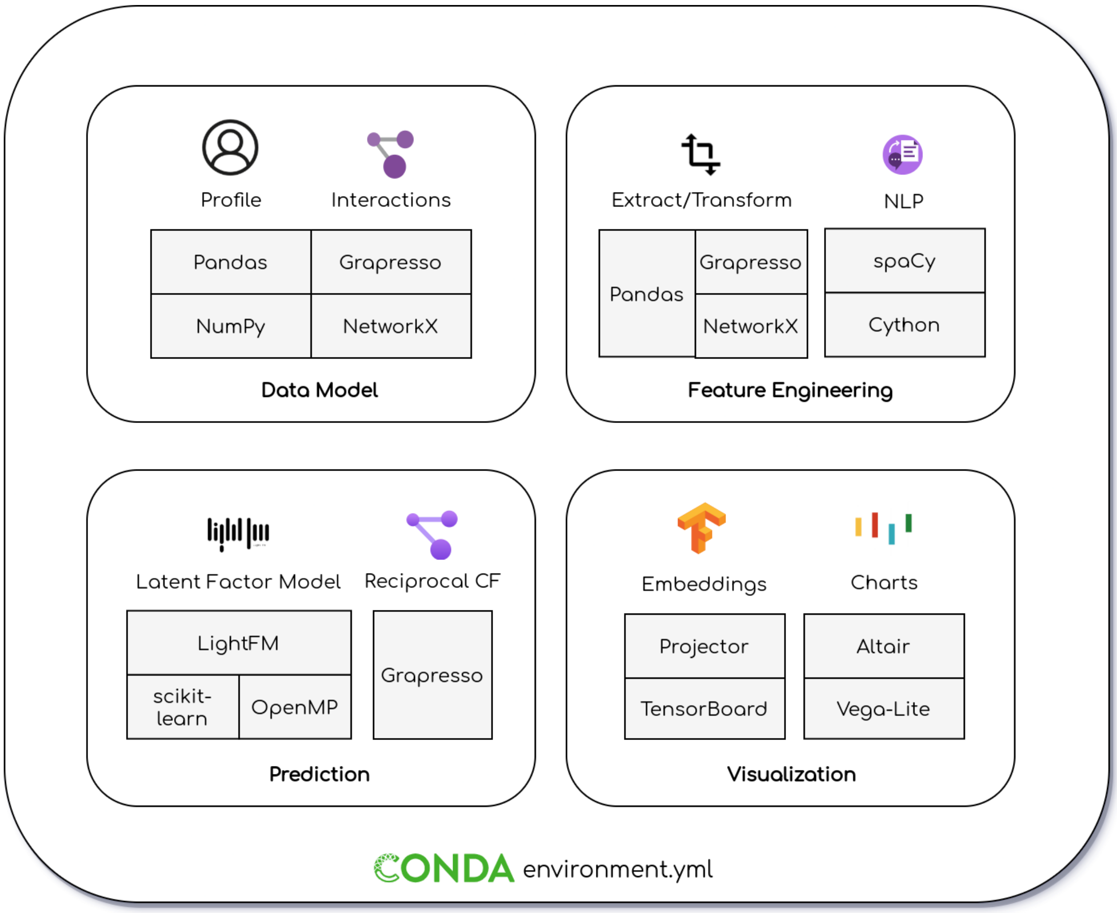


Figure 4.1.2: Technology Stack of Chaos

4.1.2.1 Data Model

The data model uses *Pandas*[1] for user profile data and *Grapresso*[2] as a graph library for user relations. Pandas is a well-known Python library often used by data scientists to represent tabular (i.e. 2-dimensional) data in *data frames*. It partially builds on top of more primitive but powerful *NumPy* arrays which are *vectorized* in order to achieve a higher performance level.

Grapresso is a meta graph-library that has been refined by the author during the work on this thesis. It aims to provide one simplified API that can be used for a variety of *backends*. In case of Chaos, the mainly used backend is the popular pythonic *NetworkX* graph library which is integrated in order to gain access to the broad range of network analysis methods that it provides [HSS08]. Grapresso is used as an intermediary where possible to be able to exchange the backend at a later stage of development if necessary more effortlessly.

4.1.2.2 Feature Engineering

To process the data model as a step of feature engineering (see section 2.3.2), transformations and extractions are performed by using the technology of the data model.

However, we need to deal with user information types that are non-standardized (such as user biographies, user status, favorite activities or other free text fields) instead of restricted to a specific choice of options (such as age or location). This is a challenge for (partly) automatizing feature engineering. Therefore, we integrated *spaCy* as a fast Natural Language Processing (NLP) library [Hon+20] that attempts to understand humans.

4.1.2.3 Prediction

We choose the aforementioned *LightFM* (see section 2.2.3) as a reference implementation for the framework's hybrid prediction/recommendation capabilities. In general, the integration of an existing latent factor model implementation that is normally only applied to retrieve user-item recommendations is uncommon in the field of RRSs [cf. Pal+20]. Nevertheless, considering the objectives and the research question of this work, the following determining advantages have been determined: Frameworks for user-item RSs are widely available

[1]`https://pandas.pydata.org/`
[2]`https://git.io/grapresso`

[Kul15] [Goo18], well-tested in production [Kul15, p. 6] [CAS16] and in the literature they are extensively known and described [Agg+16]. By contrast, for RRSs, the algorithms are largely only available on a pseudo-code basis and major studies rely on proprietary datasets [Ake+11] [Piz+10b] [Xia+16] [NP19a], thus being questionable regarding their results' *validity* and *reliability*, as already noted in the introductory remarks (section 1.2).

The "constrain" to integrate an existing code foundation is actually an advantage that helps with the overall goal of reproducibility. *LightFM* has been evaluated on the publicly available *MovieLens* dataset [Kul15], which is one of the most well-studied datasets for RSs [Eks+11]. Finally, supporting a hybrid approach allows us to unite the fundamental advantages of content- and collaborative information over their pure variants as highlighted before (in section 2.2.3 and section 2.3). For instance, it aids to fulfill the requirement of explanations because the learned embeddings can be used to estimate a user's similarity to his/her specific attributes. Even more importantly, it helps with mitigating the cold-start problem as new (unknown) users can be constructed based on their profile data.

4.1.2.4 Visualization

For visualizing learned embeddings, the framework uses the *Projector* UI component from *TensorBoard*. It enables the projection of high-dimensional embeddings in a 3-dimensional space that can actually be *perceived* by the human eye and *interpreted* by the human brain. This greatly increases the explainability of latent factor hybrid models.

For visualizing evaluation results, *Altair* is used. It is a statistic visualization library that builds upon a small visualization grammar called *Vega-Lite* and features a declarative way of defining diagrams that often makes it possible to provide meaningful statistics with just a few lines of codes [Van+18]. The precondition is that it follows the basic conventions of *tidy data*, which "provide a standardized way to link the structure of a dataset (its physical layout) with its semantics (its meaning)" [Wic+14, pp. 1–4]:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

Where appropriate and possible, Chaos makes use of these three basic principles.

4.2 Core Components

In the following, the core components of the framework are described. Each component is accompanied by a simplified UML diagram to provide an object-oriented view on the framework. For better readability, methods/attributes that do not play a direct role in the explanation text have been removed.

Notebook 4.2.1: Interactive Chaos

Readers who want to explore Chaos practically are invited to start an interactive notebook with *JupyterLab* right at this point. For easy setup, please refer to the project's `README.md`, "Scenario 1: Learning Group".

4.2.1 Data Model

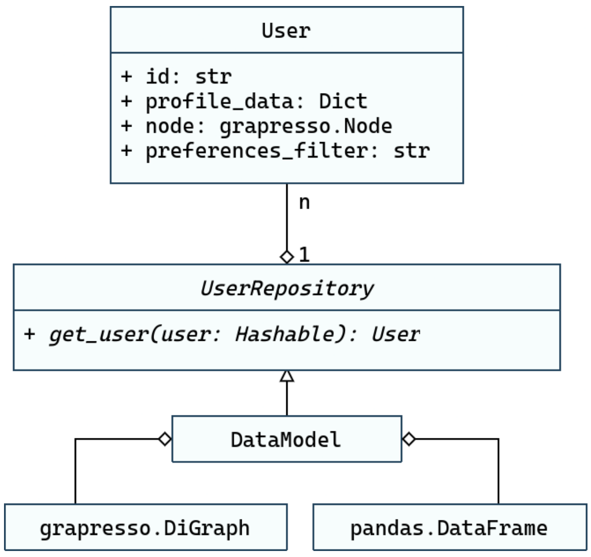


Figure 4.2.1: Data Model UML diagram

The data model consists of *user interactions* on the collaborative side (section 4.2.1.1) and *user profile data*[3] on the content side (section 4.2.1.2). Together, both data types form the input to a hybrid model, and alone they respectively form the input to a CF or CBF implementation.

As shown by fig. 4.2.1, the `DataModel` is derived from the most generic `UserRepository` and

[3]For more explicitness, please note that we prefer to use *profile data* over *metadata* as the latter describes "data over data". In the former chapters, we used *metadata* in the context of hybrid RSs where the *interactions* (or embeddings) were the user's "main data" and the supplementary content data therewith "metadata".

the user is – in total – represented by a unique `id` (the user's name), a dynamically defined `profile_data` dictionary (section 4.2.1.2) and a reference to the user's graph `Node` which is a direct hint to the next section.

4.2.1.1 User Interactions

As summarized in table 3.1.2, typical (a)symmetric interactions for implicit feedback are profile views, messaging, follow/friend requests or user likes. For some implicit feedback interactions, there is a – often time-constrained – usage limit, e.g. Bob can send a friend request to Alice only once a month. To the contrary, some are not limited in any way, e.g. Alice can view Bob's profile a thousand times. This indicates that implicit feedback interactions are unbalanced between users by nature. Even if the interactions happen solely on a symmetric, one-time basis (EoIs), it might be a requirement to differentiate between an proactive ("ask for friendship") and a reactive interaction ("accept friendship"), see table 3.2.1.

For the above reasons, we choose a directed graph for the interaction model with the users as nodes. Furthermore, each interaction can be weighted differently; for instance, a "view" interaction is less meaningful than a "sent chat message" or an "accepted friendship". To not confuse these weights with the common *weight* term in graph theory, we use the term *strength* to indicate how strong the feedback of an interaction is.

Therefore, we specify that an edge between two nodes u and v approximates the strength of all interactions between them (denoted as $I(u, v)$) by the sum of all (atomic) single interaction strengths s_i:

$$s(u, v) := \sum_{i \in I(u,v)} s_i \text{ with } s_i > 0 \tag{4.2.1}$$

The constrain on positive strengths comes from the framework's specialization in implicit feedback, where no negative feedback is modeled explicitly (section 2.2.3.2) and it furthermore aids the simplicity and compatibility with many algorithms from a network analysis as well as RS standpoint (see upcoming section 4.2.3 and section 4.3.3). For the predictor introduced in the following section 4.2.4.3, the strength between two nodes is the *confidence* we have that the interaction is indeed a positive interaction.

Figure 4.2.2 has been automatically rendered by the framework with help of *NetworkX*. It

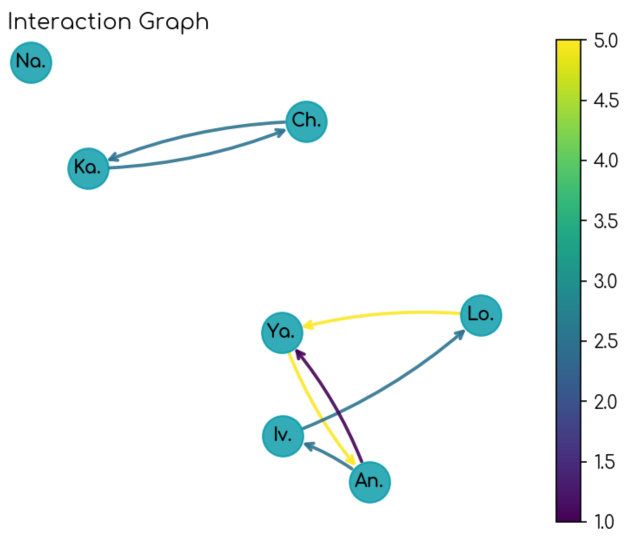


Figure 4.2.2: Example of a directed interaction graph/network with color bar

consists of various interaction types: One cold-start node with zero interactions, two pairs of reciprocally (bidirectionally) connected nodes and three one-sided interaction edges. Together with the color bar on the right, the color of the edge between u and v indicates $s(u, v)$ from eq. (4.2.1). Note that without any further normalization of $s(u, v)$, some edges might be outliers – see the upcoming section 4.2.3 for further guidance on limiting over-weighting.

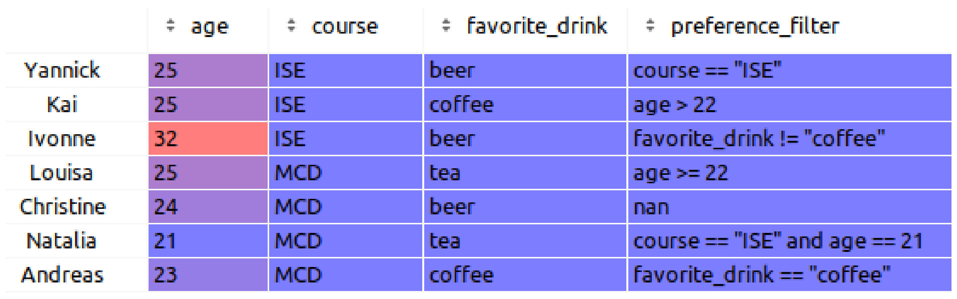
In an attempt to mirror a real-world situation commonly found on SNS, this graph is only partially reciprocal. The overall reciprocity of $\frac{4}{7}$ is given by the ratio of bidirectional edges and total edges – a complete reciprocal graph has a result of 1 [see also HSS08].

Example 4.2.1: Interaction Protocol and Finite State Machine

To support as many use-cases as possible, the framework's model makes no limiting assumptions about the *order* or *amount* of interactions. Nevertheless, in many SNS, the interactions can be modelled as a *network protocol* between two clients (the users). For instance, the state of "message sent" can only happen *after* "friend request accepted" or similar. From a theoretic standpoint, they can also be represented as a *finite state machine*, but due to the involved complexity of representing two parties' interaction *states* per transition, *communicating finite state machines* might be more appropriate, which have been introduced in [BZ83].

4.2.1.2 User Profile

Each node in the interaction graph should (but does not need to) have an entry in the user `DataFrame`. The primary key to link each node from the previous section to the user profile row in a `1:0..1` way is the *user name* which is a string to uniquely identify a user across all of the different components, thus improving the framework's usability and clarity from a developer perspective.



	age	course	favorite_drink	preference_filter
Yannick	25	ISE	beer	course == "ISE"
Kai	25	ISE	coffee	age > 22
Ivonne	32	ISE	beer	favorite_drink != "coffee"
Louisa	25	MCD	tea	age >= 22
Christine	24	MCD	beer	nan
Natalia	21	MCD	tea	course == "ISE" and age == 21
Andreas	23	MCD	coffee	favorite_drink == "coffee"

Figure 4.2.3: User profile `DataFrame` with one row per user and columns as attributes

Figure 4.2.3 shows an artificial/exemplary data frame captured in the *SciView* of the popular Python IDE *PyCharm*[4]. Each row contains a user (indexed by the unique user name), each column describes an attribute of the user. Chaos is fully agnostic towards the contained attributes, except for the last column: The `preference_filter` is a special column which contains a *numexpr*[5] that is used in conjunction with Pandas `query` function to efficiently retrieve other users matching the criteria. For instance, the user row "Yannick" contains a must-have criteria for users who study ISE, i.e. only the other two users ("Kai" and "Ivonne") will be matched (himself excluded). We examine the component that implements this functionality in section 4.2.4.2.

4.2.2 Data Source

As shown in fig. 4.2.4, a data-source can be as simple a CSV reader that reads-in two local files (one for interactions and one for the user profiles) or more complex, fetching data from a remote *GraphQL* endpoint (described later in section 4.4.1). Users of the framework are encouraged to write their own implementation of the very basic data `Source` interface in case a specific persistence technology is missing and optimally contribute it to the project.

[4] `https://www.jetbrains.com/pycharm/`
[5] `https://numexpr.readthedocs.io/`

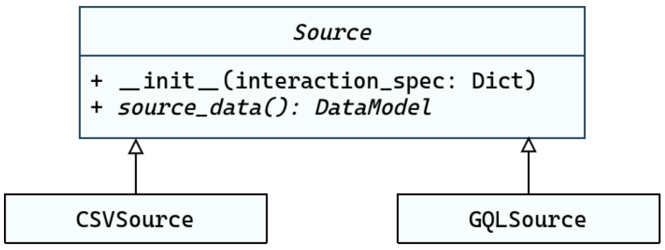

Source
+ __init__(interaction_spec: Dict)
+ source_data(): DataModel
CSVSource
GQLSource

Figure 4.2.4: Data Source and implementations class diagram

```
from,interaction,to
Yannick,chat,Andreas
Yannick,chat,Andreas
Kai,chat,Christine
# ...
Andreas,view,Yannick
```

Listing 4.2.1: Excerpt of an interactions CSV file

The excerpt in listing 4.2.1 corresponds to fig. 4.2.2 with the following strengths defined in an *interaction specification* (see also constructor of `Source` in fig. 4.2.4):

```
view:
  strength: 1
  description: 'Occurs when viewing a profile.'
chat:
  strength: 2.5
  description: 'Occurs when sending a chat message.'
```

Listing 4.2.2: Interaction specification with different strengths

4.2.3 Data Processor

`Processors`, as modelled in fig. 4.2.5 can be used to transform and extract features. They help the researcher to perform feature engineering more efficiently (cf. section 3.3.1.2). We differentiate between `Extractors` which potentially add a column to the user profile `DataFrame` and `Transformers` which are only allowed to alter existing columns.

A `Pipeline` further simplifies the workflow by aggregating multiple `Processors` to one. Its

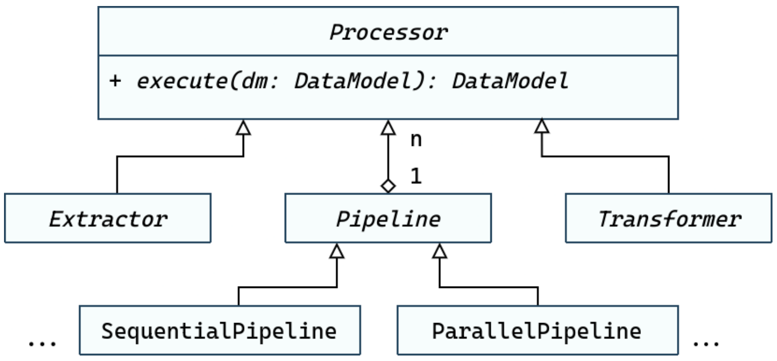


Figure 4.2.5: Data Processor and Pipelines class diagram

capabilities are best outlined by the following code example:

```python
1  pipeline = SequentialPipeline([
2    GraphEdgeMapper(
3      strength=lambda e: math.log(1 + e.strength, 2)
4    ),
5    GraphEdgeMapper(
6      capacity=lambda e: e.strength,
7      cost=lambda e: 1 / e.strength
8    ),
9    ParallelPipeline([
10     GraphPopularityExtractor('popularity', add_as_node_attrib=True,
11                              labels=['low', 'medium', 'high', 'prominent']),
12     DataFrameBucketExtractor('age', 'age_bucket',
13                              labels=['young', 'middle', 'old'])
14   ])
15 ])
```

Listing 4.2.3: Data Pipeline for feature engineering

Referring to listing 4.2.3: In lines 2 to 4, the edges' strengths are first smoothed with a `GraphEdgeMapper` by applying a $\log_2$. This exemplary normalization step ensures that there are less outliers in the interaction graph caused by edges that have a high strength, e.g. caused by many high-strength interactions. In lines 5 to 8, we show that more attributes (namely `capacity` and `cost`) are added to the edge effortlessly. One way to interpret each edge's strength for maximum flow problems is as capacity and for shortest path problems as inverse cost. It essentially opens the way for further experimentation with network algorithms.

Starting from line 9, we show that `DataPipelines` can be nested. Furthermore, they can be parallelized (useful for long running I/O-consuming tasks): The two inner `Extractors` perform an isolated process per `Processor`. Thereafter, the data is synchronized by merging it optimistically; this means that no checks occur if two extractors modify the same column ("the last one wins"). The `GraphPopularityExtractor` (line 10) approximates the popularity of a node and puts it in the "popularity" column. To calculate a node's popularity, we utilize graph *centrality* metrics. In the framework's first proof of concept, three metrics are supported and based on *NetworkX* [HSS08] algorithms [see also PRG16, p. 2]:

1. ***Degree** centrality*: Possibly the most simple metric to measure centrality in a graph, defined as the number of edges incident to a node.

2. ***Betweenness** centrality*: Metric based on how many times a node is passed for each of the graph's shortest different $u \to v$ paths (more efficient approximations are done by sampling only a few).

3. ***Eigenvector** centrality*: Complex metric based on the assumption that nodes connected to highly influential nodes result in a higher score than connections to less influential nodes. *Google*'s proprietary algorithm to rank pages is based on a customized version of this metric [PRG16].

By default, the *degree centrality* is used by the `GraphPopularityExtractor`. The latter two algorithms rely on the prior `cost`-assignment. If we would use the plain strength without inverting it, the objective would be inverted instead; that is, instead of measuring centrality, we would measure the opposite of decentralization, which is an inappropriate estimation of popularity.

Continuing with listing 4.2.3, the `DataFrameBucketExtractor` in lines 12 and 13 assigns one of the three labels based on *quantiles* to the age and writes them in the `age_bucket` column, which is a feature engineering step we encountered before in section 2.2.3 to discretize the continuous age value.

4.2.3.1 Alleviate Popularity Bias

To alleviate the popularity bias, please note that we could also use the result from the `GraphPopularityExtractor` to discount outgoing interactions to popular nodes, for instance by using the before calculated relative *degree centrality* (ranging from 0 to 1):

```
GraphEdgeMapper(
  strength=lambda e: e.strength-(e.strength * 0.5 * e.v.data['degree'])
)
```

Listing 4.2.4: Discounting edges based on relative *degree centrality*

By inserting the `GraphEdgeMapper` from listing 4.2.4 to the pipeline listing 4.2.3 (after line 11), the edges' strengths to highly influential nodes (`v`) are discounted by a maximum of 50% in an attempt to alleviate the popularity bias which we introduced in section 3.3.1.3.

There are many other experimental possibilities and combinations to embed network-based metrics into the interaction model: We could also discount[6] by the inverse reciprocity of a node (see section 4.2.1.1), or by the ratio of incoming and outgoing interactions to approximate the *responsiveness* of a node. This way, nodes that are believed to be overwhelmed by incoming interactions (or EoIs, see section 3.1.2.3) [Kle+18] are in the end less likely to be recommended all over again.

4.2.3.2 Summary

In the above example we only showed a variation of Chaos's feature engineering capabilities, a more versatile pipeline is presented in the following section 4.4.2.

Notably, Chaos features a clear, concise and declarative way for common feature engineering steps that invite to experiment with. For comparison, from a purely software architectural view, the pipelines from the popular ML toolkit *scikit-learn* are conceptualized similarly [Bui+13].

[6]either *discount* edges to *popular* nodes or conversely *promote* edges to *unpopular* nodes

4.2.4 Recommendation

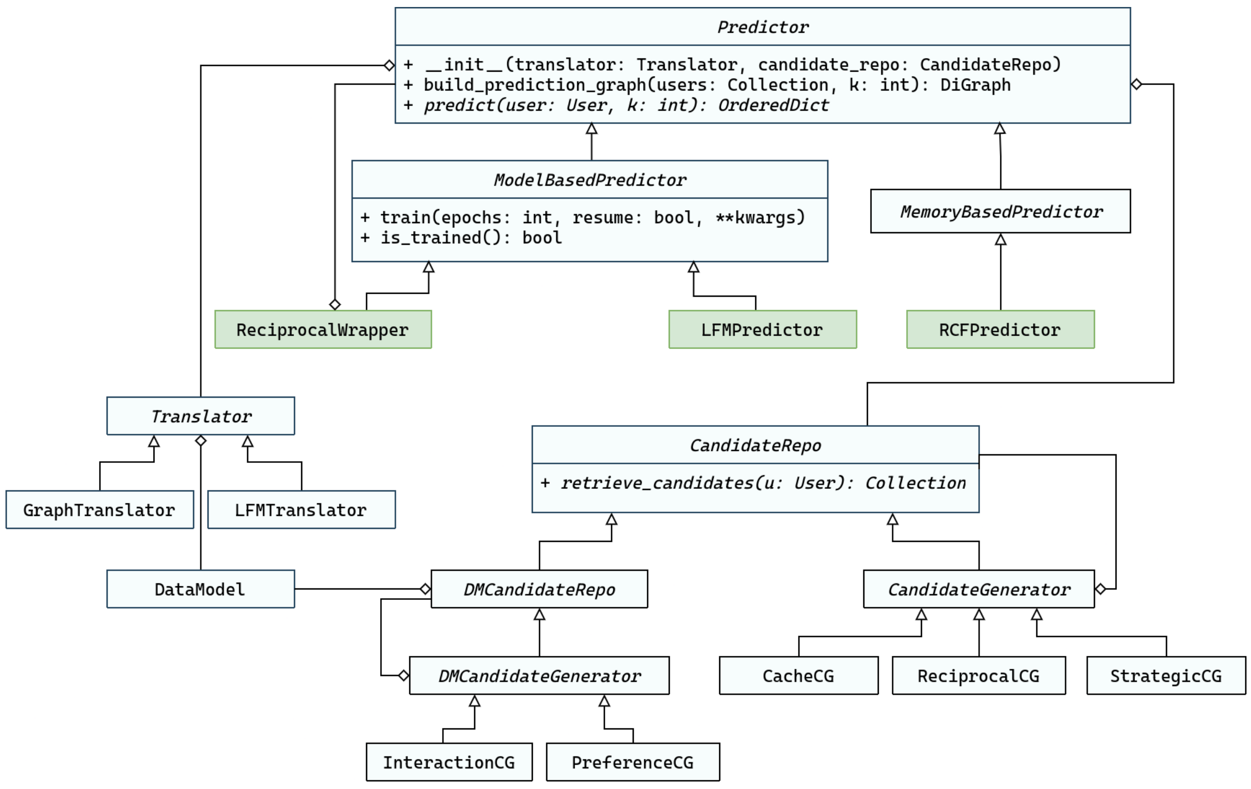


Figure 4.2.6: Major components around the Predictor class

As fig. 4.2.6 illustrates, multiple components are involved in the process of recommendations. The components with the main algorithmic logic are highlighted in green and will be explained separately in section 4.3, as they are the algorithmic heart of the framework.

4.2.4.1 Translator

Recommendation algorithms are allowed to be based on a different model than the framework's native `DataModel` which we described in section 4.2.1. For this purpose, the `Translator` exists: It is the two-way bridge between Chaos' `DataModel` and the `Predictor` (see section 4.2.4.3). For instance, the `LFMTranslator` class is able to translate the model to a representation that *LightFM* can understand: The user profile `DataFrame` from section 4.2.1.2 is translated to a user matrix where each feature is one-hot-encoded automatically. The graph from section 4.2.1.1 is translated to a *SciPy* sparse matrix[7] (see section 4.3.1 for more details).

[7]https://docs.scipy.org/doc/scipy/reference/sparse.html

4.2.4.2 Candidate Generator

Because Chaos is a single-class RRS framework (see section 3.1.2.2), generating potential candidates, which are users who are compatible to a given user, is an important step. From an efficiency viewpoint, generating candidates *prior* to recommendation is also a logical step; users that do not fulfill must-have criteria (e.g. given by the `preference_filter` in section 4.2.1.2) should be excluded as early as possible from further calculations.

Therefore, the `CandidateGenerator` is implemented and executed *before* calling the recommendation method for a user. Under the hood, it is implemented as a decorator pattern (see the UML diagram fig. 4.2.6) and instantiated either by using a builder or directly – in the latter case, the code can get more obfuscated due to the nested structure. This is shown by the comparative code listing 4.2.5 where both generators deliver equal results:

```
# Using nested Decorator (execution →, filtering ←):
cg1 = ReciprocalCG(
  CacheCG(PreferenceCG(DMCandidateRepo(data_model))
)
# Using Builder (with syntactic sugar for CacheCG and ReciprocalCG):
cg2 = (CandidateGeneratorBuilder(DMCandidateRepo(data_model)) # filtering ↓
  .filter(PreferenceCG).cache()
  .reciprocal()                                               # execution ↑
  .build())
assert cg1.retrieve_candidates('Kai') == cg2.retrieve_candidates('Kai')
```

Listing 4.2.5: Candidate Generator: Decorator vs. Builder

The full flow-of-control of the example code is shown by the sequence diagram fig. 4.2.7 where each `retrieve_candidates`-call is delegated to its inner successor. Focusing on this diagram, the semantics of each Candidate Generator (CG) are (from right to left):

- `DMCandidateRepo`: The repository for the candidates. The implementation is minimal and trivially returns the index values (user names) of the `DataFrame` but without the user `u` for whom the candidates are generated for (users do not receive recommendations with themselves).
- `PreferenceCG`: Filters the candidates returned by the repository and only returns the ones that match the `preference_filter`.

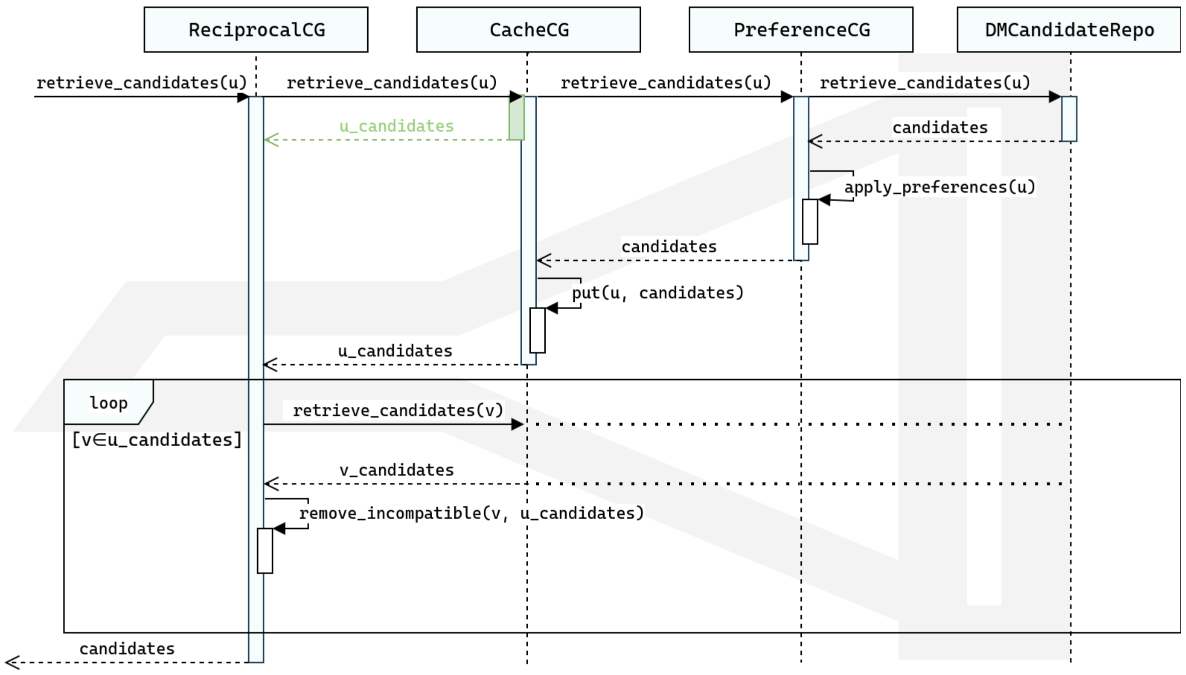


Figure 4.2.7: Candidate Generator Decorator Sequence Diagram

- `CacheCG`: Caching layer that saves the result for `u` in a dictionary (can be extended to e.g. make use of a LRU strategy). As indicated in green, if the user is available the next time the layer is called ("cache hit"), the path to the right is cut and candidates for `PreferencesCG(DMCanddiateRepo(dm))` are returned immediately.

- `ReciprocalCG`: Special CG that ensures that the set of `u_candidates` (candidates for `u`) only contains `v` who also have `u` in their set of `v_candidates`. In other words, this candidate generator ensures that the users are *reciprocally* compatible (compare with eq. (3.1.2)). In fig. 4.2.7, the repetition of the other CG's sequences is omitted for simplification (indicated by the vertical dot-line). Furthermore, this CG is subject to profit from the cache layer as requests for `u_candidates` are likely to be implicitly repeated, thus known to the cache. From a runtime perspective, this CG highlights very well how it consequentially is `len(u_candidates)`-times more expensive to filter for reciprocal interests.

Finally, with this knowledge, we are able to calculate the output of the call in line 10 of listing 4.2.5 before and after the `ReciprocalCG`: Before, all other users except “Natalia” will be returned as `u_candidates`. Then, the `ReciprocalCG` additionally removes “Ivonne” (because of her `preference_filter` $\neq$ `"coffee"`) which leaves us to 4 other users (Yannick, Louisa, Christine, Andreas) as recommendation candidates.

The naming “CandidateGenerator” is based on a component described in the paper [CAS16] about a large-scale DL-based RS for *YouTube*. It similarly filters videos by a funnel, which highlights the importance in the sense that this approach is (commonly) applied in real-world scenarios.

For Chaos, we found and utilized a design pattern that works particularly well: It enables a transparent, pluggable and highly customizable way to compose a CG ready to serve candidates for a specific use case

4.2.4.3 Predictor

As modelled in fig. 4.2.6, the `Predictor` class holds a `Translator` instance for two-way data model translations and a `CandidateRepo` for candidate generation. Furthermore, it has the abstract `predict` function for *one* user’s top `k` recommendations that are returned as a descending score-ordered dictionary in the form `'<username>': <recommendation_score>`. The `build_predict_graph` function builds a prediction graph for the selected `users` with `k` edges for each user (see parameters). It is essentially implemented as a for-loop that adds nodes/edges to a graph based on the former single-user `predict` function that needs to be implemented by concrete classes.

In the class hierarchy, we differentiate between `ModelBasedPredictors` and `MemoryBased-Predictors`, just as in the preliminary fig. 2.2.1:

The `ModelBasedPredictor` has an additional method to `train` (fit) the model in accordance to the interaction data that is provided by the `Translator`. With each `epoch`, the model is fitted more to the interaction training data. In case of the `LFMPredictor`, the training data consists of a user matrix and an interaction matrix that is used to train an internal *LightFM* model (see section 4.3.1). The `ReciprocalWrapper` makes it possible to wrap another `Pre-dictor` and deliver reciprocal-optimized results for the predictions (see section 4.3.2).

On the `MemoryBasedPredictor` side, the framework implements a pure CF approach originally described in the online dating study [Xia+15] by Xia et al. in 2015 that is commonly referred to as “baseline RRS” and has later been referred to as RCF which stands for Reciprocal CF [Pal+20] (see section 4.3.3).

4.2.5 Evaluator

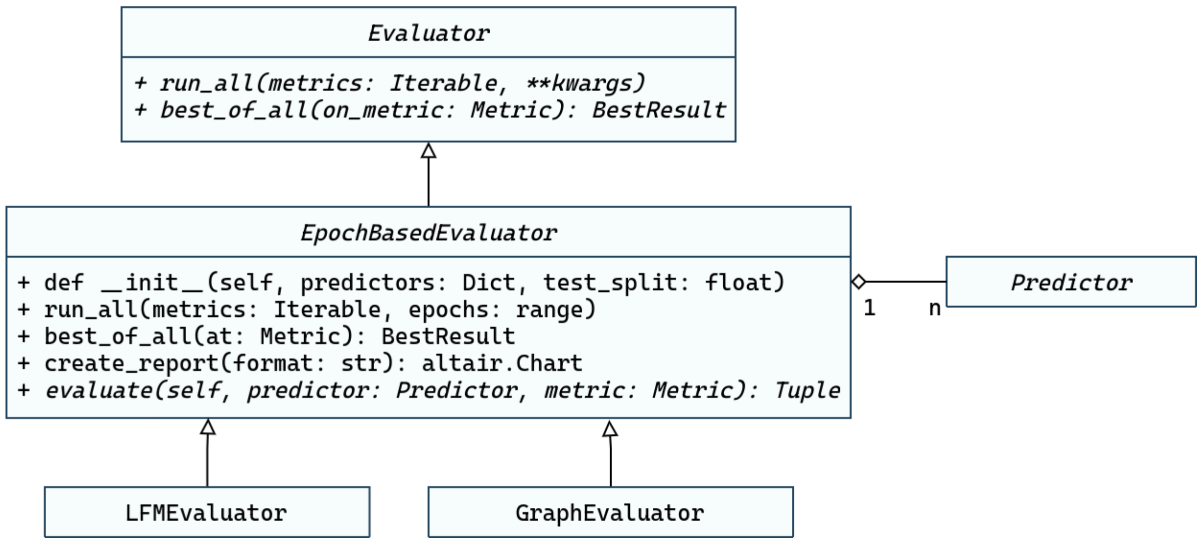

Evaluator
+ run_all(metrics: Iterable, **kwargs)
+ best_of_all(on_metric: Metric): BestResult
EpochBasedEvaluator
+ def __init__(self, predictors: Dict, test_split: float)
+ run_all(metrics: Iterable, epochs: range)
+ best_of_all(at: Metric): BestResult
+ create_report(format: str): altair.Chart
+ evaluate(self, predictor: Predictor, metric: Metric): Tuple
Predictor
1
n
LFMEvaluator
GraphEvaluator

Figure 4.2.8: Evaluator class diagram

Figure 4.2.8 shows the `Evaluator` interface which can be seen as a supervisor that measures and compares the performance of multiple `Predictors` based on historic data and is able to return the best among of them. In general, for its specialization, the `EpochBasedEvaluator`, the process is outlined as follows:

1. Initialization: Split the know known *successful connections* randomly into two disjoint sets: A *training set* (usually the majority) a *test set* (smaller, often around 20%). For implicit data, interactions indicate positive feedback – that is, each edge in the graph from section 4.2.1.1 indicates a successful connection.[8]

2. On `run_all`: Train the model `epoch`-range-wise[9] on the *training set*. Then call `eval-uate` per *each metric* and *each predictor*. The implemented `evaluate` method must validate based the given metric against the *test set* (see step 1).

[8]One could also define an edge as successful if it exceeds a certain $s(u, v)$, but here, we simplify.

[9]The `epochs` parameter accepts a positive Python `range`, e.g. `range(0, 12, 2)` means train for $5 * 2$ epochs (stop parameter = 12 is exclusive, start = 0 functions as train state reset and starting point in diagrams).

3. A researcher can now call `best_of_all` to retrieve the best `Predictor` including its epoch at a given metric (summarized by the `BestResult` object) or `create_report` to gain more detailed insights by getting served an automatically generated chart for which the framework uses *Altair* and the *Altair Viewer* package.

Additional note on step 2: The `evaluate` method must also be able to validate the given metric based on the *training set* (the method returns a 2-value `Tuple`, see fig. 4.2.8), which is a good health check to see if the correctly adapts to the training data (and converges), but it is not a meaningful indicator to measure *real-world* performance – after all, a predictor that is 100% correct based on known data is not making any predictions but plainly repeating historic data, which would defeat the purpose of RSs. Part of this process (splitting dataset and validating on the test set) is commonly referred to as *cross validation* [Kul15], as the training interactions do not contain any test interactions and vice versa.

For RSs, multiple well-known metrics exist, which will be introduced as follows.

4.2.5.1 Metrics

One way to think about a successful metric is to see the recommendations (predictions) from the user's perspective. By calling the `predict` method (see section 4.2.4.3), the recommendation subject (user) gets an ordered list of top k recommendations: If the user has mostly recommendations in the top k list that turn out to be *successful connections* based on the *test set* (see step 1 in the previous section), our system performs particularly well.

The proportion of the successful connections in the top k recommendations for a given user is commonly called *precision at k* [Kul15]. It is defined as follows:[10]

$$p@k = \frac{|\text{successful(predictions@k)}|}{|\text{predictions@k}|} = \frac{|\text{successful(predictions@k)}|}{k} \tag{4.2.2}$$

Another commonly used metric is *recall at k*, which validates the system's reliability [Pal+20]. As denoted in eq. (4.2.3), it is defined as the proportion of successful predictions and successful connections of a given user [cf. Pal+20]:

[10] The formula has been adapted from [Pal+20] to better fit into this work's notation.

$$r@k = \frac{|\text{successful(predictions@k)}|}{|\text{successful(connections)}|} \tag{4.2.3}$$

Although especially the former *p@k* eq. (4.2.2) is often by itself a very meaningful metric for validating the usefulness of a RS as it evaluates that the top *k* recommendations are actually relevant to the user, it is sometimes necessary (or wished) to aggregate both metrics to one single score [see NP19a, eq. 11]:

$$f1@k = \frac{2 * p@k * r@k}{p@k + r@k} \tag{4.2.4}$$

This metric, named *F1 Score*, makes use of the *harmonic mean* which we encountered before in section 3.1.2.1, eq. (3.1.5). Notably, if both metrics are equally important to the researcher, they need to be *aggregated*: The same holds for the aggregation of preferences for the recommendation subject and each of the top *k* recommendation objects (see section 3.1.2.1) with a RRS. The attentive reader will notice that we did *not* consider reciprocity in the metric definitions yet. In fact, the metrics were only one-sided until now. As pointed out in table 3.2.1, *reciprocity* requires overriding the *successful* operator/term:

$$\text{successful(E)} := \{(u, v) \in \text{E and } (v, u) \in \text{E}\} \tag{4.2.5}$$

Consider that we use the `build_prediction_graph` function that returns a predicted graph of recommendations. We reiterate that “the recommendation is successful if both users respond positively to it” (see table 3.2.1). Applied to the data model of *Chaos*, this means that only the bidirectional edges (connections) in the graph should be considered as successful, that is, a pair of nodes u, v where u is connected to v, and v is connected to u.

To summarize: The above definitions hold true for both, non-reciprocal and reciprocal user-to-user RSs, but the latter require an additional check if the object v is connected to the subject u. This has been shown in a similar way before by the `ReciprocalCG` in fig. 4.2.7. As pointed out in [NP19a], for RRSs, it is even more important that the recommendations are of a high quality at k (precision, eq. (4.2.2)) than being able to recall all successful connections (eq. (4.2.3)), because the “cost of failure” is higher and double-sided through reciprocity.

4.2.5.2 Example and Summary

Consider the synthetic interaction graph consisting of two study communities[11] in fig. 4.2.9 as an input: The violet nodes indicate users who study "MCD" and the blue-green nodes indicate users who study "ISE" (this is the "course" attribute/column from fig. 4.2.3).

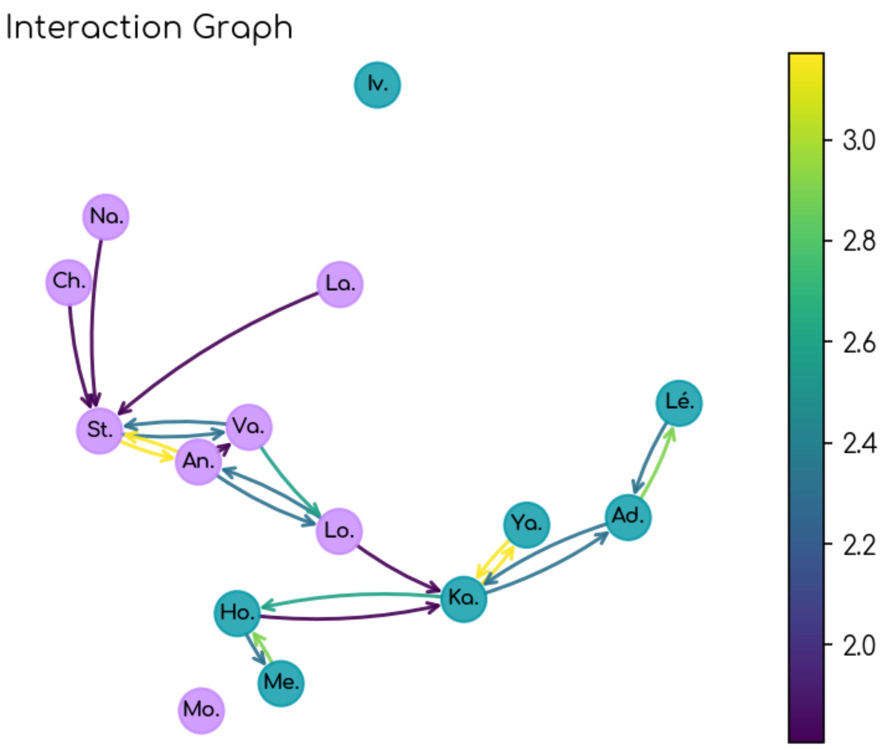


Figure 4.2.9: Interaction Graph with two different courses

[11]This is the same graph that is used in the optional notebook 4.2.1.

Listing 4.2.6 describes how the `LFMEvaluator` (which uses the evaluation algorithms provided by the *LightFM* package) is used in conjunction with multiple differently configured `LFMPredictors`.

```
 1 hp = {'learning_rate': 0.003, 'no_components': 32}
 2 evaluator = LFMEvaluator({
 3   'Hybrid, course with ID':
 4     LFMPredictor(LFMTranslator(dm, ['course']), **hp),
 5   'Hybrid, course + popularity with ID':
 6     LFMPredictor(LFMTranslator(dm, ['course', 'popularity']), **hp),
 7   'Collaborative Filtering only':
 8     LFMPredictor(LFMTranslator(dm, []), **hp)
 9   }, translator.interaction_matrix, test_split=0.3
10 )
11 evaluator.run_all(epochs=range(0, 144, 4))
12 evaluator.create_report().show()
```

Listing 4.2.6: Evaluation of 3 different predictors

All of the models share the same hyperparameters, denoted by `hp` in the first line of listing 4.2.6 (a learning rate of 0.003 and and a dimensionality of 32). For the purpose of this example, we intentionally overfit (massively) here by training each predictor for a total of 140 epochs with 35 data points (line 11). The interaction graph is split into 70% training data and 30% test data (see line 9). Afterwards, we create a report in form of multiple charts and render it, see fig. 4.2.10.

The upper row in the diagram fig. 4.2.10 shows the results based on the *train* data set (y-axis label "train"), the lower row shows the results based on the *test* data set (y-axis label "test"). Each column represents a predictor (the titles are analogous to the dictionary keys/identifiers in listing 4.2.6). As shown by the raising lines, each epoch (x-axis) fits the model more to the training data, but also more to the test data in this specific example.

Among the already known *precision* (eq. (4.2.2)) and *recall* (eq. (4.2.3)) metrics, this implementation additionally supports one other metric that we did not examine yet, the AUC score. AUC is the probability that for a given user, a randomly chosen successful connection is ranked higher than a randomly chosen connection that is unsuccessful [Goo20] [Kul15] (for implicit feedback, every edge that is *not* in the interaction graph). Other than the metrics

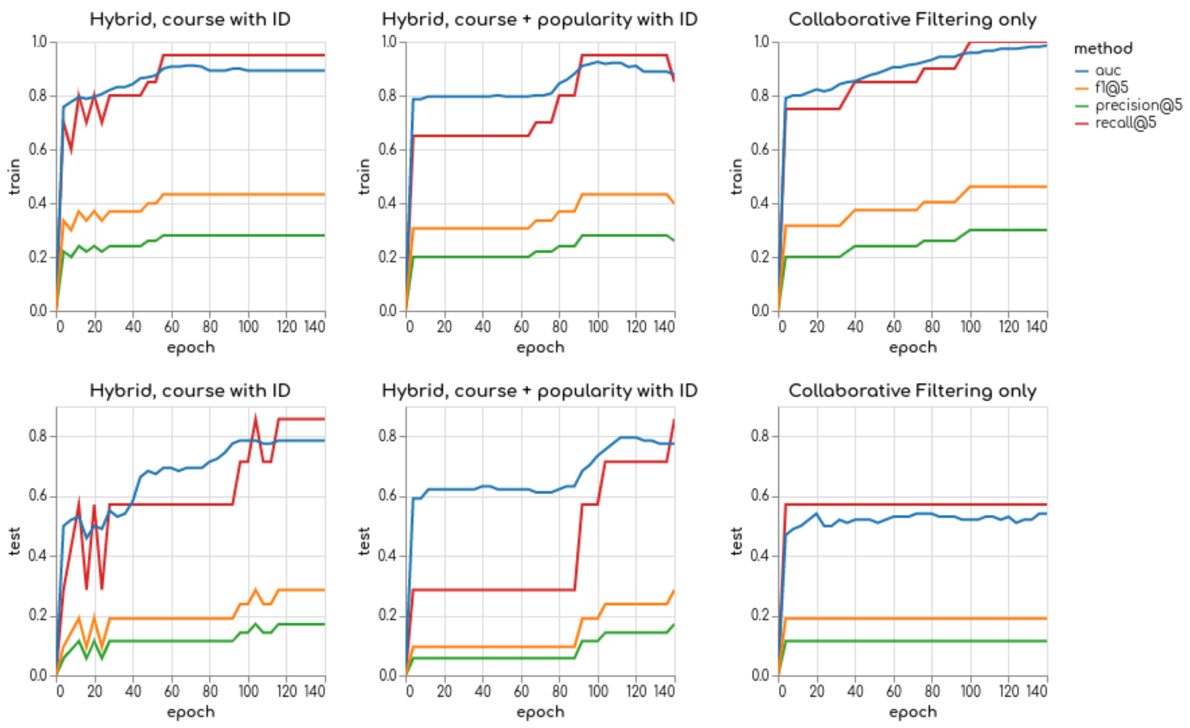


Figure 4.2.10: Evaluation report: Chart comparison of 3 predictors and 4 different metrics

before, AUC is scale-invariant: 1.0 means the RS is always correct and 0.0 means the model is always wrong (the middle of 0.5 represents a random-based predictor, i.e. simply choosing two random nodes).

The results in fig. 4.2.10 are interpreted as follows: While the rightmost CF model performs best on the training data in the first row (it has plenty of room to adjust each individual training user's embeddings represented by the indicator feature, see section 2.2.3), it performs worst on the test data in the second row, with an AUC that is slightly better than random. The two leftmost *hybrid* predictors have the advantage of embedding the "course" feature: There is a clear pattern of two communities; the graph could be partitioned into two communities by simply cutting the connection between the nodes "Lo." and "Ka.". The underlying algorithm (see section 4.3.1) learns that "MCD" users prefer "MCD" users and "ISE" users prefer "ISE" users. The hybrid model in the center adds *popularity* as a feature (see section 4.2.3 for feature extraction). This experimental feature has been added with the intention to alleviate the *popularity bias* (see section 2.3.3). To some level, the model learns the relation between the nodes' popularity (for instance, do "medium"-popular users prefer other users with a comparable popularity? If so, the model will learn that). Moreover, we can clearly see the overfitting, especially with the “Collaborative Filtering only” model, since the maximum is reached at a very early epoch. This suggests that we should tune the `epochs` and/or the `learning_rate`. Finding effective and efficient parameters is important and time-consuming, but Chaos aims to help the researcher with finding the best predictor by offering the comparative report (fig. 4.2.10) for their parameter configurations and combinations. The described learned behavior is also indicated in listing 4.2.7 where we sample recommendations based on the best predictor for the $F1@5$ metric:

```
1  res = evaluator.best_of_all('f1')
2  print(f"Best F1: '{res.predictor}' @ epoch {res.epoch} with {res.value}")
3  # Out:  Best F1: 'Hybrid, course with ID' @ epoch 104 with 0.286
4  predictor = evaluator[res.predictor]
5  print(best_predictor.predict('St.'))
6  # Out: {'Lo.': 0.95, 'An.': 0.81, 'Va.': 0.78, 'Mo.': 0.58, 'Na.': 0.56}
7  print(best_predictor.predict('Iv.'))
8  # Out: {'Ho.': 1.00, 'Ya.': 0.86, 'Lé.': 0.83, 'Ka.': 0.77, 'Ad.': 0.59}
9  print(best_predictor.predict(User.from_data({'course': 'MCD'})))
10 # Out: {'St.': 1.00, 'Lo.': 0.85, 'An.': 0.71, 'Va.': 0.70, 'Mo.': 0.56}
```

Listing 4.2.7: Recommendations from best predictor at F1 metric

Listing 4.2.7 shows how we first select the “Hybrid, course with ID” model that performed best on the F1 metric. Thereafter, in lines 6 to 10, we produce top $k = 5$ predictions recommendations by using `predict`, with relative scores that represent the model’s confidence that the order is correct: First, we predict for the *non-cold-start* user “St.” studying “MCD”. The result shows that it does not simply (re)produce recommendations based on the direct neighbours. Next, the prediction for the *cold-start* user “Iv.” with course “ISE” confirms that only users from the course “ISE” are recommended. Lastly, the prediction based on an inline-constructed user for the data `{'course':'MCD'}` highlights the powerful advantage of `LFMPredictor`’s hybrid capabilities inherited from *LightFM*: Users can be estimated on-the-fly as a sum of their latent feature vectors, see section 2.2.3.1, without prior training. For this purpose, the `LFMPredictor` internally constructs a weighted feature matrix for the provided user (see the upcoming section 4.3.1 for user-to-user details).

Example 4.2.2: Bias - Good or Bad?

We introduced bias here: The model will learn that the two communities are historically mostly separated and that they prefer to stay among themselves. Algorithmic specialization always requires some degree of bias – in that sense, the *inductive bias* (section 2.3.3) is a potentially good mechanism for prediction, as "learning involves the ability to generalize from past experience in order to deal with new situations that are 'related to' this experience." [Mit80] (**1980** paper about ML-bias).
It would be entirely different if a *human* chose to only include a subset of nodes so that "an underrepresentation or overrepresentation of observations from a segment of the population" [HDB20] is introduced. In that case, the model would suffer from a *sampling bias* (also called *selection bias*) which typically happens in the data generation process and should be avoided at all cost.
If interdisciplinary work between the two study communities is wished, one could reinforce these connections by assigning a higher strength (e.g. $2 * s(u, v)$ eq. (4.2.1)) to *inter*-community connections than *intra*-community connections. It often depends on the objective and leaves further room for discussion, see also [HDB20], especially "The world as it should be vs. the world as it is".

Finally, two important considerations have to be made: Training based on an interaction graph with only few nodes and edges as shown in fig. 4.2.9 typically results in overfitting very fast (i.e. in an early epoch, dependent on the learning rate). Additionally, it implies that there are only a few edges in the test set available, which explains the relatively high variations between the data points in fig. 4.2.10; subsequent model training runs would perform very differently depending on the random set of edges to be chosen in the train/test set, partly leaving some nodes to cold-start (e.g. consider that both edges from and to user "Me." in fig. 4.2.9 would be removed from the train set and put in the test set). Additionally, full reproducibility is not possible as the `LFMPredictor` uses *LightFM*'s parallelized version of SGD with random initialization, always resulting in a non-deterministic output per run (unless fixed to one single thread and a consistent random seed which can be accomplished by properly calling the constructor, but leads to a performance penalty).

Second, the `LFMEvaluator` does not calculate *reciprocal* metrics (see eq. (4.2.5)), as it relies on *LightFM*'s evaluation algorithms, a library which was not developed for reciprocal recommendations (but as shown in the following section 4.3.1, the library is suitable to be adapted accordingly on the recommendation side). For cross-algorithm and cross-validation based reciprocal evaluation, the `GraphEvaluator` prototype is currently[12] being tested, which will support all implementations of the `Predictor` interface by internally calling the `build_prediction_net` function (see outlook section 5.3.3). At a later point, the `LFMEvaluator` class in listing 4.2.6 can simply be interchanged and the other code parts stay the same, because both classes are descendants of the `EpochBasedEvaluator`.

[12]at the time of writing (January 27, 2021)

4.3 Recommendation Algorithms

In the following, we explore the supported reciprocal recommendation algorithms of Chaos. All of the mentioned algorithms are based on the assumption that *similar users like similar users*. This is supported by the majority of RRS papers [Pal+20] [Ake+11] [Xia+15] and a precondition for CF to work well – just as in traditional RSs (see section 2.2.2), we assumed that *similar users like similar items*.

4.3.1 LFMPredictor - Hybrid Model using LightFM

The latent factor model *LightFM*, which has been introduced in section 2.2.3, usually works with users and *items*. For Chaos, we need to adapt the model input in order to make user-to-user recommendations possible. We divide the alterations into two categories, content and collaborative – both sides representing the input of the LightFM model's hybrid nature:

1. Content: Only one feature matrix is needed and constructed, that is, the users'. It is inputted for both, the item matrix parameter as well as the user matrix parameter [cf. KC20]. Referring to section 2.2.3.1, we set $F^U = F^I$. Therewith, we only change the *external* parameterization and not the *internals* of the LightFM model, which means that both earlier introduced equations eq. (2.2.11) and eq. (2.2.12) are still valid. To convert the data model in section 4.2.1 to *LightFM*'s model, the `LFMTranslator` iterates through each row of the user profile `DataFrame` and sets a normalized weight to each included content feature of the user, with $f_u \subset F^U = F^I \supset f_i$ (see section 2.2.3.1).

2. Collaborative: The interaction graph is translated by going through each edge $(u, v) \in E$ and setting $s(u, v)$ (eq. (4.2.1)) for the uth row and vth column in the interaction matrix format the *LightFM* model expects. In accordance with Chaos' data model, the matrix is comparable to an adjacency matrix (section 4.2.1.1) and has a shape of $|V|^2$ with V denoting vertices (users), but its true memory footprint is much smaller, as *LightFM* uses sparse matrices [KC20].

These are the only alterations of model input parameters that were needed. From a purely practical developer's perspective, it is important to keep these in mind, as when calling the LightFM model's API methods, parameters are named according to a user-to-item RS, that "is not aware of" that it actually contains user-to-user data. Overall, by not changing the internals of the *LightFM* model or re-implementing parts of it, we ensure a high reusability of *LightFM*'s code base and a high transparency of the `LFMPredictor`: Interested readers can simply refer to the LightFM paper [Kul15] (if necessary complemented/accompanied by the software manual [KC20]) and replace the above alterations in the formal model description. Putting all together, if we now call the `LFMPredictor`'s `predict` method for u as shown in listing 4.2.7, we calculate the preferences for each candidate in $CG(u)$ and get the k top recommendations objects $v \in RS(u)$:

$$RS(u) = \{v \,:\, p(u,v) > p(u,w)\; \forall v \in R \subseteq CG(u),\; \forall w \notin R \subseteq CG(u)\} \text{ with } |RS(u)| = k \tag{4.3.1}$$

Equation (4.3.1) shows the previous eq. (3.1.1) with the added candidate generator $CG(u)$ (from section 4.2.4.2) and limiting k. Internally, $p(u,v)$ returns a score which *LightFM* accomplishes by calculating the dot product [see Kul15, eq. 1][13] of q_u (eq. (2.2.12)) representing the subject user u, and p_i (eq. (2.2.11)) representing the object user v. The scores are normalized in a post-processing step by *Chaos*, as the *LightFM* model outputs a value that is potentially negative. The score value is used solely to order recommendations relative to the user (the higher, the better) [KC20], as we are dealing with an implicit feedback model where the goal is to optimize and predict the *ranking* (and *not* rating values). The post-recommendation *normalization* that *Chaos* performs is only a means of mapping the user-scoped ranking to a value in the interval $[0.0, 1.0]$ as described in section 3.1.2.1.

At this point, we successfully transformed the hybrid user-to-item RS model *LightFM* to a user-to-user RS simply by altering its input values, but it does *not* fulfill the reciprocal criteria from eq. (3.1.2) yet:

$$RRS(u) = \{v \,:\, v \in RS(u) \text{ and } u \in RS(v)\} \tag{4.3.2}$$

As repeated in eq. (4.3.2), it requires that each recommendation object v also has u in the recommendation list (compare with the more detailed eq. (3.1.3)). For this purpose, Chaos provides a universal `ReciprocalWrapper` that aligns the scores of a regular non-reciprocal `Predictor` by fusing the preferences of u and v, described in the next section 4.3.2.

[13] For completeness and correctness: The original formula is slightly more extensive, involving an added *bias term* for the features $(b_i + b_u)$ and a *sigmoid function* f applied to the result.

Notably, a conceptually similar approach has been introduced with LFRR (Latent Factor Reciprocal Recommender) in [NP19b], but in contrast to Chaos' `LFMPredictor`, a *two-class* RRS is proposed that calculates the dot product of "Male Users" and "Female Users" (and vice versa), as illustrated in fig. 4.3.1. In terms of Chaos' data model language, LFRR would only work on *bipartite* interaction graphs, as it focuses on heterosexual dating SNSs. As noted in the analysis of requirements in section 3.3.2, this is very restrictive for a *multi-purpose* framework: For instance, the graph presented in fig. 4.2.9 could not be used as an input, as its users are not dividable into two disjoint partition sets (see section 3.1.2.2 for two class criterion). Additionally, the LFRR model is limited to explicit feedback (*like* and *dislike* interactions) over observed values only with an added regularization term, thus using a formal definition that is essentially similar to the matrix factorization objective eq. (2.2.9), cf. [NP19b, eq. 10]. For this reason, the same disadvantages of explicit feedback over observed ratings only, as mentioned in section 2.2.3.2, apply here.

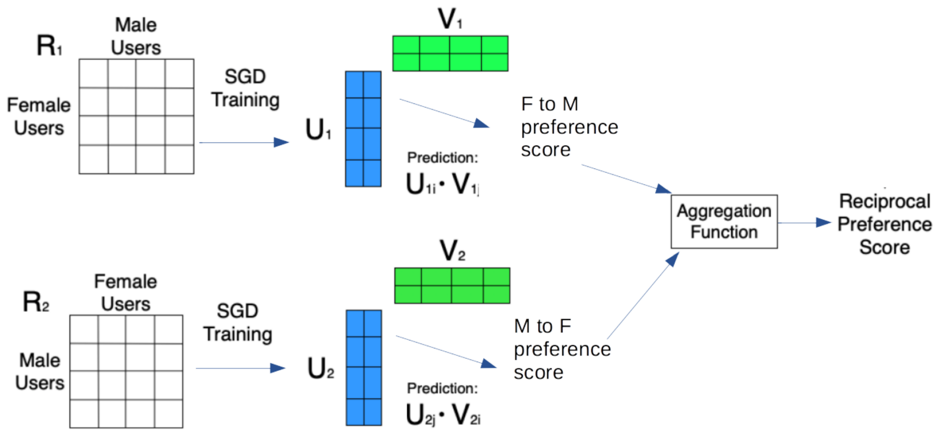


Figure 4.3.1: Conceptual overview of the LFRR model [NP19b]

Source: [Pal+20]

4.3.2 ReciprocalWrapper - Reciprocity Enabler

The `ReciprocalWrapper` acts as a symmetric preference fusing facade: It does not contain any recommendation algorithm on its own, but instead delegates recommendation tasks to the aggregated `Predictor` (see UML fig. 4.3.2).

The basic algorithm is outlined as follows, referring to listing 4.3.1:

- Line 5: Predict k objects for subject user u (see eq. (4.3.1)).
- Line 6 - 11: Predict for each object v the score for user u (see eq. (4.3.2)). Retrieve recommendations for v from (optional) cache or put to cache if needed.
- Line 11: Fuse preferences of each pair u, v according to an aggregation strategy (see eq. (3.1.3)), more details below.
- Line 13 to 16: Optionally, calculate rank violations to measure the effectiveness of the aggregation strategy (or the `ReciprocalWrapper` overall).

```
1  def predict(self, u, k: int = 5) -> Scores
2    ku = round(k * self._ku_factor)
3    kv = round(k * self._kv_factor)
4
5    u_scores = self._u2u_predictor.recommend(u, ku)
6    for v in u_scores:
7      # Retrieve predictions from cache if flag set:
8      if (v_scores := self._from_cache(v, kv)) is None:
9        v_scores = self._predictor.recommend(v, kv)
10       self._put_to_cache(v, v_scores) # Put to cache if flag is set
11       u_scores = self._aggregation_strategy.fuse(u, v, u_scores, v_scores)
12
13   if self._stats_enabled:
14     scores = list(u_scores.values()) # Scores are not yet sorted
15     self._stats['rank_violations'][u] = sum([scores[i] < scores[i + 1]
16                                              for i in range(len(scores) - 1)])
17   return {v:s for v,s in sorted(u_scores.items(), key=lambda vs: -vs[1])[:k]}
```

Listing 4.3.1: Fusion of preferences using an aggregation strategy

The aggregation strategy from line 11 can be exchanged to any implementation of the abstract `AggregationStrategy`, as shown in the class diagram fig. 4.3.2. The default behavior of the `fuse` method is to call the `score` method with $p(u, v)$ and $p(v, u)$ (i.e. it resolves to $rp(p(u, v), p(v, u))$ as in eq. (3.1.3)), modify the score dictionaries of u and v (in-place), and return the score dictionary of u. Although this might seem overly complicated at first, it has been designed with keeping a broad spectrum of possible aggregation use-cases in mind (for instance, also including the intersection of both preference lists by some extinct to reweigh scores based on recommendation list similarities). The `ku_factor` and `kv_factor` are multipliers to increase the recommendation list sizes of the subject and each object relative to k to increase the chances that `u` is found in the list `v_scores`. The algorithm is therewith of an approximate nature if `ku` or `kv` is (too) small, but it highlights the increase in complexity through reciprocity.

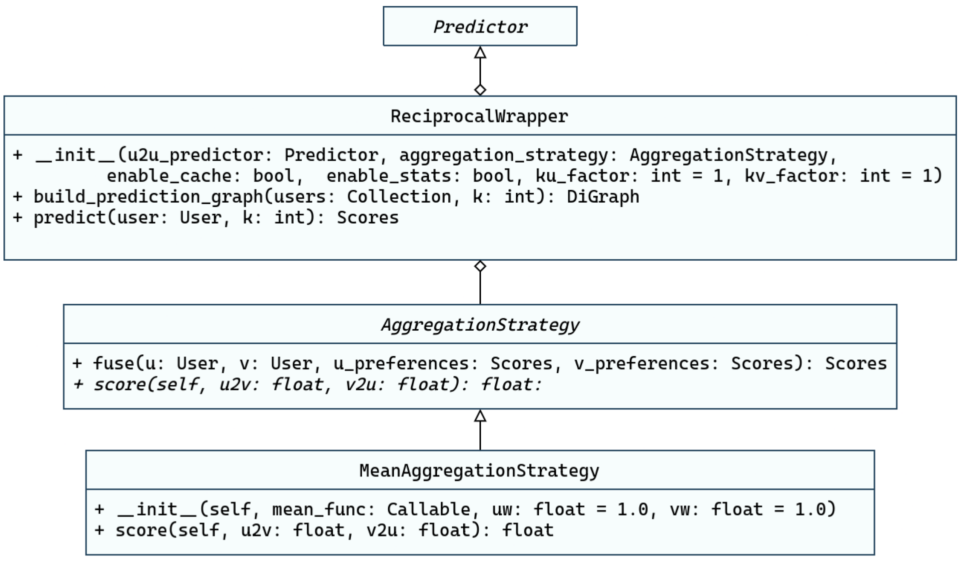

Predictor
ReciprocalWrapper
+ __init__(u2u_predictor: Predictor, aggregation_strategy: AggregationStrategy, enable_cache: bool, enable_stats: bool, ku_factor: int = 1, kv_factor: int = 1)
+ build_prediction_graph(users: Collection, k: int): DiGraph
+ predict(user: User, k: int): Scores
AggregationStrategy
+ fuse(u: User, v: User, u_preferences: Scores, v_preferences: Scores): Scores
+ score(self, u2v: float, v2u: float): float:
MeanAggregationStrategy
+ __init__(self, mean_func: Callable, uw: float = 1.0, vw: float = 1.0)
+ score(self, u2v: float, v2u: float): float

Figure 4.3.2: `ReciprocalWrapper` and `AggregationStrategy` classes

Example 4.3.1: Expensiveness of Reciprocity

Fusing the preferences as described in section 3.1.2.1 is at least `ku`-times (see line 3) more computationally expensive than traditional one-sided recommendations. In the worst case, the re-ranking does not even occur, thus not necessarily improving the predicted reciprocity (e.g. if `ku`/`kv` is too small). The increase in run-time is commonly found in RRS algorithms, which is why they are often subject to optimizations [NP19b] [Pal+20]. This is also highlighted by the algorithm in the following section.

The `MeanAggregationStrategy` accepts a function that is used to calculate in `score`, with the added possibility of prioritizing the preferences of u or v (as shown before in eq. (3.1.4)). The class also features a diverse set of pre-defined statistical means and functions, see fig. 4.3.3. It supports all of the *Pythagorean means* as mentioned in section 3.1.2.1, the *quadratic mean* (*root mean square*) and the *cross-ratio uninorm* [App+17]. Although the harmonic mean is a sensible choice in many cases [Pal+20], including more functions has been motivated by [NP19a], which puts the different strategies into comparison. Interestingly, with its mixed behavior, the *cross-ratio uninorm* has been outperforming the other means in an evaluation for a large dating SNS by using the *RCF* algorithm [NP19a], to which we will come next.

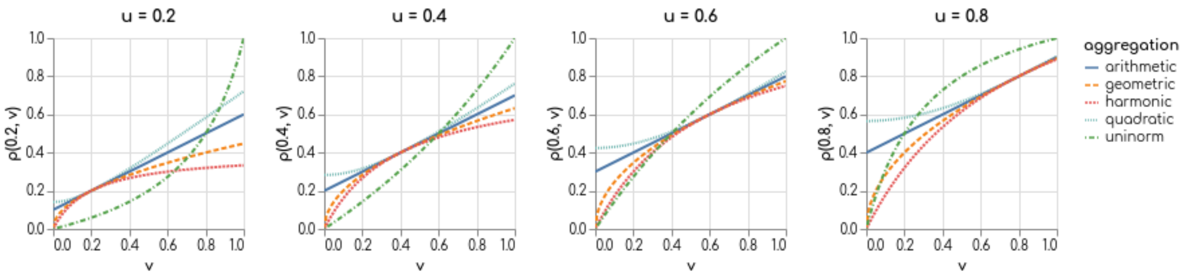


Figure 4.3.3: Samples of $p(u \in \{0.2, 0.4, 0.6, 0.8\}, v)$ suitable for `MeanAggregationStrategy`

4.3.3 RCFPredictor - Reciprocal Collaborative-Filtering

In the following, we describe a reciprocal in-memory CF method (see section 2.2), which has been introduced by Xia et al. in 2015 [Xia+15]. Initially tested on a large heterosexual dating SNS and therewith specialized in bipartite networks (two-class RRS) [Xia+15], is is also usable in *single-class* RRS approaches like Chaos – given an appropriate similarity measure. As it was one of the first in-memory CF methods that fulfills the definition of reciprocity (eq. (3.1.3)) and outperformed[Xia+15] the early reciprocal CBF approach RECON [Piz+10b], it has later been referred to as Reciprocal Collaborative Filtering (RCF) and considered as a baseline algorithm for RRS by some studies [Pal+20] [NP19a].

RCF essentially provides an algorithmic *frame* where a neighbourhood exploration function n and a similarity measure function *sim* can be placed in. Two of these behavioral similarity measures are suitable to be used in a *single-class* RRS [Pal+20] and described as follows. The *interest similarity* captures the interaction *sending* similarity of two nodes w and x [Xia+15][14]:

$$sim_I(u, v) := \frac{|\, n^+(u) \cap n^+(v) \,|}{|\, n^+(u) \cup n^+(v) \,|} \tag{4.3.3}$$

As denoted in eq. (4.3.3), it is given by the well-known *Jaccard-coefficient* between the set of nodes to which u has sent an interaction and the set of nodes to which v has sent an

[14]Our notation is based on a node's out-degree $\deg^+$ and in-degree $\deg^-$, not to be confused with like/dislike.

interaction. By contrast, the *attraction similarity* captures the interaction *receiving* similarity of two nodes u and v [Xia+15]:

$$sim_A(u,v) := \frac{|\, n^-(u) \cap n^-(v) \,|}{|\, n^-(u) \cup n^-(v) \,|} \tag{4.3.4}$$

RCF does not need any separate model like the `LFMPredictor` in section 4.3.1 and works natively on Chaos' `DataModel` (see section 4.2.1). As a pure CF algorithm, it also does not consider any of the user profile data (section 4.2.1.2) and solely relies on the interaction graph (section 4.2.1.1). The framework provides a close-to-definition implementation of the neighbourhood exploration functions n^- (`in_neighbours`) and n^+ (`out_neighbours`), as well as the similarity functions sim_I (`interest_similarity`) and sim_A (`attraction_similarity`):

```
def in_neighbours(self, u): return {e.u for e in self.graph.edges(to=u)}
def out_neighbours(self, u): return {e.v for e in self.graph.edges(from=u)}

def jaccard(self, s1, s2): return len(s1 & s2) / len(s1 | s2) # helper
def interest_similarity(self, u, v): # → sim_I
  return self.jaccard(self.out_neighbours(u), self.out_neighbours(v))
def attraction_similarity(self, u, v): # → sim_A
  return self.jaccard(self.in_neighbours(u)), self.in_neighbours(v))
```

Listing 4.3.2: Neighbourhood exploration and similarity function definitions

These are the configuration options of the algorithm. The algorithm itself is implemented compliant to the pseudo-code from [Xia+15, Algorithm 1][15] in a seamless way:

```
def predict(self, u, k: int = 5) -> Dict[str, float]:
  neighbours = in_neighbours  # choose inbound neighbours (→ n⁻)
  similarity = interest_similarity  # choose interest similarity (→ sim_I)
  scores = {}
  for v in [self.graph[c] for c in self._cg.retrieve_candidates(u)]:
    score_uv = 0.0
    score_vu = 0.0
```

[15] alternatively compare with [NP19a, Algorithm 1] or [Pal+20, Algorithm 2]

```
8       # Explore and compare v's neighbourhood
9       v_neighbours = neighbours(v)
10      for vn in v_neighbours:
11          score_uv += similarity(u, vn)
12      # Explore and compare u's neighbourhood
13      u_neighbours = neighbours(u)
14      for un in u_neighbours:
15          score_vu += similarity(v, un)
16      # Normalize
17      if len(v_neighbours) > 0: score_uv /= len(v_neighbours)
18      if len(u_neighbours) > 0: score_vu /= len(u_neighbours)
19      scores[v.name] = self._aggregation.score(score_uv, score_vu)
20    return {v: s for v, s in sorted(scores.items(), key=lambda vs: -vs[1])[:k]}
```

Listing 4.3.3: RCF algorithm implementation in Chaos

For simplicity, we describe one instantiation of the generic algorithm, where the neighbour function is set to n^{-} and the similarity function sim_I (see eq. (4.3.3)) is used (lines 2 and 3). This represents the “CF2” configuration in [Xia+15, p. 10] and the described “Algorithm 1” in [NP19a]. The latter source used this configuration as a reference implementation to compare various preference aggregation functions (as noted at the end of section 4.3.2).

The example listing 4.3.3 in accordance with eq. (4.3.3) and [Xia+15] demonstrates that Chaos enables researchers to implement graph algorithms that are very close to their scientific representation. We included two of the commonly found theoretic definitions (eq. (4.3.3) and eq. (4.3.4)) specifically for this reason here, as they highlight that Chaos aims to be the *link* between mathematical definitions and real-world implementations; a link that helps with the researcher’s overall comprehension and therewith contributes to reproducibility. Furthermore, with RCF joining the team of Chaos’ algorithms, we support a relevant baseline algorithm for future evaluation and comparison of algorithms.

4.4 Chaos for GitHub

With *GitHub* being one of the major cloud solutions for collaborative *Git*-based version control that offers a DevOps-toolchain for the lifecycle of information systems, it is a particularly interesting platform where users could benefit from social and reciprocal recommendations for finding *code collaborators*, *project members* or suitable *skill sharing* partners in general. At first sight, it might have a strong focus on *repositories* (where code is hosted), which would be an argument that it would profit more from a traditional user-to-item RS. Still, the main drivers behind the *repositories* are *humans* after all, who often search for other collaborators to bring a project forward.

Hence, it serves as an exciting platform to try out Chaos where we can demonstrate how to utilize a public third-party service to enable reciprocal recommendations, with real production data.

Notebook 4.4.1: Interactive GitHub Chaos Scenario

This part of the thesis is interactive. Please refer to "Scenario 2: Chaos for GitHub" for instructions. The goal of the scenario is to generate a highly personalized network for the reader's provided GitHub username that can be used to train a latent factor model. At the end, we visualize the embeddings with the *TensorBoard Projector* so that they can be explored interactively.

4.4.1 Data Generation

First, we explore the interaction and user profile data. Knowing the platform and exploring its data is one of the first (and most important) key steps. Relating to the preliminary workflow in fig. 4.1.1, it contributes to the first task "Select features/interactions". Next, after we have decided which interactions and features are useful (section 4.4.1.1), we need to source the original data (section 4.4.1.2). Because we work in a restricted environment without access to a full open dataset, bound to a specific API request limit, we propose an algorithm that focuses on finding reciprocal user connections suitable for training (section 4.4.1.3).

4.4.1.1 Interactions and User Profile

In this section, we start with the interactions between GitHub users to highlight our findings from the introduction to reciprocity (section 3.1) that the “root of reciprocity” lays in the environment’s existing interactions. Thereafter, we consider important GitHub user profile data for the CBF part of our hybrid model to be trained.

On GitHub, users can interact with each other directly (e.g. collaborating or following) or indirectly by interacting with a user’s repository (most commonly watching or cloning). In table 4.4.1, we identify and describe representative interactions and estimate the *implicit* feedback strength for each. On the platform, a publicly usable *explicit* feedback mechanism that directly works on repositories or users does not exist (therefore, the stars are not to be confused with explicit feedback as shown in fig. 2.1.1). This underlines the advantages of implicit feedback mentioned in section 2.2.3.2, especially the wide data availability, and it highlights Chaos’ strength as a framework specialized in this type of feedback (see section 4.2.1.1).

Table 4.4.1: Representative GitHub interactions and (estimated) implicit feedback strength.

Interaction	Entity	Symmetric	Feedback Strength
Follow	User	○	★★★★★
Collaborate	User	✓	★★★★☆
Star (Like)	Repository	○	★★☆☆☆
Watch	Repository	○	★☆☆☆☆
Clone	Repository	○	☆☆☆☆☆

Following another user can be seen as a powerful *one-time* feedback signal towards liking the other user. *Collaboration* is similar, but slightly less bound to the user and instead to a project work (and can occur *multiple* times), adding a star to a repository potentially exhibits interest towards the user (comparable to liking a post in a classic SNS) – whereas we consider *watching* or the extremely common *cloning* of a repository not as a direct sign of positive feedback towards the user, although *watching* at least shows a small amount of interest in the user’s work (therefore rated with a strength of 1 out of 5 stars). Please note that the strength column only provides a guideline. For instance, if the objective is to find suitable project partners (collaborators) and there are strong signs that the collaboration was successful (i.e. similar commit count or activity in the project), then the *collaborate* interaction might be a more powerful feedback signal than *follow*. Lastly, table 4.4.1 supports our argumentation that GitHub is likely to profit from *reciprocal* recommendations because at least one *symmetric* interaction is present (see previous section 3.1.2.3).

Referring to the "Entity" rows in table 4.4.1, an interaction with a repository can be interpreted as a *transitive* (or indirect) implicit feedback signal towards the user who owns the repository (connecting start and end node):

$$\text{Alice} \xrightarrow{\text{likes}} \text{Repository} \xrightarrow{\text{created by}} \text{Bob} \implies \text{Alice} \xrightarrow{\text{likes}} \text{Bob?} \tag{4.4.1}$$

Following this inference, there is a probability that Alice also likes Bob. Note that this might be applied to other user-to-item relationships, too (it is not specific to GitHub repositories) – for instance, given that Alice likes a movie, there is a probability that she also likes its director. It highly depends on two points: First, on the degree the owner identifies herself/himself with the item and second, how much the interaction-emitting user is aware of the item-owner-relationship and includes the information in her/his decision process. For completeness, it should be noted that there are a lot of other interactions which we left out in the exemplary table 4.4.1 that have a more complex feedback interpretation. To name a few: accepting pull requests, performing code reviews, writing an issue, answering to an issue, assigning a user to an issue, adding a project member to an organization, opening a discussion (recently introduced feature), adding a reaction emoji (e.g. thumbs-up) and many others. [Tea20]

Towards GitHub *user profile data*, users can optionally add regular textual profile features such as location, public e-mail, company and bio (short text describing the user). Additionally, users can upload an avatar image [Tea20]. Equally importantly, users define part of their user profiles through their repositories. For instance, a developer who worked on a lot of huge (or popular, e.g. in sense of received stars) Python projects is likely to be very proficient in this programming language (without the need of stating it explicitly anywhere else). Moreover, *GitHub* recently introduced a special repository (named after the username) that can be used for the users' self-representation to add a more detailed `README` about themselves that is automatically rendered on the user's profile page.

Example 4.4.1: Infer complex profile features

In the above, we only considered clearly visible features. However, features that are in fact not immediately obvious (i.e. literally latent) can have a deep meaning – consider the following example:

- Bob hosts a collection of beginner examples for the *Flutter* SDK in one repository, with a total of 13 collaborators and 150 stars, published in October 2020. He promoted his repository using various social media channels.
- Alice implements a *Flutter* library that enables state management for use with asynchronous streaming, with a total of 3 collaborators and 30 stars, published in August 2020. She makes heavy use of GitHub *Workflows* to automatically unit- and integration-test the library.

Superficially, both repositories have in common that they are written in *Flutter*, but otherwise, they can be interpreted very differently: Although *stars* might be a good indicator for *usefulness*, they are not necessarily good for estimating a user's *skills*. Bob himself might not be highly-skilled in Flutter, as his repository is a collaborative work of a collective nature (notably, each collaborator has an own "skill-share" here) whereas Alice can be viewed as a Flutter expert who follows modern practices. Therefore, given complex feature engineering steps, we could *infer* the following exemplary features to better *estimate* the skill level: repository type, contribution factor (degree of own work), novelty vs. reproduction, code quality, best-practices, promotion/publicity factor and publication time. Putting these together, the features can be boosted with the *usefulness* given by the stars to estimate the *skill*.

In summary, the many social capabilities indicate that GitHub can indeed be classified as a SNS. The recent introduction of repository-based *discussions* and the ability to add extended user profile information through a highly personalized repository reflects this direction of the popular code collaboration platform.

4.4.1.2 Chaos GQL Specification

The data generation process is accomplished by using the `GQLSource` class (see fig. 4.2.4) that retrieves data from a *GraphQL* API endpoint. The *Chaos GQL specification* defines the relationship between user interactions and a *GraphQL* `query_definition` with a very simple yet powerful DSL. It consists of three key elements:

- `query_definition`: The query is a GQL fetching operation, which is idempotent and non-mutating, similar to the REST paradigm's GET. In the specification, it is important that the query accepts at least a string parameter with the username that identifies the user to fetch. Multiple fragments can be referenced (see next item).
- `fragments`: Fragments are a built-in GQL method to compose a query consisting of multiple scoped parts. In the specification, we utilize them to reference a specific part of the query, as fragments can be easily referenced by their name, but also to further aid readability as only a single query definition is supported.
- `interactions`: This is an extended version of the interaction specification from listing 4.2.2, containing 3 special keys `fragment_in` (incoming interaction), `fragment_out` (outgoing interaction) and `fragment` (symmetric interaction). They refer to the fragment identified by the unique fragment name.

The specification becomes more clear with an example that is actually used for *GitHub* (some elements are trimmed/collapsed for conciseness):

```
query_definition: >-
  query QueryUser($login: String!) {
    user(login: $login) {
      ...userInfo
      aboutRepo: repository(name: $login) { ...readmeContent }
      repositories(privacy: PUBLIC, first: 10) {
        nodes { ...repoInfo, ...repoMentionableUsers, ...repoStargazerUsers }
      }
      ...followers, ...following, ...starredRepoOwners
    }
  }
fragments: >-
  fragment userInfo on User {
      name, bio, avatarUrl, email, company, location, # ...
  }
  fragment repoInfo on Repository {
      name, description, stargazerCount, forkCount
      languages(first: 5) { nodes { name } }, # ...
  }
  fragment readmeContent on Repository {
    object(expression: "HEAD:README.md") { ... on Blob { text } }
  }
  fragment repoMentionableUsers on Repository # { ... }
  fragment repoStargazerUsers on Repository # { ... }
  fragment starredRepoOwners on User # { ... }
  fragment followers on User { followers(first: 30) { nodes { login } } }
  fragment following on User { following(first: 30) { nodes { login } } }
interactions:
  star:
    strength: 1.0
    fragment_in: repoStargazerUsers
    fragment_out: starredRepoOwners
  communicate:
    strength: 2.5
    fragment: repoMentionableUsers
  follow:
    strength: 5.0
    fragment_in: followers
    fragment_out: following
```

Listing 4.4.1: Excerpt from original gql-spec.yaml

As shown in listing 4.4.1, the fragments are referenced in `query_definition`, e.g. `...userInfo` stands for the expansion of the `userInfo` fragment that contains the attributes `name`, `bio` etc. Some of the fragments are referenced in the `interactions` block at the end of listing 4.4.1, describing the relationship between user nodes and their respective feedback strength. The listing shows three different types of modelled interactions: `star`, `communicate` and `follow`, cf. table 4.4.1. The `communicate` interaction which relies on the `repoMentionableUsers` fragment is not equivalent to "collaborate", but an indicator that the user has previously interacted with the other user (in an undocumented way [cf. Tea20]) to be qualifiable as a "mentionable user". Therewith, the interaction is (only) an approximation for communication between users, though a very useful one as it allows us to limit the query's complexity (and therewith also the computational cost, see next section).

With *GraphQL*, the query structure mirrors the result structure. This property is extremely useful for self-documentation and aids the usability: It is always at hand what actually is returned by the service. Additionally, most of the times, only one single query is needed to retrieve a user in total (plus her/his “flat” neighbours in form of usernames), opposing to REST, where clients would need to perform multiple requests to multiple endpoints per resource resulting in a high complexity that would be much more difficult to describe in a specification. Ultimately, the use of GraphQL greatly simplifies `GQLSource`’s automatic in(tro)spection of the resulting structure to co-locate the interaction fragments for finding neighbours. In the next section, the iterative process of building the interaction graph and the user profiles by using the defined specification is explained.

4.4.1.3 Reciprocal BFS

Because the main goal when gathering training data for a reciprocal RS is in finding *reciprocal* connections, Chaos implements a modified BFS algorithm that *prioritizes* high-strength reciprocal user neighbours. The algorithm is illustrated in a simplified form as follows:

```python
def reciprocal_bfs(start_u: str, max_nodes: int, breadth: int) -> DataModel:
  graph = InteractionGraph(self._interaction_spec)
  profile_data = {}

  seen_nodes = {start_u}
  to_visit = deque()
  to_visit.append(start_u)
  for i in range(max_nodes):
    if len(to_visit) == 0:
      logger.warning(f"Stop @{i}. Could not fulfill max. of {max_nodes}.")
      break
    u = to_visit.popleft()
    interactions, profile_data[u] = query_user(u)
    graph.add_interactions(interactions)
    best_neighbours = [e.v.name for e in sorted(filter(
        lambda e:e.v not in seen_nodes,graph.bidirectional_edges(from_node=u)
      ), key=lambda e: e.strength, reverse=True)[:breadth]]
```

```
18     seen_nodes.update(best_neighbours)
19     to_visit.extend(best_neighbours)
20   return DataModel(graph, profile_data)
```

Listing 4.4.2: Simplified Reciprocal BFS algorithm of `GQLSource`

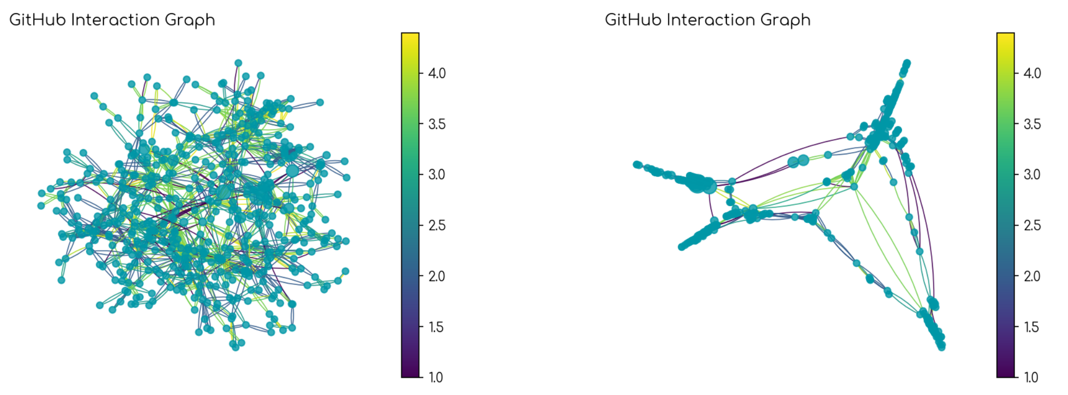
As outlined in listing 4.4.2, the BFS starts with the user `start_u` (lines 5 to 7) given by the function parameter. Then, in line 13, the current user is queried by executing the *GraphQL* query given by the `query_definition` from the specification (section 4.4.1.2). Additionally to her/his profile data (stored in the temporary dictionary `profile_data`, all of her/his interactions defined in the specification are then added to the interaction `graph` according to their respective strengths (line 14). In the next step (lines 15 to 17), the actual *prioritization* of neighbours occurs: First, all edges containing neighbours that are already “seen” are filtered out. Then, the incident edges are sorted descending by their strength and the neighbour nodes v in the edge (u, v) are added to the queue of nodes `to_visit` that still need to be visited in a first-in first-out manner (limited by the `breadth` parameter). The algorithm stops either with an early stopping criterion (if `to_visit` is empty) or if `max_nodes` could be fulfilled as specified in the function’s parameter.

For GitHub, the `GQLSource` is initialized as follows to accomplish the above-described behavior.

```
gh_source = GQLSource(spec=yaml.safe_load(open('gql-spec.yml')),
  endpoint='https://api.github.com/graphql',
  auth={'Authorization': 'bearer {token}'},
  start_user='{USERNAME}', profile_key='user', neighbour_key='login',
  breadth=7, max_nodes=5000)
```

Listing 4.4.3: Initialization of `GQLSource` in conjunction with GitHub

As demonstrated by listing 4.4.3, even though a personalized access token is required by `auth`, it has to be noted that any user on GitHub can be used as a `start_user` (GitHub users are publicly discoverable). The token is only a means to fully exploit the API limit of 5000 points per hour for the free-tier [given by Tea20, “Resource Limitations”]. The more complex a query gets, the more points it costs. Still, the query in listing 4.4.1 only consumes 1 point, therewith resulting in 5000 requests per hour and 5000 `max_nodes` as defined in



(a) Spring Layout **(b)** Spectral Layout

Figure 4.4.1: Two GitHub interaction graphs with 500 nodes with the same data, but each having a different layout

the initialization listing 4.4.2. The `profile_key` is needed to store the user's profile data according to the structure defined in the specification (listing 4.4.1) from line 3 to 10 (result is stored in a dictionary). The `neighbour_key` is used to locate the neighbour nodes – for instance, given lines 23 to 25 of the specification (listing 4.4.1), the `query_user` algorithm will recurse through the structure to find the `login` field (line 24).

At this point, it is very important to highlight that the resulting network is largely influenced by the given start node. In other words, it arguably suffers from a *selection bias* (see example 4.2.2) that is given by the start node's u past choices to some degree: The higher the `breadth` parameter (and the lower the `max_nodes`), the more the network is influenced (biased) by the start node. Setting `breadth` to a lower value will result in a graph that goes "deeper" or farther away from the start node, thus lowering the selection bias, but making the network less personalized for *each* user (not only for the start user). When using such a network to generate recommendations for *cold-start* nodes, this should be kept in mind (cf. section 4.2.5.2). Nevertheless, Chaos' built-in *reciprocal BFS* shows a unique way to build a highly personalized reciprocal network based on a third-party API.

Before continuing with the next steps, we can examine the graph that has been created with help of the algorithm: Figure 4.4.1a and fig. 4.4.1b actually show the same GitHub interaction graph with 500 nodes (for better readability, we reduced to 10% of GitHub's API limit) in two different layouts (central nodes are made slightly bigger). The spring layout (left graph) positions the graph based on the edges simulating physical spring forces pulling nodes together, and the nodes alone pushing themselves apart ("anti-gravitational") [HSS08]. The spectral layout (right graph) calculates the distances based on the *Eigenvector* (cf. section 2.3.2 *Eigenvector centrality*) [HSS08]. While both graph images are generally suited for providing an overview of the reciprocal BFS algorithm's (listing 4.4.2) output, their exploration/interpretation capabilities are limited due to their large size in combination with the static image nature. For this reason, a more interactive (standardized) graph exploration interface is planned in a future release.

4.4.2 Feature Engineering

Continuing the process in fig. 4.1.1, after sourcing the original data, we need to further process it. The previous example fig. 4.2.3 mainly showed data of a categorical and discrete (non-continuous[16]) nature. Besides this simple data type, the user profile `DataFrame` supports dynamic-length data in form of iterables (*sets* for unique data). For the GitHub environment, this is necessary and natural; users rarely enter the same amount of data, e.g. every user has a different set of skills.

Figure 4.4.2 provides an illustrative overview for the whole feature engineering process of an artificial GitHub user who we call "Alice" in the following. The upper part of the image describes the real and unmodified user profile. Alice is defined through her bio[17], location and corporation (not shown in fig. 4.4.2, see listing 4.4.1, `userInfo`) but also her repositories (see listing 4.4.1, `repositories`). The lower coloured "Processed User Profile" describes Alice after processing, defined through her specific dynamic-length "bins" (also commonly called "bag-of-words") that are used to organize the data. The bins represent the actual dynamic-length data of the user profile iterables in the `DataFrame`. Users are free to enter *anything* in the text fields: Therefore, finding text features that carry useful information becomes necessary. We use *spaCy* to *tokenize* (sequencing words), remove *stop-words* (common words of a language, e.g. "at" or "the") and further *lemmatize* (finding the base form of a word, e.g. "raining" → "rain") and then store these features into the `bio_tags` bin. Moreover, we use *spaCys named-entity recognition* pipeline [Hon+20] to find outstanding textual entities

[16]Except the continuous "age" field, which is later discretized in section 4.2.3.

[17]From here, whenever referring to "bio", we refer to the actual size-limited profile bio *concatenated* with the recently introduced special repository to enter extended information in a `README`

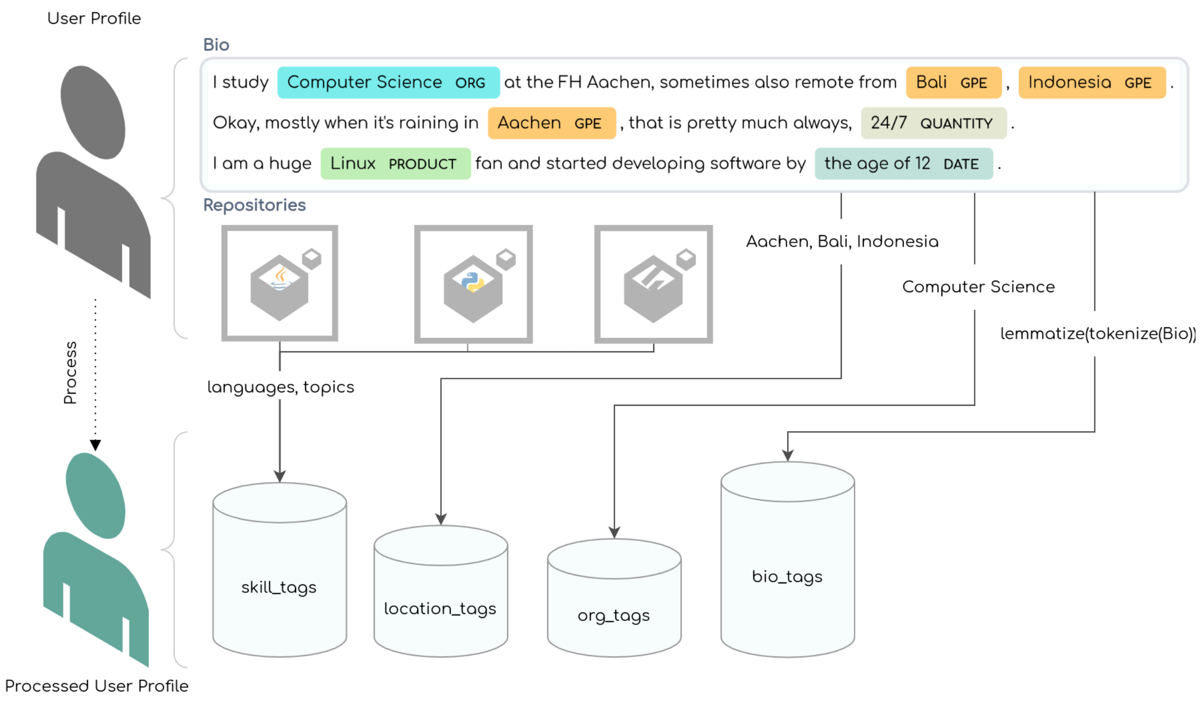

User Profile
Bio
I study Computer Science ORG at the FH Aachen, sometimes also remote from Bali GPE , Indonesia GPE .
Okay, mostly when it's raining in Aachen GPE , that is pretty much always, 24/7 QUANTITY .
I am a huge Linux PRODUCT fan and started developing software by the age of 12 DATE .
Repositories
Aachen, Bali, Indonesia
Computer Science
lemmatize(tokenize(Bio))
languages, topics
Process
skill_tags
location_tags
org_tags
bio_tags
Processed User Profile

Figure 4.4.2: Processing GitHub user profiles by extracting textual features from bio and repositories

and sort them into a suitable bin. For instance, the `location_tags` bin contains GPEs[18] highlighted in orange in the user's bio. Lastly, the repositories also carry useful information; the programming languages (in Alice's case, Java, Python and Flutter) and the user-specified topics (basically a limited amount of tags, such as "framework", "machine-learning" or "recommender-system"). These are both put in the `skill_tags` bin accordingly. Most importantly, the "Processed User Profile" is only an approximation of the real user, as in the process, loss of information is inevitable. After all, this loss is not necessarily "bad", as it is useful for the model to generalize better; correlating features as outlined in section 2.2.3.1 is easier when only a relevant subset of meaningful user features needs to be considered during training.

That being said, the above outlined feature extraction is represented by the following code:

[18] For a full list of the supported named entities, see `https://spacy.io/api/annotation#named-entities`

```python
pipeline = SequentialPipeline([
  # ...
  SequentialPipeline(name='User Profile Preparation', processors=[
    GitHubPreprocessor(skills_per_user=25, repo_languages_per_user=6),
    TextConverter('bio', from_format=ColumnFormatType.MARKDOWN),
    NLPEntityExtractor('bio', {
     'GPE': 'location_tags', 'LOC': 'location_tags',
     'ORG': 'org_tags', 'org_tags', 'NORP': 'org_tags'
    })]),
  ParallelPipeline(name='User Profile Categorization', processors=[
    SequentialPipeline(name='Bio', processors=[
      NLPTokenExtractor('bio', 'bio_tags'),
      MostUsedExtractor('bio_tags', 'bio_tags', usage_threshold=2),
    ]),
    SequentialPipeline(name='Organizations', processors=[
      NLPTokenExtractor('company', 'org_tags'),
      MostUsedExtractor('org_tags', 'org_tags', usage_threshold=2),
    ]),
    SequentialPipeline(name='Location', processors=[
      NLPEntityExtractor('company', {'GPE': 'location_tags'}),
      NLPTokenExtractor('location', 'location_tags'),
      MostUsedExtractor('location_tags', 'location_tags', usage_threshold=2)
    ]),
    SequentialPipeline(name='Process skills', processors=[
      NLPEntityExtractor('repo_descriptions', {'%': 'skill_tags'}),
      MostUsedExtractor('skills', 'skill_tags', usage_threshold=2),
      MostUsedExtractor('repo_languages', 'skill_tags', top=40,
       ↪  usage_threshold=2),
      MostUsedExtractor('skill_tags', 'skill_tags', top=1000,
       ↪  usage_threshold=2)
    ]),
  ]),
])
```

Listing 4.4.4: Pipeline for feature engineering

The first part of the pipeline in listing 4.4.4 is omitted as it is essentially similar to the previous listing 4.2.3. Line 4 shows the `GitHubPreprocessor` that extracts repository-specific information such as *topics* (for the users, we interpret these as *skills*) and the `repo_languages` to columns in the user profile `DataFrame`. Among other tasks, the highly specific preprocessor also approximates a user's activity and concatenates the `bio` with the text from the special repository where users can post extended information about themselves (see note in section 4.4.1 and line 21 in listing 4.4.1). This is followed by a *Markdown*-to-text conversion in line 5 for the `bio` column, which is a preparing step for NLP; it strips the *Markdown* syntax elements that users can freely use for formatting the bio. In the following lines, the `NLPEntityExtractor` and `NLPTokenExtractor` are extensively used to extract textual features of the fields via *spaCy*. Furthermore, the `MostUsedExtractor` strips away rarely used (text) items via `usage_threshold` or limits the items based on occurrence[19] to a specific `top` value. Including words that are only used by a single user make them an indicator feature, which is often not what we want for user profile data itself.

Summarizing, by using modern NLP techniques, Chaos' automatized feature engineering capabilities as firstly introduced in section 2.3.2 are complemented to simplify experimenting with various textual features, thus helping researchers to embed typical user profile attributes.

4.4.3 Results

In this last section, we inspect the results: Given the introductory workflow fig. 4.1.1, we processed the data model (task "Process Data Model"), which is now ready to be fed into the translator and model. In this section, we will go through the last tasks.

The dataset that we use as a foundation for the following experiment results has been created at the January 18, 2021 using the author's GitHub username[20] as a start node. Reciprocal BFS (section 4.4.1.3) has been initialized with 5000 `max_nodes` and a `breadth` of 7 nodes. The *Chaos GQL specification* is likewise constructed as in listing 4.4.1 and the feature engineering steps are by and large given in listing 4.4.4. The above overall parameters lead to a total of 4186 content profile features, described and distributed among the categories as follows:

[19] In the future, more sophisticated processing could make use of the *inverse document frequency* to measure the relevance of a word in the text corpus, see [Agg+16, p. 145]

[20] `https://github.com/kdevo`

Table 4.4.2: Features, their occurrences, origins and overall restrictions

Feature	Occurrences	Profile Origin(s)	Restriction(s)
💬 `bio_tags`	2174	Bio	Used by ≥ 2
💻 `skill_tags`	1000	Repository	Top ≤ 1000, Used by ≥ 2
🏭 `org_tags`	543	Company, Bio	Used by ≥ 2
📍 `location_tags`	469	Location, Bio, Company	Used by ≥ 2

On top, 5000 indicator features are introduced, i.e. one per user (as described in section 2.2.3.1), resulting in a total of 9186 features. Although the dataset is based entirely on publicly available data, for privacy reasons, we will not show any other GitHub usernames.

The graph density is given by ≈ 0.001055, where 1 is a fully connected graph and 0 a graph with no edges at all. This means that we have $\approx$ 99.89%-sparse data at hand, but since the `LFMPredictor`'s underlying *LightFM* was developed with sparsity in mind [Kul15], this should not pose an obstacle.

The reciprocity is $\approx$ 0.7984, which is possibly unsurprising, as this is what the reciprocal BFS (section 4.4.1.3) optimizes for. Nevertheless, it shows that many GitHub users tend to *reciprocate*, further supporting the argument that it should be classified as a RE (suitable for a RRS).

4.4.3.1 Evaluation

In the following, we evaluate different configurations of the `LFMPredictor` (comparable to section 4.2.5.2). One of the advantages of the categorization of features into different "bins" (as illustrated in fig. 4.4.2) is that we are able to easily select only a semantic subset of features via `LFMTranslator`, so that we can test which feature categories are suspected to have the most predictive properties. In addition to a simple list of features as shown before in listing 4.2.6, the `LFMTranslator` supports (initial) normalized weighting of features by passing a dictionary in the format `'<feat>': <weight>`.

Table 4.4.3 shows a variation of different configurations and their maximum result in the test set for *p*@5 (*precision* at 5) and *r*@5 (*recall* at 5). Each icon in the "Feature/Weight" column corresponds to the feature category from the previous table 4.4.2. We also compare two different configurations of "Hyperparameters"; each sub-column stands for a *LightFM* hyperparameter, d denoting the *dimensionality* and η denoting the *learning rate*. The "Hybrid all", "Hybrid orgs + bio" and "Collaborative Filtering only" models were trained with $d = 48$ and $\eta = 0.40$. On the other hand, the "Hybrid all tuned" model was trained with hyperparameters that have been found by using the the novel hyperparameter optimization framework *Optuna* [Aki+19], with a smaller dimensionality of $d = 42$ and a higher learning rate $\eta = 0.418$, among changes in the L_2 penalties (eq. (2.2.8)) that are left out of table 4.4.3.[21]

Table 4.4.3: Different `LFMPredictor` configurations (features and hyperparameters)

	Feature/Weight				Hyperparameters		Test (max.)	
	💬	💻	🏭	📍	d	η	$p@5$	$r@5$
Hybrid all	0.4	0.2	0.2	0.2	48	0.400	0.0414	0.1067
Hybrid all tuned*	0.4	0.2	0.2	0.2	42	0.418	0.0416	0.1075
Hybrid orgs + bio	0.0	0.0	0.5	0.5	48	0.400	0.0396	0.1022
Collaborative Filtering only	0.0	0.0	0.0	0.0	48	0.400	0.0363	0.0899

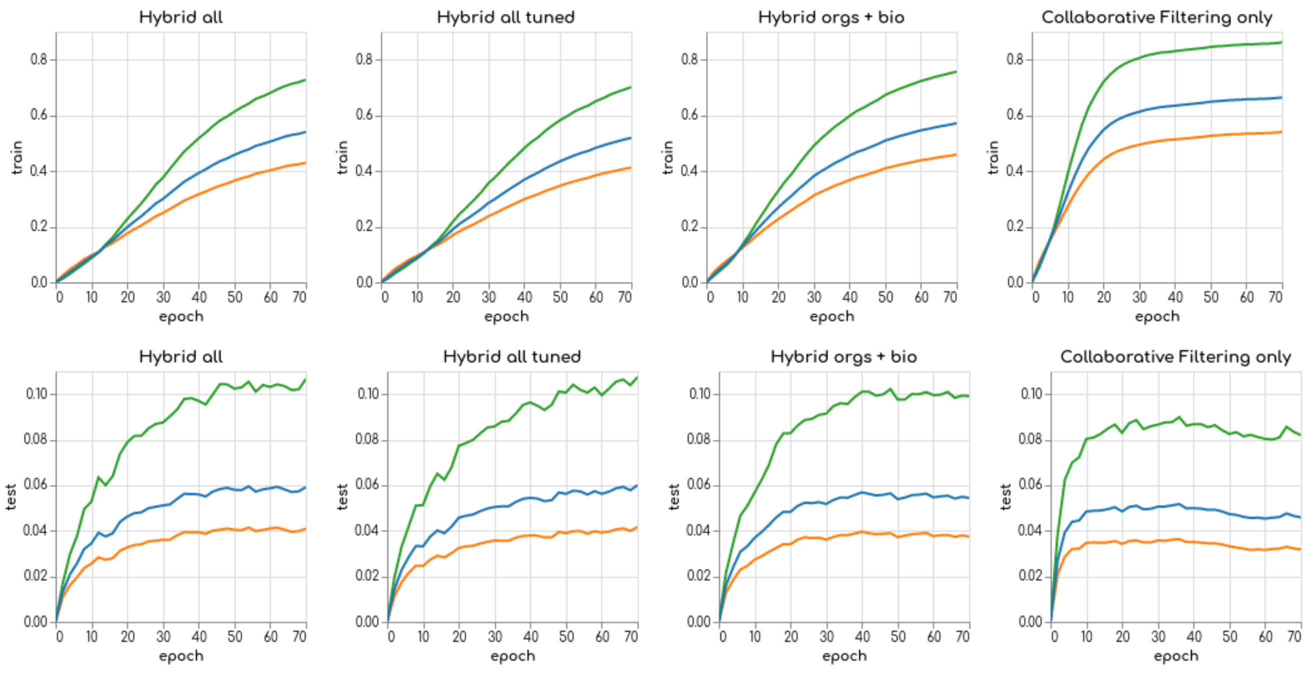


Figure 4.4.3: Evaluation report of the different `LFMPredictor` configurations (see table 4.4.3)

[21] Found with optimization objective on maximizing the F1 score, after 250 trials, $d \leq 64$ and a non-linear search space for other hyperparameters. Details can be found in the class `LFMHyperparameterOptimizer`.

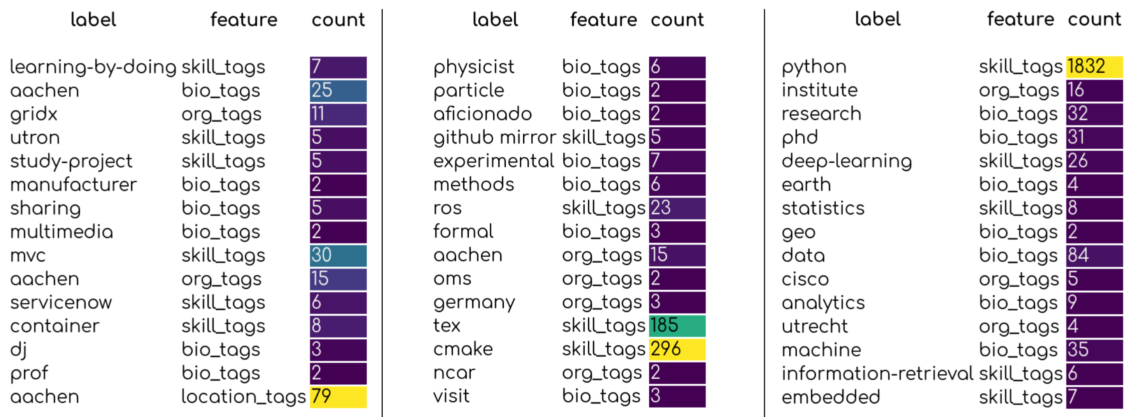
Figure 4.4.3 shows the corresponding chart report, with $p@5$ (eq. (4.2.2)) in orange, $r@5$ (eq. (4.2.3)) in green and $f1@5$ (eq. (4.2.4)) in blue. Additionally to the already named representative configurations, we tested each feature category separately, where only-`org_tags` and only-`bio_tags` gave the best results among them, but interestingly none of them outperformed the combination of all features (i.e. "Hybrid all (tuned)") on their own. The chart report indicates that the hybrid model with optimized hyperparameters "Hybrid all tuned" gives the best result with a precision $p@5 \approx 0.0416$ at epoch 70 (of 70), but is overall very close to the non-optimized version "Hybrid all". This is interesting, because usually models with a higher dimensionality also perform better [Kul15], as they are able to capture increasingly complex latent relationships between the users' and their profile features. Although the tuned version has a smaller $d = 42$ (smaller by 6 dimensions), it is still slightly better, which likely is due to the other hyperparameters which are found to work better *in combination*.

For the above reason, we choose the "Hybrid all tuned" model and wrap it up via `ReciprocalWrapper` to fulfill reciprocity (see section 4.3.2). This model is less time-consuming to train, faster for prediction tasks and uses approximately 12.5% less memory, all with a similar (or slighly better) qualitative performance as the "Hybrid all" model. In conclusion, performing a hyperparameter search/optimization prior to final model deployment after offline training (section 4.1.1) is recommendable: In production, it enables a better performance from a qualitative as well as computational perspective.

4.4.3.2 Explainability and Visualization

An important advantage of the hybrid model is its ability to help out with reciprocal as well as non-reciprocal explanations (see previous section 3.3.1.4). To enable serving actual user explanations, the `LFMPredictor` contains a `similar_features` method that (simplified) calculates the cosine-similarity for a given tag and all other tags [adapted from KC20] [Kul15], sorts them and puts the top k most similar ones inside a result data frame.

The three subfigures of fig. 4.4.4 (a to c) show the results of different tag queries with $k = 15$ for three different feature categories (table 4.4.2): The tag `fh` (6 occurrences in `bio_tags`), the tag `rwth` (13 occurrences in `org_tags`) and lastly the tag `machine-learning` (51 occurrences in `skill_tags`). The `count` column in the result data frames represent the share in the total occurrences of each category (see table 4.4.2) with background colours based on a heatmap similar to the previous interaction graphs. We observe the following:



label	feature	count
learning-by-doing	skill_tags	7
aachen	bio_tags	25
gridx	org_tags	11
utron	skill_tags	5
study-project	skill_tags	5
manufacturer	bio_tags	2
sharing	bio_tags	5
multimedia	bio_tags	2
mvc	skill_tags	30
aachen	org_tags	15
servicenow	skill_tags	6
container	skill_tags	8
dj	bio_tags	3
prof	bio_tags	2
aachen	location_tags	79

(a) fh

label	feature	count
physicist	bio_tags	6
particle	bio_tags	2
aficionado	bio_tags	2
github mirror	skill_tags	5
experimental	bio_tags	7
methods	bio_tags	6
ros	skill_tags	23
formal	bio_tags	3
aachen	org_tags	15
oms	org_tags	2
germany	org_tags	3
tex	skill_tags	185
cmake	skill_tags	296
ncar	org_tags	2
visit	bio_tags	3

(b) rwth

label	feature	count
python	skill_tags	1832
institute	org_tags	16
research	bio_tags	32
phd	bio_tags	31
deep-learning	skill_tags	26
earth	bio_tags	4
statistics	skill_tags	8
geo	bio_tags	2
data	bio_tags	84
cisco	org_tags	5
analytics	bio_tags	9
utrecht	org_tags	4
machine	bio_tags	35
information-retrieval	skill_tags	6
embedded	skill_tags	7

(c) machine-learning

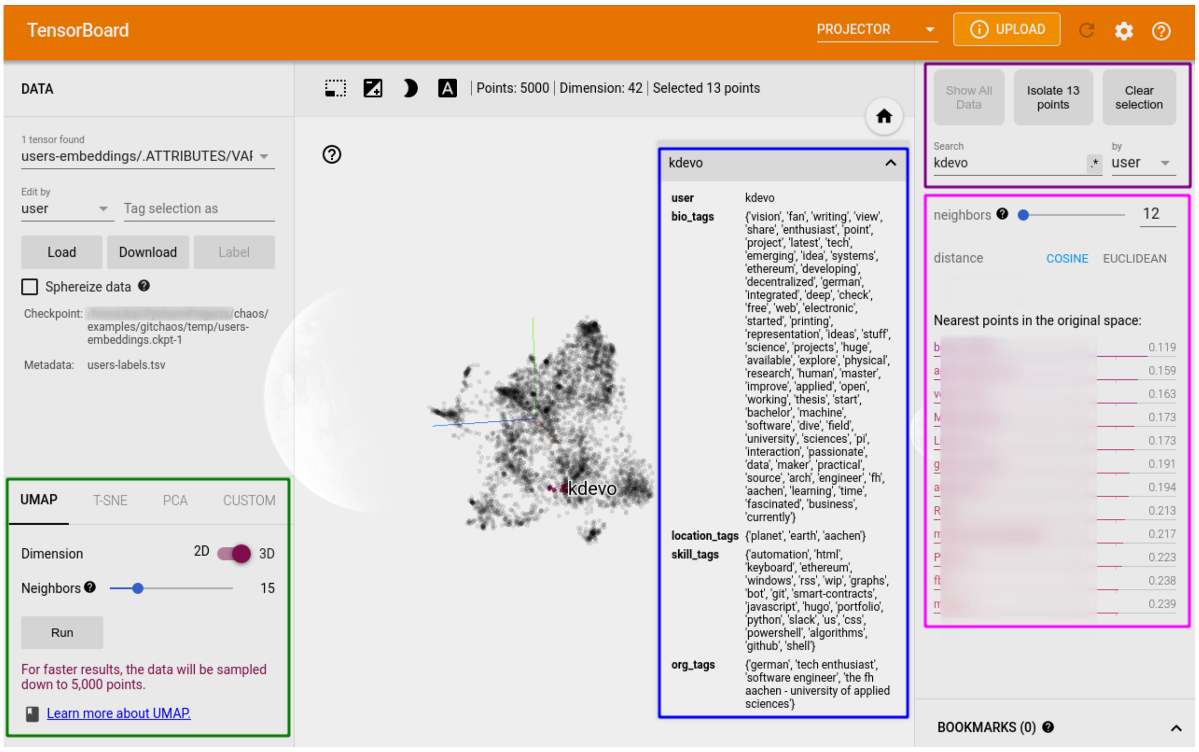
Figure 4.4.4: Different queries for GitHub user tag similarities

- The feature processing through section 4.4.2 adds some (unwanted) ambiguity. For instance, "FH Aachen" was often split into two words (depending on the textual context) and failed to be recognized as an *organization* as a whole. This is partly "as-defined" behavior because of the tokenization (via `NLPTokenExtractor`), but also partly unwanted due to the chosen *spaCy* model[22] for entity recognition in the bio (via `NLPEntityExtractor`) which was trained on an English text corpus. Notably, the quality is nevertheless good enough for the model to learn the semantic relationship between profile attributes in the embedding space which we can query to retrieve interesting similarities, as highlighted by the three examples.
- As we can see by the `count` of each result row and their varying occurrences, the model is *not* simply repeating the user profiles by using a statistical count-based metric: The model learns *user similarities* based on their interactions, i.e. the user profile embeddings are adjusted to best fit to the input, that is, the interaction graph (see section 2.2.3.1 in combination with the alterations in section 4.3.1).

That being said, although humans tend to see "structure in randomness" [Mun+17], observing and comparing the learned latent user profile feature embeddings is an explainability advantage that not only helps the researcher to better *comprehend* and partly evaluate what the model has learned, but also helps the end-user, e.g. by teaming up with the explanation method from section 3.3.1.4. The same stakeholders profit from the user embeddings visualization which we will analyze in the following.

[22]See `https://spacy.io/models/en`, `en_core_web_md` (medium-sized multi-task English model).

Chaos' `LFMPredictor` provides a built-in `visualize` method that dumps the embeddings in a *TensorFlow* compatible format to retrieve insights into the complex (as in high-dimensional) learned embeddings. That is, an embedding tensor which is serialized in a *checkpoint* file and supplemented with a metadata CSV containing user profile tags. After dumping the checkpoint, the *TensorBoard Projector*[23] reads-in the files and offers an interesting way to investigate the embeddings interactively, see screenshot fig. 4.4.5.



Points: 5000 | Dimension: 42 | Selected 13 points

1 tensor found
users-embeddings/.ATTRIBUTES/VAI

Edit by: user — Tag selection as

☐ Sphereize data

Checkpoint: [redacted]chaos/examples/gitchaos/temp/users-embeddings.ckpt-1

Metadata: users-labels.tsv

UMAP | T-SNE | PCA | CUSTOM

Dimension: 2D / 3D

Neighbors: 15

Run

For faster results, the data will be sampled down to 5,000 points.

Learn more about UMAP.

user	kdevo
bio_tags	{'vision', 'fan', 'writing', 'view', 'share', 'enthusiast', 'point', 'project', 'latest', 'tech', 'emerging', 'idea', 'systems', 'ethereum', 'developing', 'decentralized', 'german', 'integrated', 'deep', 'check', 'free', 'web', 'electronic', 'started', 'printing', 'representation', 'ideas', 'stuff', 'science', 'projects', 'huge', 'available', 'explore', 'physical', 'research', 'human', 'master', 'improve', 'applied', 'open', 'working', 'thesis', 'start', 'bachelor', 'machine', 'software', 'dive', 'field', 'university', 'sciences', 'pi', 'interaction', 'passionate', 'data', 'maker', 'practical', 'source', 'arch', 'engineer', 'fh', 'aachen', 'learning', 'time', 'fascinated', 'business', 'currently'}
location_tags	{'planet', 'earth', 'aachen'}
skill_tags	{'automation', 'html', 'keyboard', 'ethereum', 'windows', 'rss', 'wip', 'graphs', 'bot', 'git', 'smart-contracts', 'javascript', 'hugo', 'portfolio', 'python', 'slack', 'us', 'css', 'powershell', 'algorithms', 'github', 'shell'}
org_tags	{'german', 'tech enthusiast', 'software engineer', 'the fh aachen - university of applied sciences'}

Search: kdevo by: user

neighbors: 12

distance: COSINE | EUCLIDEAN

Nearest points in the original space:

b[redacted]	0.119
a[redacted]	0.159
v[redacted]	0.163
M[redacted]	0.173
L[redacted]	0.173
g[redacted]	0.191
a[redacted]	0.194
R[redacted]	0.213
m[redacted]	0.217
P[redacted]	0.223
fl[redacted]	0.238
m[redacted]	0.239

Figure 4.4.5: Using TensorBoard's Projector to visualize and interpret latent user embeddings

The coloured markings in fig. 4.4.5 are explained as follows (from left to right):

1. **Green**: Select a dimensionality reduction algorithm that *projects* the embedding space $\mathbb{R}^d \to \mathbb{R}^3$ (or $\to \mathbb{R}^2$). 3D and 2D representations can be seen and interpreted by humans (opposing to $d \geq 4$, such as $d = 42$ as in the model at hand). Explaining the different algorithms (namely UMAP, tSNE and the more simple PCA) is out of this thesis' scope and they all have different pros and cons, depending on the goal that needs to be achieved, e.g. finding possible clusters with UMAP as shown in the screenshot.

[23] Also available as a web app, see `https://projector.tensorflow.org/`

2. **Blue**: This is the author's profile data; it shows a broad spectrum of different tags and opens up when searching or clicking on a point (user).
3. **Purple**: The upper part of this panel can be utilized to search for a user by name or by tags (regular expressions are supported) and the lower part makes it possible to calculate *k* neighbours either based on *euclidean* or *cosine* similarity (see the early section 2.2).

Summarizing, the visualization provided by the *TensorBoard Projector* in fact serves the purpose of a suitable explanation tool for researchers, fulfilling the mentioned guidelines in section 2.3.4 by providing a domain-specific, familiar (as in "well-known") and objectively transparent interface.

4.4.3.3 Summary

All in all, we demonstrated the versatility of the framework Chaos by using it in the specific scenario of finding code collaborators and followers on GitHub. We first analyzed the platform's user interactions and profile data (section 4.4.1.1) and classified the platform as a SNS that is very likely to profit from reciprocal recommendations. This is followed by a unique data generation approach that uses the Chaos GQL specification (section 4.4.1.2) to incrementally build a social network by fetching the selected interaction and profile data from the prior analysis via *Reciprocal BFS* from a public API endpoint (section 4.4.1.3). Next, the resulting user profiles were prepared for model input by using the built-in feature engineering capabilities of Chaos, including an approach to automatically categorize relevant text entities (section 4.4.2).

Finally, we evaluated the different profile feature categories (bio, skill, organization, location) and their predictive power by performing cross-validation (section 4.4.3.1) and interpreted the resulting embeddings (section 4.4.3.2). Moreover, we have shown how the framework to gain deep insights into the social universe of the platform's users.

While one might concentrate on the repositories and versioning aspect only at first sight, open-source and free software relies on contributions made by humans, thus applying Chaos on a popular code collaboration platform is an important step to improve the social dimension of software engineering.

5 Conclusions

There are only patterns, patterns on top of patterns,
patterns that affect other patterns. Patterns hidden by
patterns. Patterns within patterns. If you watch close,
history does nothing but repeat itself.
What we call chaos is just patterns we haven't recognized.
What we call random is just patterns we can't decipher.
What we can't understand we call nonsense.

Chuck Palahniuk • Survivor

In the previous section, we introduced the framework Chaos by first presenting its general building blocks (section 4.2), its algorithms (section 4.3) and then applying them in a real-world scenario (section 4.4), which can be reproduced by the reader.

In this final chapter of the thesis, we first examine the contributions, opportunities and limitations of this work (section 5.1) and wrap them up by referring to the objectives from the introduction (section 5.2). At the very end, we provide an outlook (section 5.3) for the future of Chaos and address research possibilities for RRSs in general.

5.1 Contributions

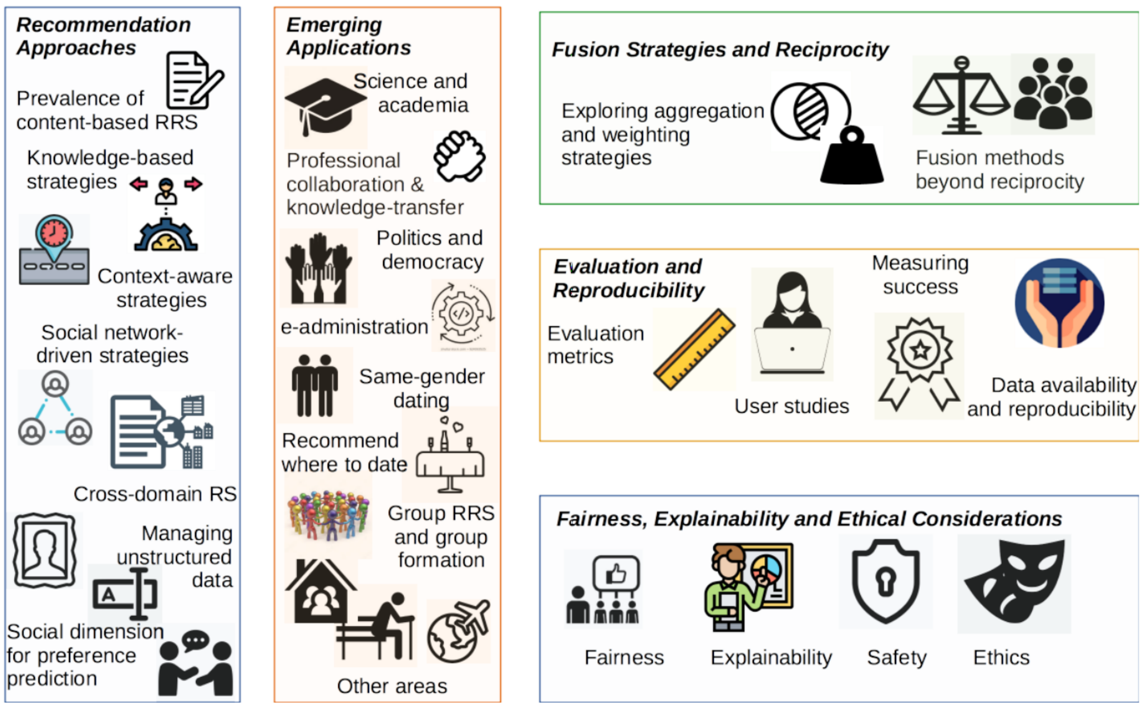

Recommendation Approaches
Prevalence of content-based RRS
Knowledge-based strategies
Context-aware strategies
Social network-driven strategies
Cross-domain RS
Managing unstructured data
Social dimension for preference prediction
Emerging Applications
Science and academia
Professional collaboration & knowledge-transfer
Politics and democracy
e-administration
Same-gender dating
Recommend where to date
Group RRS and group formation
Other areas
Fusion Strategies and Reciprocity
Exploring aggregation and weighting strategies
Fusion methods beyond reciprocity
Evaluation and Reproducibility
Evaluation metrics
User studies
Measuring success
Data availability and reproducibility
Fairness, Explainability and Ethical Considerations
Fairness
Explainability
Safety
Ethics

Figure 5.1.1: Perspective and challenges of RRSs

Source: [Pal+20]

The contributions of this work are best summarized by relating to the most recent paper [Pal+20] about the state-of-art for RRSs. Figure 5.1.1 shows the various research opportunities of RRSs divided into 5 perspectives. In the following, we go through each area section-wise and describe what Chaos has contributed and where it still needs to be improved.

5.1.1 Recommendation Approaches

5.1.1.1 Approach to the research question

To the best of the author's knowledge, Chaos is the first of its kind RRS framework which incorporates an existing *hybrid* RS latent factor model library. Thus, answering the **research question** from the end of section 1.2:

Can a user-to-item RS be utilized to generate reciprocal user-to-user recommendations?

Utilizing and integrating an existing user-to-item RS is indeed possible, as we have demonstrated in section 4.3.1 in accordance with the reciprocity-fulfilling (eq. (3.1.2)) algorithm in section 4.3.2. This finding might be obvious, considering that RRSs are a subarea (or children) of RSs, although the relationship between the two is sometimes referred to as "*fundamentally* different" in literature, which we consider as inaccurate: The base *fundament* of algorithms is *inherently* similar and *not* different, although RRSs are inherently more complex to handle, for example due to their wide-ranging differences given by their always prevalent social dimension as highlighted in section 3.2 and section 3.3. Due to the rareness of our presented approach (see section 4.1.2.3), it might be more appropriate to change the wording here to a less drastic version that encourages RS researchers to enter the area of RRSs for *transfer learning* to take effect.

In comparison, with LFRR [NP19b], another RRS latent factor model has been introduced in September 2019 (see section 4.3.1 for details) which integrates an existing well-known CF MF model (section 2.2.2.1). However, it relies on *explicit* feedback and requires to partition the interaction matrix based on the amount of classes (e.g. male and female) with each being trained in isolation. By contrast, Chaos is specialized in *implicit* feedback and has the advantage of simplicity by only needing one single RS (based on *LightFM*) to be trained that is used to derive recommendations for n user classes (section 4.3.1). Moreover, it is able to embed profile data effortlessly (as demonstrated in section 4.4.2), where in [NP19b] the latent factors are only used for CF, without the ability to enrich the model by embedding comprehensive user profile data to mitigate the cold-start problem (section 3.3.1.1). Nonetheless, we clearly note that our approach from section 4.3.1 still needs to be evaluated more extensively in further studies to validate the effectiveness (see outlook section 5.3.2 and section 5.3.3).

5.1.1.2 Other approaches contributed to

Referring to fig. 5.1.1: With Chaos supporting a hybrid model (section 4.3.1) *and* a pure CF-based approach (section 4.3.3) is a good opportunity to tackle the "prevalence of content-based RRS". Furthermore, Chaos natively uses a social network as data model (section 4.2.1), which eases researching "social-network driven strategies", thus also supporting the well-known RCF (section 4.3.3). Moreover, we hope that the novel method to build a prediction graph leads to new post-optimization possibilities (see outlook section 5.3.1)

With the proposed architecture and feature engineering techniques (section 2.3.2 and section 4.4.2) we also contributed to the process of "managing unstructured data" and enable new "cross-domain" solutions by combining users from multiple industries and by sourcing data from a *public* third-party API as demonstrated with the GitHub GraphQL API (section 4.4.1.2).

Regarding the "social dimension for preference prediction", Chaos enabled the use of common algorithms from social network analysis by the presented data model (section 4.2.1), but we see room for supporting advanced link prediction methods [Pal+20, p. 44] [see also Agg+16, pp. 326 ff.] and in supporting group formations, especially in learning/knowledge-transfer scenarios [Pal+20, p. 24]. Additionally on the limitations side, we did not consider "context-aware strategies" or "knowledge-based strategies", neither within the framework nor within this thesis, see upcoming outlook section 5.3.1.1.

5.1.2 Emerging Applications

First and foremost, Chaos does not conceptually exclude users from the recommendation process and therewith enables a broad spectrum of *applications* to be used in. As a non-binary single-class RRS (see section 3.1.2.2), Chaos is *inclusive by default* (see also analyzed requirements in section 3.3). In section 4.2.4.2, we presented a reciprocal candidate generation approach to still consider both users' must-have criteria. This is useful to support explicit user wishes (e.g. filtering for project partners that have experience in a specific programming language) or in a dating RE, supporting multiple sexual orientations. Chaos is designed in a generic way to be able to also handle two-class scenarios, whereas in contrast two-class RRSs typically fail to be used for single-class or ≥ 3 classes (see fig. 4.3.1 for example). Still, explicitly supporting *two-class* RRSs *in addition* to the framework's above-described approach might bring advantages. For instance, if the two classes never share a common set of attributes, it might be more fitting (e.g. from an efficiency viewpoint) to specialize to a two-class RRS, which is why we leave the door open here for contributions of any kind (see outlook section 5.1). In addition to that, as Chaos' data model and algorithms are specialized in implicit feedback, we support a highly useful feedback type that is typically widely available (section 2.2.3.2). Still, in some applications, it might be wanted to to consider explicit feedback (either additionally or separately) in a later release, e.g. by adapting the interaction model to also support negative strengths (see section 4.2.1.1).

Referring to "professional collaboration and knowledge transfer" in fig. 5.1.1, with the real-world GitHub example (section 4.4), we have demonstrated that Chaos is suitable to be used for reciprocal follower or code-collaborator recommendations. As we have analyzed in section 4.4.1, the platform can in fact be considered as a SNS with complex interaction types that are subject be further examined. For instance, we introduced the term of *transitive interactions* in eq. (4.4.1) which we consider relevant for REs, but their effect and effectiveness requires further studying.

5.1.3 Fusion Strategies and Reciprocity

The fusion of single-sided preferences is the most important aspect of a RRS that differentiates them from a classic RRS [Pal+20] (see section 3.1.2.1). Therefore, in section 4.3.2 we presented a generic reciprocity enabler that wraps a user-to-user RS to fulfil the reciprocity condition (see section 3.1.1 for the definition) by aggregating the user preferences according to a specified strategy (fig. 4.3.2). Supporting different strategies has been motivated by [NP19a] and [Pal+20]: With Chaos, we expect to see upcoming experiments and comparisons between the strategies which are enabled by the evaluation capabilities (fig. 4.4.3) in reproducible fashion, which brings us to the next section.

5.1.4 Evaluation and Reproducibility

With our approach in making chapter 4 partly interactive with help of *JupyterLab* (see notebook 4.2.1 and notebook 4.4.1), we aimed to include the reader in a practice-oriented way and therewith promote reproducibility. Furthermore, with the *GitHub* scenario (section 4.4), we have decided for public data and shown how to actually *generate* data from a publicly available third-party API (section 4.4.1). We aim for the introduced *reciprocal BFS* algorithm (section 4.4.1.3) and the presented *GQL specification* (section 4.4.1.2) to lead to a literal network effect for researchers to use publicly available data for evaluation instead of proprietary datasets that no external party can access.

Regarding the cross-algorithm evaluation of RRSs, further research and improvements are required, see outlook section 5.3.3. Moreover, the other aspects of this field in fig. 5.1.1 were out of this thesis' scope due to the framework focus, i.e. the primary objective of engineering a RRS framework is *not* to perform a user study or to measure success for multiple stakeholders [Pal+20], but we especially acknowledge that evaluation metrics should be more "user-centered" and "reciprocity-aware" as posed in [Pal+20, p. 44].

5.1.5 Fairness, Explainability and Ethical Considerations

Regarding *fairness* and *ethical considerations*, we have shown an experimental approach in section 4.2.3 and especially section 4.2.3.1 to incorporate the *popularity* of a user into the model to mitigate the *popularity bias* which we introduced in section 2.3.3 (RS) and section 3.3.1.3 (RRS).

Towards *explainability* (and the emerging field of explainable AI): We successfully analyzed and presented the current state-of-the-art for this particularly important user-satisfaction enhancing method in section 3.3.1.4. Thereafter in section 4.4.3, we have shown a way to calculate the similarity of user profile attributes and outlined how this can be used in conjunction with the method described in section 3.3.1.4 to generate *domain-specific*, *transparent* and *effective* explanations (see section 2.3.4 for the exact guidelines). On top of that, with the automatized export to the featureful *TensorBoard Projector*, we aimed to provide an explanation method targeted at *researchers* that fulfills the above-noted guidelines. Remarkably, the first step in providing high-quality explanations for *end-users* lays in offering a suitable explanation interface for the humans who develop the explanation functionality itself. If *we* are not able to comprehend the way the system recommends, neither will the end-users (at least not in a truthful transparent way, see *objective transparency* in section 2.3.4). For limitations regarding this aspect, see the outlook section 5.3.2.

5.2 Objectives accomplished in Code

Back to the beginning: We faced the problem of limited reproducibility regarding RSs and RRSs in section 1.2. For RRSs, we see that the history of their parent (RSs) is *inherited* and *repeated*: As found and formulated by Ekstrand et al. for RSs, "algorithmic enhancements are typically published as mathematical formulae rather than working code, [...] important optimizations such as preprocessing normalization steps may be omitted, leading to new algorithms being incorrectly evaluated [...]" [Eks+11] – the same applies to our analyzed state-of-the-art of RRSs.

Therefore, from the experience of this work, with respect to the analyzed literature, we derive and propose the following three guidelines for future research:

1. **Pseudo-code is not code**: Pseudo-code might be a suitable tool to create drafts and/or to show the concept of an algorithm, but it is neither standardized nor executable.

2. **Provide code over pseudo-code**: When providing pseudo-code, it should be a supplement and not the only resource. The (re-)implementation based on formulae or pseudo-code is often error-prone and time-consuming as it always allows room for interpretation [see also Eks+11]. In the field of data science and ML, Python is the de facto standard [DH20] and therefore an appropriate domain-specific choice to accompany a scientific RRS paper with hosted code.[1]

3. **Aim for open data**: The task of providing SNS open data requires interdisciplinary work (legal, diplomatic, security/privacy-related) and is indeed non-trivial. However, if possible, RRS algorithms should be tested against publicly available data [Pal+20, p. 45], just as in the RS landscape (for instance *MovieLens* dataset), which aids the comparability of different algorithms in combination with a unified evaluation process (outlook section 5.3.3).

Further meta-science guidance can be obtained from the *manifesto for reproducible science* [Mun+17] which formulates generic concepts that are in fact also highly applicable to RSs/RRSs.

Ultimately, with direct reference to the objectives of this thesis: With Chaos, we engineered a framework for RRSs that figuratively speaking aims to build a solid bridge between the research and development departments of RRSs, with the ultimate goal that in the future,

[1] A suitable good example how the harmony between an executable code-base and scientific RS model definitions can be accomplished is given by the *LightFM* project [KC20] and its paper [Kul15].

improvements are *not* developed in a “decentralized fashion” [Eks+11] anymore, thus greatly accelerating the research. Comparing RSs and RRSs in the textual work (the thesis at hand) as well as in the implemented work (the framework) helps with finding out which *good* aspects can be inherited from the parent of RSs. We also aimed to lower the entry barrier for new participants in the RRS domain by outlining the most important and recent challenges as well as algorithms.

5.3 Outlook

5.3.1 Recommendation Optimizations

We see opportunities in optimizing the recommendation core of Chaos from an effectiveness (making good recommendations), as well as from an efficiency (making recommendations in reasonable time) viewpoint.

5.3.1.1 Effectiveness

Towards increasing the effectiveness of recommendations:

- **Prediction graph exploration and exploitation**: Building a *prediction graph* as described in section 4.2.4.3 helps with gaining a full view on the recommendations and users who are predicted to be reciprocally compatible (symmetric edges between two nodes). Therefore, algorithms might be performed on the prediction graph itself to post-optimize recommendations. For instance, a cycle-based algorithm can be used to find (indirectly) reciprocally compatible pairs or algorithms from *trust networks* (see [Agg+16]) could increase effectiveness. Interestingly, RCF (section 4.3.3) could also be executed on the non-reciprocal prediction graph that is e.g. produced with the bare `LFMPredictor` (section 4.3.1), perhaps improving the recommendation quality.
- **Tackling reciprocal cold-start**: Enhance RCF to support cold-start; for instance, by finding a user with similar attributes as a *reference user* before executing the algorithm [Agg+16] (i.e. by combining it with a pure CBF-based approach as a pre-strategy [Pal+20]). This method of finding a similar user by content data could also be integrated into the `ReciprocalWrapper` (section 4.3.2).

All of the opportunities have in common that they *should* consider the *contextual informa-*

tion of the user [Pal+20, p. 43]. Hence, in the future, strategies that combine different recommendation approaches according to user context (recent interactions, time, location, activity, reciprocity or other graph-based metrics, but also popularity) might have a significant impact on the effectiveness. For this reason, we also see potential in closely examining *multi-armed bandit* algorithms (see [see Agg+16, pp. 418 ff.]) in the reciprocal context.

5.3.1.2 Efficiency

On the computational efficiency side:

- **Parallelization**: RCF (see section 4.3.3) needs to be parallelized to make it scalable for bigger datasets, as it is highly inefficient when calculating recommendations for a user given a large amount of candidates and neighbours through its in-memory nature (see also [NP19b] which confirms these findings).
- **Graph-based**: RCF as well as the data model in general (section 4.2.1) would profit from a high-performance graph backend; for this, *Grapresso* (see section 4.1.2) might require an upgrade to support lower-level backends, e.g. based on *Cython*. Grapresso also supports copying to other backends; as such, it would not loose access to the broad algorithmic spectrum of *NetworkX* (see section 4.2.3).
- **Reciprocal Factorization Machine**: The *LightFM* model on which the `LFMPredictor` relies (section 4.3.1) is a specialization of a FM, as outlined by the end of section 2.2.3.1. With FMs' sophisticated interaction model [Ren10], it might be worth to take a look into how FMs can represent *user-to-user* interactions directly without using *LightFM* as an intermediary that needs to be adapted first for this purpose (section 4.3.1). Moreover, they might enable the integration of contextual information such as time or a few last (possibly reciprocal) interactions with other users.

5.3.2 Interpretability and Explainability

The presented hybrid model (`LFMPredictor`) which integrates *LightFM* needs to be more extensively studied regarding its interpretability and explanation capabilities. With direct reference to section 4.3.1, the following research questions are proposed:

1. How does the semantic of embeddings from eq. (2.2.11) and eq. (2.2.12) (resembling u and v in section 4.3.1) differ when calculating the profile attribute or user *similarities*?

2. Is it sometimes (e.g. in cold-start scenarios) preferable to calculate the *cosine* similarity of u to other users instead of calculating the *dot product*?

5.3.3 Unified Evaluation

As noted by the end of section 4.2.5.2, a first prototype of the `GraphEvaluator` has been developed which is able to perform *reciprocal cross-validation* (see section 4.2.5) on any implementation of the `Predictor` interface. However, the functionality still needs to be validated and possibly refined due to the following two uncertainties that need to be eliminated:

1. **Mixed metrics**: The introduced data model (section 4.2.1.1) permits unidirectional as well as bidirectional interactions (section 4.2.1.1). Therefore, if we solely follow the success definition from eq. (4.2.5), all training data where u only interacted with v (and not the other way around) will be *excluded* from the test set. However, it might still be wished to evaluate if the `Predictor` is able to estimate the one-sided preference of u towards v *additionally*, hence there is a need for a *mixed reciprocal/non-reciprocal metric* here, as predicting one side is still better than predicting neither.

2. **Successful train vs. test**: If the `Predictor` correctly predicts a successful interaction from the *training* set, should it be in- or excluded from the score calculation against the known positive interactions from the *test* set? While libraries such as *LightFM* leave both options open [KC20], we still see the need for discussion here and also the requirement of finding a proper naming for the method.

Currently, research itself lacks of a stable and standardized evaluation method for RRSs, especially in the context of *cross-validation* and *implicit feedback*. Many papers re-define the evaluation for their own purposes, cf. for instance [NP19b] [Xia+15] [Ake+11] [Piz+10b]. Even though some of them might refer to one and the same, they are described all over again in various notations – the RRS landscape will profit from a *unified* evaluation methodology that is able to work across various algorithms and that can be used as a reference in research papers. Therefore, as a first move, we propose a peer-(code)review of the `GraphEvaluator` after Chaos has been deployed to the public, which leads us to the final section of this thesis.

5.3.4 Collaborative Chaos

Lastly and most importantly, Chaos will be fully open-source and the publication is scheduled for the beginning of spring 2021: In the long run, the framework can only flourish when it is published and other contributors join.

Thinking ahead of time, Chaos **H**elps **A**ctivating **O**nline **S**ocieties – maybe it can be used by itself to form a reciprocally compatible society of contributors; just as chaotic as it might seem at first sight.

Bibliography

Books

[Agg+16] Charu C Aggarwal et al. *Recommender Systems. The Textbook*. Vol. 1. Springer, 2016.

[Fal19] Kim Falk. *Practical Recommender Systems*. 2019, p. 432. ISBN: 9781617292705.

Articles and Conference Papers

[Abd+19] Himan Abdollahpouri et al. "The unfairness of popularity bias in recommendation". In: *arXiv preprint arXiv:1907.13286* (2019).

[Ake+11] Joshua Akehurst et al. "CCR — a content-collaborative reciprocal recommender for online dating". In: *Twenty-Second International Joint Conference on Artificial Intelligence*. 2011.

[Aki+19] Takuya Akiba et al. "Optuna: A Next-generation Hyperparameter Optimization Framework". In: *CoRR* abs/1907.10902 (2019). arXiv: `1907.10902`. URL: `http://arxiv.org/abs/1907.10902`.

[App+17] Orestes Appel et al. "Cross-ratio uninorms as an effective aggregation mechanism in sentiment analysis". In: *Knowledge-Based Systems* 124 (2017), pp. 16–22.

[Bee+16] Joeran Beel et al. "Towards reproducibility in recommender-systems research". In: *User modeling and user-adapted interaction* 26.1 (2016), pp. 69–101. (Visited on 08/01/2020).

[Bob+11] Jesus Bobadilla et al. “A collaborative filtering approach to mitigate the new user cold start problem”. In: *Knowledge Based Systems - KBS* 26 (Jan. 2011), p. 14. DOI: 10.1016/j.knosys.2011.07.021.

[Bui+13] Lars Buitinck et al. “API design for machine learning software: experiences from the scikit-learn project”. In: *arXiv preprint arXiv:1309.0238* (2013).

[BZ83] Daniel Brand and Pitro Zafiropulo. “On communicating finite-state machines”. In: *Journal of the ACM (JACM)* 30.2 (1983), pp. 323–342.

[CAS16] Paul Covington, Jay Adams, and Emre Sargin. “Deep neural networks for Youtube Recommendations”. In: *Proceedings of the 10th ACM conference on recommender systems*. 2016, pp. 191–198. DOI: 10.1145/2959100.2959190. URL: https://storage.googleapis.com/pub-tools-public-publication-data/pdf/45530.pdf.

[Dac+19] Maurizio Ferrari Dacrema et al. “A Troubling Analysis of Reproducibility and Progress in Recommender Systems Research”. In: (2019). arXiv: 1911.07698 [cs.IR]. (Visited on 08/01/2020).

[DH20] Kai Dinghofer and Frank Hartung. “Analysis of Criteria for the Selection of Machine Learning Frameworks”. In: *2020 International Conference on Computing, Networking and Communications (ICNC)*. 2020, pp. 373–377. DOI: 10.1109/ICNC47757.2020.9049650.

[Eks+11] Michael D Ekstrand et al. “Rethinking the recommender research ecosystem: reproducibility, openness, and LensKit”. In: *Proceedings of the fifth ACM conference on Recommender systems*. 2011, pp. 133–140.

[GF17] Bryce Goodman and Seth Flaxman. “European Union regulations on algorithmic decision-making and a “right to explanation””. In: *AI magazine* 38.3 (2017), pp. 50–57.

[GJG14] Fatih Gedikli, Dietmar Jannach, and Mouzhi Ge. “How should I explain? A comparison of different explanation types for recommender systems”. In: *International Journal of Human-Computer Studies* 72.4 (2014), pp. 367–382. DOI: 10.1016/j.ijhcs.2013.12.007.

[HDB20] Thomas Hellström, Virginia Dignum, and Suna Bensch. “Bias in Machine Learning – What is it Good for?” In: (2020). arXiv: 2004.00686 [cs.AI].

[HKV08] Yifan Hu, Yehuda Koren, and Chris Volinsky. “Collaborative filtering for implicit feedback datasets”. In: *2008 Eighth IEEE International Conference on Data Mining*. IEEE. 2008, pp. 263–272.

[KBV09] Yehuda Koren, Robert Bell, and Chris Volinsky. “Matrix factorization techniques for recommender systems”. In: *Computer* 42.8 (2009), pp. 30–37.

[Kle+18] Akiva Kleinerman et al. “Optimally balancing receiver and recommended users’ importance in reciprocal recommender systems”. In: *Proceedings of the 12th ACM Conference on Recommender Systems*. 2018, pp. 131–139.

[KRK18] Akiva Kleinerman, Ariel Rosenfeld, and Sarit Kraus. “Providing explanations for recommendations in reciprocal environments”. In: *Proceedings of the 12th ACM Conference on Recommender Systems*. 2018, pp. 22–30. DOI: 10.1145/3240323.

[Kul15] Maciej Kula. “Metadata Embeddings for User and Item Cold-start Recommendations”. In: *Proceedings of the 2nd Workshop on New Trends on Content-Based Recommender Systems co-located with 9th ACM Conference on Recommender Systems (RecSys 2015), Vienna, Austria, September 16-20, 2015.* Ed. by Toine Bogers and Marijn Koolen. Vol. 1448. CEUR Workshop Proceedings. CEUR-WS.org, 2015, pp. 14–21. URL: http://ceur-ws.org/Vol-1448/paper4.pdf.

[Lew13] Kevin Lewis. “The limits of racial prejudice”. In: *Proceedings of the National Academy of Sciences* 110.47 (2013), pp. 18814–18819.

[Mit80] Tom M Mitchell. “The need for biases in learning generalizations”. In: (1980).

[Mun+17] Marcus R Munafò et al. “A manifesto for reproducible science”. In: *Nature human behaviour* 1.1 (2017), pp. 1–9.

[NP19a] James Neve and Ivan Palomares. “Aggregation Strategies in User-to-User Reciprocal Recommender Systems”. In: *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)*. IEEE. 2019, pp. 4031–4036.

[NP19b] James Neve and Ivan Palomares. “Latent factor models and aggregation operators for collaborative filtering in reciprocal recommender systems”. In: *Proceedings of the 13th ACM Conference on Recommender Systems*. 2019, pp. 219–227.

[Pal+20] Ivan Palomares et al. “Reciprocal Recommender Systems: Analysis of State-of-Art Literature, Challenges and Opportunities on Social Recommendation”. In: (2020). arXiv: 2007.16120 [cs.SI]. URL: https://arxiv.org/pdf/2007.16120v2.pdf.

[Piz+10a] Luiz Pizzato et al. “Reciprocal Recommenders”. In: (2010). URL: http://ceur-ws.org/Vol-606/paper5.pdf.

[Piz+10b] Luiz Pizzato et al. “RECON: a reciprocal recommender for online dating”. In: *Proceedings of the fourth ACM conference on Recommender systems*. 2010, pp. 207–214.

[PRG16] Fereshteh-Azadi Parand, Hossein Rahimi, and Mohsen Gorzin. “Combining fuzzy logic and eigenvector centrality measure in social network analysis”. In: *Physica A: Statistical Mechanics and its Applications* 459 (2016), pp. 24–31.

[Ren10] S. Rendle. “Factorization Machines”. In: *2010 IEEE International Conference on Data Mining*. 2010, pp. 995–1000. DOI: 10.1109/ICDM.2010.127.

[SFR+06] K Shyong, Dan Frankowski, John Riedl, et al. “Do you trust your recommendations? An exploration of security and privacy issues in recommender systems”. In: *International Conference on Emerging Trends in Information and Communication Security*. Springer. 2006, pp. 14–29.

URL:
http://files.grouplens.org/papers/lam-etrics2006-security.pdf.

[Ste10] Harald Steck.
"Training and testing of recommender systems on data missing not at random".
In: *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2010, pp. 713–722.

[Van+18] Jacob VanderPlas et al. "Altair: Interactive statistical visualizations for python".
In: *Journal of open source software* 3.32 (2018), p. 1057.
URL: https://altair-viz.github.io/.

[Wic+14] Hadley Wickham et al. "Tidy data".
In: *Journal of Statistical Software* 59.10 (2014), pp. 1–23.
URL: http://vita.had.co.nz/papers/tidy-data.pdf.

[Xia+15] Peng Xia et al. "Reciprocal recommendation system for online dating".
In: *2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*. IEEE. 2015, pp. 234–241.
URL: https://arxiv.org/abs/1501.06247v2.

[Xia+16] Peng Xia et al.
"Design of reciprocal recommendation systems for online dating".
In: *Social Network Analysis and Mining* 6.1 (2016), p. 32.

Software and Documentation

[Hon+20] Matthew Honnibal et al.
spaCy: Industrial-strength Natural Language Processing in Python. 2020.
DOI: 10.5281/zenodo.1212303. URL: https://spacy.io/api.

[HSS08] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart.
Exploring Network Structure, Dynamics, and Function using NetworkX.
Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman.
Pasadena, CA USA, 2008. URL:
https://networkx.org/documentation/stable/reference/algorithms/.

[KC20] Maciej Kula and Contributors. *LightFM 1.15.* 2020.
URL: https://making.lyst.com/lightfm/docs/lightfm.html (visited on 12/11/2020).

[Tea20] GitHub Team. *GraphQL Queries - GitHub Docs.* 2020.
URL: https://docs.github.com/en/free-pro-team@latest/graphql/reference/queries (visited on 12/01/2020).

Websites and Blog-Articles

[Goo18] Developers at Google. *Recommendation Systems Google Course.* Nov. 30, 2018.
URL: https://developers.google.com/machine-learning/recommendation/collaborative/matrix.

[Goo20] Developers at Google. *Machine Learning Course.* Feb. 10, 2020.
URL: https://developers.google.com/machine-learning/crash-course/ml-intro (visited on 08/01/2020).

[Kul18] Maciej Kula. *Explicit vs. Implicit Recommendations.* Jan. 21, 2018.
URL: https://github.com/maciejkula/explicit-vs-implicit (visited on 08/01/2020).

[Lun20] Eric Lundquist.
Factorization Machines for Item Recommendation with Implicit Feedback Data. June 28, 2020. URL: https://towardsdatascience.com/-5655a7c749db.

[Mcm19] Thomas Mcmullan. *Are the algorithms that power dating apps racially biased?* Feb. 17, 2019.
URL: https://www.wired.co.uk/article/racial-bias-dating-apps.

[Pre16] Gil Press. *Cleaning Big Data: Most Time-Consuming, Least Enjoyable Data Science Task, Survey Says.* Mar. 23, 2016.
URL: https://www.forbes.com/sites/gilpress/2016/03/23/data-preparation-most-time-consuming-least-enjoyable-data-science-task-survey-says.

[Ren19] Emre Rençberoğlu.
Fundamental Techniques of Feature Engineering for Machine Learning. Apr. 1, 2019. URL: https://towardsdatascience.com/-3a5e293a5114.

[Ros16] Ethan Rosenthal. *Learning to Rank Sketchfab Models with LightFM*. 2016. URL: `https://www.ethanrosenthal.com/2016/11/07/implicit-mf-part-2/`.

A Appendix

If you physically hold this thesis, you can find a 16GB USB stick on this final page.

It contains a digital copy including all needed project files, created at January 28, 2021 and

in trust given to the the 1st and the 2nd examiner.